Talking To The Database In A Semantically Rich Way

Henrietta Dombrovskaya
Enova
200 W. Jackson Blvd.
Chicago, IL 60606

hdombrovskaya@enova.com

Richard Lee
Enova
200 W. Jackson Blvd.
Chicago, IL 60606

rlee@enova.com

ABSTRACT

Conventional recommendations for Object Oriented application design include the concept of Object-Relational Mapping and suggest clear separation of business logic from interaction with the database. While these requirements seem natural to application developers, it prevents them from using the full power of the database engine, and thereby become the most essential source of application performance degradation. Acknowledging the widespread usage of the above concepts, our approach provides an algorithm for "splitting" logic between different layers of classes. We identify the parts of logic that are essential for data retrieval and thereby belong to the database, and the parts of logic that drive the computation or other data transformation and can reside in the application model. Although the splitting logic algorithm, as yet, is not implemented in any tool, we consider it an important part of the application design process. In our paper we provide examples of redesigned methods as well as before-and-after performance data from the production system.

General Terms

Performance, Design, Human Factors.

Keywords

Application development, object-relational impedance mismatch, code analysis, code factoring, delinquent coding patterns

1. INTRODUCTION

Everybody wants applications to function efficiently. When it comes to the database application, the most important way to reach this goal is to ensure that the application interacts efficiently with the database, since this interaction is the most time consuming portion of the application activity. After all – why choose to use databases in the first place? Why not develop the data accessing tools along with your application? The reason is that the DBMS is specialized software designed to manage data in the most efficient way. Nevertheless, the most common complaint of the application developers is "the database is slow".

What is the reason for this complaint and what are the areas a database developer would usually start looking at? Traditional database optimization is targeted towards optimization of stand-alone queries, including query rewriting and database schema changes on one hand, and tuning of the database system parameters on the other [1-4]. Application and database developers have a great variety of tools available, which help with these kinds of optimization, including DBMS–specific tools [5,6] along with third-party products [7].

However, in many cases the reason for poor performance is "too many too tiny queries", which consume a significant portion of database resources. Recently, a growing number of database researchers and practitioners have started to target this area. The general direction of research is to find the ways of restructuring of the application code that interacts with the database; in effort to reduce the total number of database calls. The perfect example of this approach is the AppSleuth tool, described in [8]. The fact that application code often suffers from the "delinquent patterns", most commonly the repeated processing of a single record in a loop, is evident and hard to ignore. However, the frequency of the problem is most often attributed to the fact that "that is how the programmers are taught to write programs at school". We argue that the problem has deeper roots, and therefore it won't vanish if we just teach people "how to code properly".

We believe that the reason for this type of programming, in most cases, is the problem commonly referred to as "impedance mismatch" [9,10]. The problem has been known since the mid 80's when the first database programming languages started to appear. However, when the object-oriented approach to application design and development became the dominant trend, the problem became apparent. Conventional recommendations on the object-oriented application design [11] suggest clear separation in the levels of classes. Best programming practices favor the separation of classes into different levels: those that implement the business logic and those that interact with the database, in particular. While these requirements seem natural to application developers, they prevent using the full power of the database engine.

With the emergence of Object Oriented (OO) programming and design, many software developers tried to simplify the ways in which applications interact with the database. The most commonly used approach is Object-Relational Mapping (ORM) [12]. This approach maps a database object to the in-memory application object. While solving the problem of abstraction from

10.5441/002/edbt.2014.74

details of data storing, it does not provide an effective means of manipulation of data sets, thus does not solve the impedance mismatch problem.

In the case of ORMs, the problem became known as object-relational impedance mismatch (ORIM) [13]. It is commonly defined as a set of conceptual and technical difficulties that is encountered when a relational database management system (RDBMS) is being used by a program written in an object-oriented programming language or style, particularly apparent when objects, or class definitions, are mapped in a straightforward way to database tables or relational schemata.

One of the reasons for ignoring the problem is the widespread concept of "performance not being an issue". In present development practices the developers time is considered to be the most expensive part of the application development process, and a common approach is that if a development methodology allows us to produce correct results fast enough, the performance issues can be resolved by adding more hardware: more (and faster) CPUs, faster disks, etc. Unfortunately, such an approach masks performance problems for an extended period of time, but with growing data volumes the problem re-emerges sooner or later, and when it re-emerges, it is far more difficult to remedy. Most often, by the time the organization reaches the hardware limits, it is close to impossible to restructure the application code because the system has already been in production for quite a while.

As infrequently as this problem is addressed, even less has been done to propose a solid solution that would remain in the boundaries of OO frameworks. In many cases, when application performance becomes a critical issue, developers tend to introduce a shortcut and execute embedded SELECT statements outside of the model. These SELECT statements are not organized in a class structure, and quickly spread all over the application making simultaneous changes to the query logic impossible.

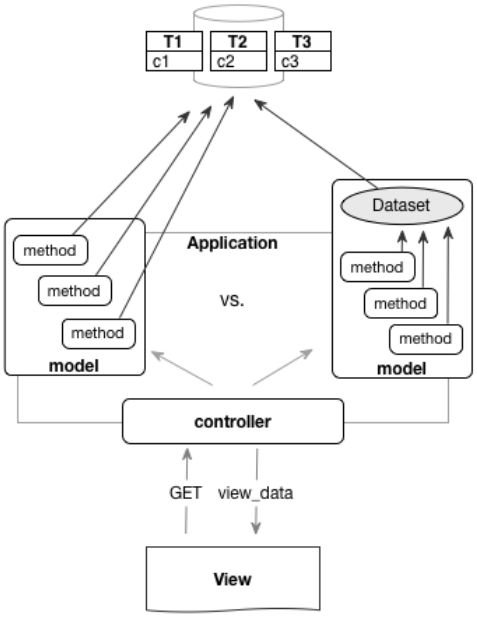

Figure 1. The sketch of proposed approach

In this paper, we propose an application code design and refactoring methodology, which combines the power of the database search engine with the application model logic. The main idea of our approach is illustrated on Figure 1. This methodology is easy enough to be used by application developers, with minimal changes to their standard process, and provides enough performance gain so that they will be willing to accept the change. It also works well when we need to make changes to an existing application, in which case the existing top-level methods' interfaces must remain unchanged.

The preliminary version of this work was first presented at PG Open 2013 Conference in Chicago, IL September 16-18 2013 [14], the abstract and the presentation video are available at the conference website.

The rest of this paper is organized as follows: in section 2 we give a formal description of our methodology. In section 3 we present a simple case study. We describe Enova's application environment and the original structure of methods. Then we show how we applied the new methodology to refactor this method, and how this rewriting helped both to make the code clearer and improve the overall system performance. In section 4 we present more complex example of method rewriting based on the same principles. In section 5 we discuss execution statistics. In section 6 we review related work and outline the differences of our approach. In section 7 we summarize results and discuss future work.

2. PROPOSED METHODOLOGY

In this section we present a description of the principles and guidelines we use to separate the application logic into "layers".

2.1 Behind a Screen Rendering

This project was initiated after a group of application and database developers in our organization decided to take a closer look at the application logs, in order to find out "why the application is slow". We were fully aware of the fact that the application is written in an imperative way, and that the number of database calls is more than optimal. As it turned out, we were not aware of the magnitude of the problem.

For an OLTP application a "logical unit" of application code is a "screen rendering" – a collection of methods that are invoked when an application user clicks a button on the screen (or presses "Enter"). No additional information is passed to the application between the moment of clicking the button and the moment the screen is rendered; which means that, at the moment the screen rendering starts, the application has enough information to retrieve all necessary information from the database. However, our analysis of the application logs showed that in some cases the application produces up to a 1,000 database calls per screen rendering.

Why does it happen? The reason is not that application developers do not know how to code, but the fact that they code in compliance with OO development standards, which inevitably lead to Object-Relational Impedance Mismatch. Typically the smallest programming unit considered in the OO environment is a method (which belongs to some class). Each method is designed to perform some specific task. For example: calculate outstanding balance for a loan.

What happens when an application, developed in line with the OO principles, renders a screen view? Since elements of the screen may belong to different classes and obtaining their values may require using different methods, this means that an application is

unable to retrieve all the necessary data "in one shot". Instead, it will submit queries to a database "as needed" thus treating the database like main memory. This results in a large number of database calls and, while each single SELECT execution may be very fast, total execution time may significantly increase application latency [15, 16]. In our case, with average execution time for each query as low as 20-50 milliseconds, a screen rendering could take 30 or more seconds, which would result in application timeouts. This observation prompted us to look for alternative ways of method structuring and resulted in this methodology, which we named Logic Split.

2.2 How the Logic Split Works

We start our logic split process for a specific screen rendering (or, more precisely, for a specific controller action execution) by building a tree of methods, which are called to populate all fields in the screen. The sample screen view is presented on Figure 2.

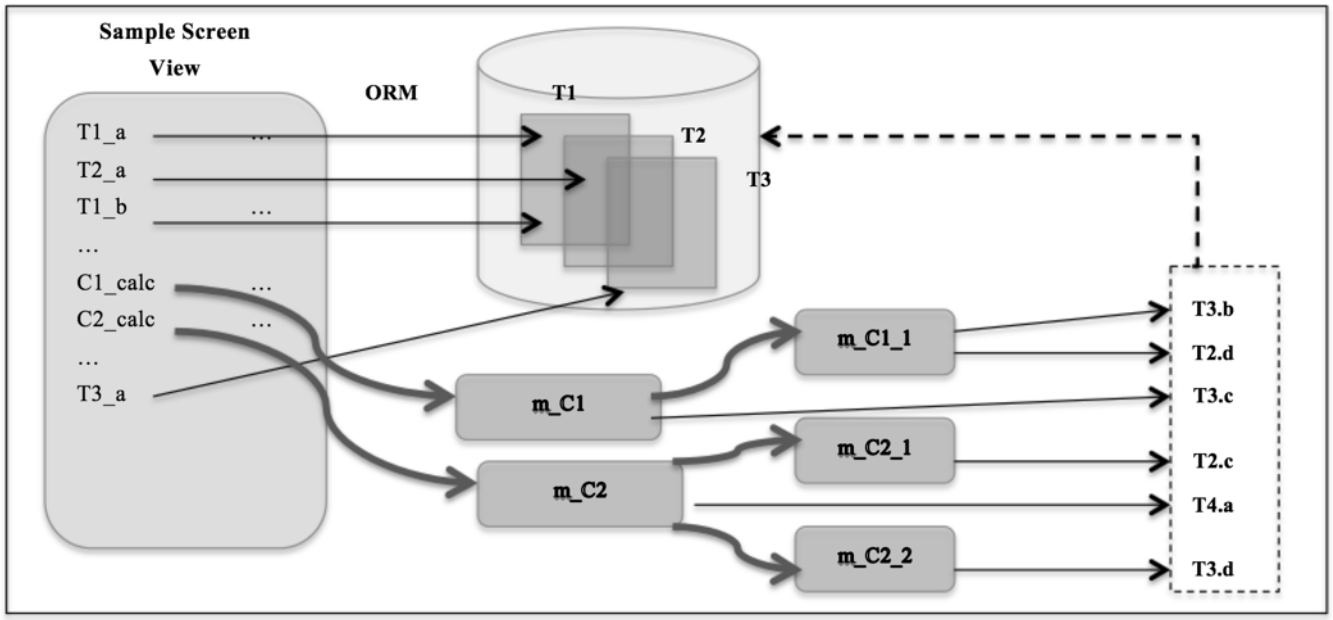

Figure 2. Splitting logic schema

Since we are using the ORM framework, some of the attributes are mapped directly to database table fields and do not require additional method execution. However, we want to make sure that we retrieve the data from all tables using one SELECT statement, not one SELECT statement per table; or worse: one SELECT statement per attribute. The ORM model defines relationships between classes of objects as joins between tables, which we can utilize (similar to "eager loading"[17], but selecting only the fields we need, not the whole table). This mapping is presented in Figure 2 by links between the screen attributes T1_a, T1_b, T2_a and T3_a and tables T1, T2 and T3.

Other attributes on the screen are populated by *methods,* which are designed according the OO guidelines. In Figure 2 these attributes are C1_calc and C2_calc. The first, C1_calc, is computed by m_C1 method, the second C2_calc, by m_C2 method.

Until this moment our methodology is not significantly different from the standard ORM approach, according to which the application would execute each of the methods and populate the remaining attributes with obtained results. Instead, we start to examine the internal structure of each of the methods called.

Each of the methods can be broken into smaller, more granular tasks; each of those tasks can be disassembled into yet smaller tasks. This way the method structure can be presented as a tree of method calls where the leaves present the low-level "atomic" methods. Some of these tasks may require data collection while others perform calculations based on the previously obtained

```
SELECT  T1_a
       ,T1_b
       ,T2_a
       ,T3_a
FROM T1 INNER JOIN T2
               ON T1.id=T2.T1_id
       INNER JOIN T3
               ON T2.id=T3.T2_id
```

Figure 3. ORM-based SELECT

values. This distinction is important for our methodology but typically, application developers do not take this characteristic into consideration, they just present the "imperative" way of execution and all data sources are considered an "extension of main memory". In contrast, our approach makes this distinction the most important factor in making decisions about the methods' internal structure.

In the example presented on Figure 2 m_C1 method selects attributes b and c from T3 table and calls m_C1_1 method, which in turn selects attribute d from T2 table. The other method (m_C2) calls two methods: m_C2_1 and m_C2_2, which select data from T2 and T3 respectively. The right portion of Figure 2 (enclosed in the dashed rectangle) contains all data elements, which need to be retrieved from the database. Since we can obtain this list of

elements before any method execution is started, we can add them to the SELECT statement, shown on Figure 3.

The complete SELECT is presented on Figure 4.

```
SELECT  T1_a
       ,T1_b
       ,T2_a
       ,T3_a
       ,T3.b
       ,T2.d
       ,T3.c
       ,T2.c
       ,T4.a
       ,T3.d
FROM T1 INNER JOIN T2
              ON T1.id=T2.T1_id
       INNER JOIN T3
              ON T2.id=T3.T2_id
       INNER JOIN T4
```

Figure 4. Complete SELECT statement

Now we can create a leaf-level method, which will retrieve all data elements from the database, wrapping into a method the SELECT statement above. When all the values from the database are retrieved and reside in the application memory, the remaining methods can use them as variables. Going in the reverse order, m_C2_1 method uses T2.c value, m_C2_2 method uses T3.d value, and they pass their execution results to m_C2 method, which in addition utilizes T4.a value, and then continue to the root methods.

Note the there are no changes to the logic of original methods, we only refactor them to use extracted data as input parameters.

2.3 Defining a Division Line

The methodology described in the previous section seems rather obvious but this type of analysis is not something that is typically performed during OO development. This happens because when a database object is mapped on the main memory object, the application programmer considers all data "equally reachable" and does not care about order of access of these objects.

When we construct the lower level SELECT statement, we need to be aware of the ways the objects are associated with one another, so that related objects can be identified. Which means that, along with identifying the data elements, we are identifying their relationships.

This is the part of the logic we are taking away from the model, which is exactly the piece of logic that allows us to write complex queries and enable database optimization.

By applying our method of Logic Split we achieve two separate goals. First, we make the method logic more visible; the code becomes shorter and easier to understand. Second, we can avoid execution of a large number of small database queries and utilize the database optimizer. As an extra bonus, any database structural changes that occur, due to the need to support data volume growth or new requirements, become "invisible" to the application and do not require any upper level methods changes. In short, we allow optimization of application code and database query optimization to be performed independently.

To summarize, the Logic Split methodology consists of the following five steps:

1. Disassemble method into atomic steps
2. Identify ones which require data retrieval
3. Using knowledge about database objects relationships, construct a single query
4. Execute
5. Use retrieved data in other steps

3. CASE STUDY: CALCULATION OF OUTSTANDING LOAN AMOUNTS

3.1 Enova Environment

Enova is a Ruby on Rails shop that uses the ActiveRecord Object Relational Mapping (ORM) library to communicate with a Postgres database. ActiveRecord is named after the 'active record' pattern defined by Martin Fowler in his book, *Patterns of Enterprise Application Architecture*[8]. This pattern is an approach to accessing and manipulating data in a database within an object oriented system by providing the translations and tools for interaction between the objects defined in the system and the tables of records in the database. With ActiveRecord a database table or view is mapped into a class and an object instance is tied to a single row in the table.

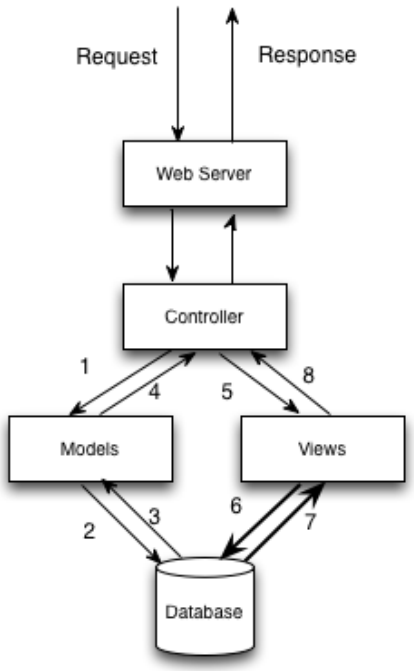

Figure 5. Interaction between the website and the database

The ActiveRecord library creates a persistent domain model from business objects and database tables, where logic and data are presented as a unified package. ActiveRecord adds inheritance and associations to the pattern above, solving two substantial limitations of that pattern. A set of macros acts as a domain language for the latter, and the Single Table Inheritance pattern is integrated for the former; thus, ActiveRecord increases the functionality of the active record pattern approach to database interaction. ActiveRecord is the default "model" component of the

model-view-controller web-application framework Ruby on Rails, and is also a stand-alone ORM package for other Ruby applications. The interaction between the application and the database is presented on Figure 5.

Due to the lack of awareness by methods of the underlying interaction with the database, one controller performs multiple trips to the database. For example, when an application needs to retrieve a loan summary, it would execute the sequence of calls presented in Figure 6.

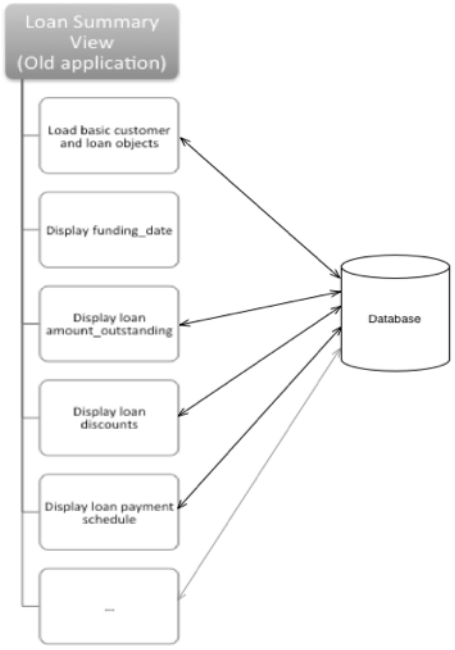

Figure 6. Methods to display loan summary view

3.2 Choosing the Test Case

One of the statements that caught our attention during the preliminary performance analysis was the SELECT shown in Figure 7. The screenshot presents a part of the log with the list of "top 100 offenders", the queries that take the most total execution time throughout the day. The first column is the sequential number of the query in the list; the second the total execution time of all occurrences of this query during the day; the third, the number of executions; the forth, the average execution time of the individual query. This fragment of the log shows that the second and third position in the list of top offenders is occupied by two different versions of one query.

These two SELECT statements have an average execution time of 20 to 40 milliseconds, but are executed about 700,000 times during a 24 hour period, bringing their total execution time to about 5 hours and putting unnecessary load on the database.

This observation prompted us to find the method which was executing this SELECT statements and choose it as our test case. This test case illustrates only a subset of our methodology. Specifically, we illustrate rewriting of one of the methods, which is used to compute a calculated field on the screen (like m_C1 method on Figure 2). In Section 4 we present an example of a more generic case and then show how the Logic Split methodology is applied to the whole screen view rendering.

3.3 Original Method Description

The first method we selected for optimization is called `amount_outstanding`, and appears to be one of the most often executed methods.

It takes a loan number as an input parameter, along with some optional parameters, and calculates the outstanding amount for this loan. The UML diagram of the original method is presented on Figure 8.

#	Time	Executions	Avg Time	Query
2	**3h05m12s**	524,027	21.21	SELECT vl.value AS account, SUM(CASE vl.value WHEN pt.debit_account_cd THEN pt.amount ELSE 0 END) - SUM(CASE SUM(CASE vl.value WHEN pt.debit_account_cd THEN pt.amount ELSE 0 END) AS debit, SUM(CASE vl.value WHEN pt.credit_account_cd THEN pt.amount ELSE 0 END) AS credit FROM payment_transactions_committed pt INNER JOIN valuelists vl ON vl.type_cd = '' AND vl.value IN (pt.debit_account_cd, pt WHERE (loan_id = 0 AND installment_id = 0 AND committed = true) GROUP BY vl.v Show examples
3	**1h55m29s**	166,544	41.61	SELECT vl.value AS account, SUM(CASE vl.value WHEN pt.debit_account_cd THEN pt.amount ELSE 0 END) - SUM(CASE SUM(CASE vl.value WHEN pt.debit_account_cd THEN pt.amount ELSE 0 END) AS debit, SUM(CASE vl.value WHEN pt.credit_account_cd THEN pt.amount ELSE 0 END) AS credit FROM payment_transactions_committed pt INNER JOIN valuelists vl ON vl.type_cd = '' AND vl.value IN (pt.debit_account_cd, pt WHERE (loan_id = 0 AND committed = true) GROUP BY vl.value; Show examples

Figure 7. Database execution log report

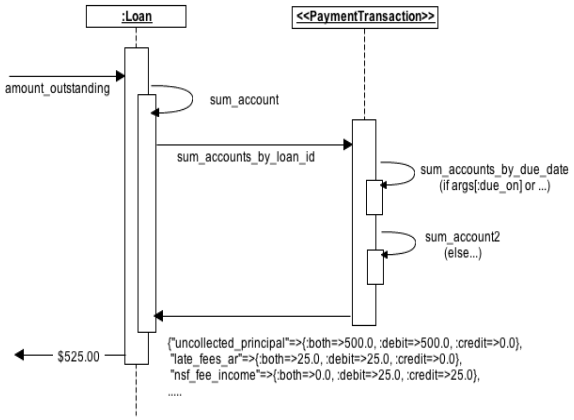

Figure 8. UML diagram of original amount_outstanding method

The `amount_outstanding` method summarizes balances of several accounts in order to get the value for each of the balances. This is done using a call to the `sum_account_by_loan_id` method. Which, in turn calls a sequence of either `sum_account` or `sum_account2` methods, depending on whether the due date is specified. These two methods access the database and perform SELECT statements. Note that two separate SELECTs are maintained in two separate classes, and there is no guarantee they will be updated simultaneously should a change will be required.

Depending on the input parameters, one of several Ruby methods, which directly interact with the database, is executed. Each executes one of the following SELECT statements:

```
SELECT
 vl.value AS account
,SUM(CASE vl.value WHEN pt.debit_account_cd
                   THEN pt.amount ELSE 0 END)
- SUM(CASE vl.value WHEN pt.credit_account_cd
                    THEN pt.amount ELSE 0 END)
       AS sum
FROM payment_transactions pt
       JOIN valuelists vl ON vl.type_cd
       ='transaction_account'
        AND vl.value IN (pt.debit_account_cd
       ,pt.credit_account_cd)
       AND loan_id=?
```

Or:

```
SELECT  vl.value AS account
     ,(SUM(CASE WHEN pt.debit_account_cd =
  vl.value AND
          (debit_account_due_date <= '{?}'
             OR debit_account_due_date IS NULL)
            THEN pt.amount ELSE 0 END)
     - SUM(CASE WHEN pt.credit_account_cd =
     vl.value AND
          (credit_account_due_date <= '{?}'
     OR credit_account_due_date IS NULL)
THEN pt.amount ELSE 0 END)) as sum
FROM payment_transactions_committed pt
JOIN valuelists vl ON vl.type_cd =
'transaction_account'
AND vl.value IN (pt.debit_account_cd
,pt.credit_account_cd)
```

These are the SELECT statements that we observed in the list of "top offenders".

3.4 Drawbacks of Existing Method

Technically the method described in the previous subsection would allow retrieving all the information related to one loan "in one shot". However, due to the application developers being unaware of the underlying levels, there appears to be no difference in whether to obtain the values of all accounts balances one by one through following the logic of the method (in an imperative way) or to obtain them simultaneously.

At the top level the business logic defines the components of the "outstanding amount" for a loan, which are defined in the model the following way:

```
AccountsOutstanding=
                  AccountsUncollected +
                   FeesOutstanding +
                   InterestOutstanding +
           PrincipalAccounts +
                   AccountsDue
```

Each of them is defined as an aggregate of a set of "atomic" accounts for example:

```
AccountsUncollected =
         uncollected_principal +
         uncollected_installment_principal
```

According to these definitions the same SELECT statement is executed multiple times, and each time only one atomic account is selected. This explains, why one single execution of `amount_outstanding` method produced multiple database calls.

3.5 Utilizing the Logic Split

In Figure 9 we present a new UML diagram of `amount_outstanding_by_loan_id` method. In the modified method we isolated SQL parts, and instead of calling several SELECT statements in different parts of the method, call it in just one place.

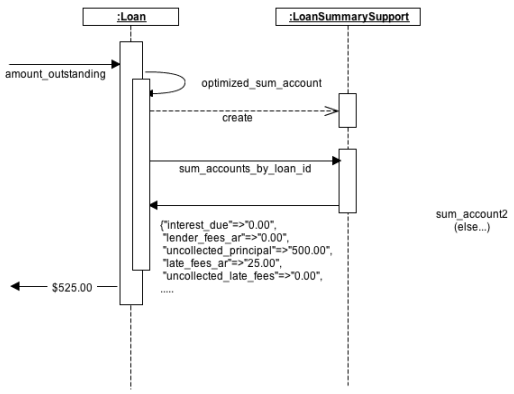

Figure 9. UML diagram of modified amount_outstanding method

Specifically, the optimized_amount_outstanding_by_loan_id method calls the sum_accounts_by_loan_id optimized method, which invokes the execution of a single PostgreSQL function, generating and executing one single SELECT statement.

The particular SELECT statement is generated based on the method parameters. The important part is that now there is no need for special precautions regarding keeping all SELECT statements in sync.

Now the SELECT statement, which is generated to retrieve all outstanding balances, looks like this:

```
SELECT  loan_id
 ,sum(CASE WHEN debit_account_cd =
'uncollected_principal' THEN pt.amount
       ELSE 0  END
-CASE WHEN credit_account_cd =
'uncollected_principal' THEN pt.amount
       ELSE 0  END) AS uncollected_principal
  <...>
  , sum(CASE WHEN debit_account_cd =
  'uncollected_nsf_fees'
                 THEN pt.amount ELSE 0  END
      - CASE WHEN credit_account_cd =
      'uncollected_nsf_fees'
             THEN pt.amount ELSE 0  END) AS
      uncollected_nsf_fees
      , sum(CASE WHEN debit_account_cd =
'installment_principal'
                   THEN pt.amount ELSE 0  END
       -   CASE WHEN credit_account_cd =
           'installment_principal'
       -    THEN pt.amount ELSE 0  END) AS
           installment_principal
  FROM payment_transactions_committed pt
            INNER JOIN loans l ON
l.id=pt.loan_id
       WHERE customer_id={?}
       GROUP BY loan_id
```

The due_on date, along with some other parameters, can be added during SELECT generation.

We also added one more parameter, the customer number. If application developers want to use this method the way they did before, they can continue to do so. However, if they want to write more efficient code, they can pass the customer number as a parameter. In this way, they can retrieve outstanding balances for all loans for a specified customer, with no additional database load.

4. MORE COMPLEX CASE STUDY AND CALLBACKS

4.1 Task Description

The methodology described in Section 2 is based entirely on the maximal efficiency of the resulting implementation. In this section we present a more complex example of logic split between the database and the application code. We also introduce an additional feature of our methodology that allows reprocessing some of the data selected from the database and, in Ruby code, return a callback result into the dataset.

One of the methods we implemented using our new methodology returns, as a result, the account balance of a line of credit. It includes some complex calculations that, for compliance reasons, had to be performed in the model, rather than in the database.

To implement the Account_Balance method, we first listed the upper-level steps. They included:

1. Obtain account principal balance
2. Obtain outstanding fees and interest as of next payment due date
3. Calculate interest credit (unearned interest) for number of days left before payment due date
4. Obtain existing customer balance
5. Calculate total account balance using values obtained in steps 1-4

4.2 Utilizing the Logic Split

If we would utilize a traditional OO programming approach we would write an Account_Balance method, which would call Principal_Balance, Interest_Amount, Fees_Amount, Customer_Balance, and Interest_Credit methods. Each of these methods would interact with the database independently. Instead, we continued drilling down into each of the steps:

1. Principal balance can be computed the same way as described in item 3, so this is an atomic operation that can be executed with a single database call
2. Outstanding interest and fees can also be obtained using one database call each
3. Calculation of the interest credit involves several steps:
    3.1. Obtain daily interest rate for this customer
    3.2. Obtain base amount, which is used to calculate total interest
    3.3. Obtain number of days for which interest should be credited
    3.4. Calculate amount of credit, based on results from previous three steps
4. Customer balance can be obtained using one database call, same as steps 1-3

Step 3.3, in turn can be separated into the following steps:

3.3.1. Obtain next payment due date

3.3.2. Calculate number of days based on obtained date and today's date

Now that we have disassembled the steps into atomic steps, we can combine together the ones that deal with data retrieval. These steps are: 1, 2, 3, 3.1, 3.2, 3.3.1, and 4, which means that the database access method should execute the following task:

> *For a given loan, retrieve payment transactions, which show principal balance, current interest, fees and customer balance, also the loan's daily interest rate and next payment due date*.

This task can be relatively easily executed using just one SELECT statement:

```
SELECT l.id AS loan_id
         ,sum( CASE WHEN debit_account_cd =
'principal'
                  AND t.acct_date<=
v_current_date
                 THEN t.amount  ELSE 0  END
                      - CASE WHEN
credit_account_cd  ='principal'
               AND t.acct_date<= v_current_date
```

```
              THEN t.amount ELSE 0  END  )  AS
          amount_payable
        ,sum (CASE WHEN
t.debit_account_cd=’fees_provisional’
                              THEN t.amount ELSE
0 END
        - CASE WHEN t.credit_account_cd=
        ‘fees_provisional’
                      THEN t.amount ELSE 0 END )
         AS fees_provisional
        ,sum (CASE WHEN t.debit_account_cd=’
interest_provisional’
                               THEN t.amount
ELSE 0 END
        - CASE WHEN t.credit_account_cd
        =’interest_provisional’
                      THEN t.amount ELSE 0 END )
        AS interest_provisional
       ,st.end_date AS next_closing_date
      ,l.daily_rate AS interest_rate
      ,sum (CASE WHEN t.debit_account_cd
='customer_balance'
                         THEN -t.amount ELSE 0
END
       - CASE WHEN t.credit_account_cd
      ='customer_balance
                  THEN -t.amount ELSE 0 END )
            AS customer_balance
      FROM loans l
      LEFT OUTER JOIN
      payment_transactions_committed t ON
      l.id=t.loan_id
          LEFT OUTER JOIN statements st ON
                                l.id=st.loan_id
                WHERE l.id={?}
             GROUP BY l.id
                      ,l.daily_rate
                      ,st.end_date
```

This SELECT statement, when executed as a part of the method invoked from the application, delivers all the data at once. Then, the Ruby method calculates the final account balance amount:

```
def account_balance
    amount_payable = amount_outstanding +
amount_charged_off
    amt = amount_payable - customer_balance
    provisional_interest = interest_provisional
    provisional_fees = fees_provisional
    if provisional_interest > 0.0
      amt = amt + provisional_fees + [
provisional_interest - [unearned_interest,
0].max, 0 ].max
    else
      amt = amt + provisional_fees
    end
    amt = amt.round_near
    return [0, amt].max
  end

  def unearned_interest
    amt = -1 * (principal_amount +
[(customer_balance - interest_provisional -
fees_provisional), 0].max)
    next_closing_date =
Date.parse(self.next_closing_date)
    return 0 unless next_closing_date
    interest_period = [next_closing_date -
Date.today + 1, 0].max
    interest_rate = oec_daily_rate.to_f
    interest = amt * interest_period *
interest_rate
    interest = interest.round_down
    interest < 0.0 ? 0 : interest
  end
```

4.3 Usage in Screen Rendering (Callback)

In our application this methods is used as a callback in a more complex method. The important thing is to apply the logic split concept consistently. We established that, for the purposes of efficiency and compliance, we want all of the calculations to be executed in the Ruby model. So, in the case where we need a “virtual column” as one of the fields of the data output, a Ruby callback can be inserted into the data set.

In our example the Account_Balance method is used as a callback in a more complex method, which returns several elements of the loan summary. Most can be selected directly from the database, except for account balance. The callback produces the required value, and the whole data set is passed to the upper-level method for further processing, then to the web application. Note that all data elements can still be retrieved with a single SELECT, which would not be possible within the standard ORM framework.

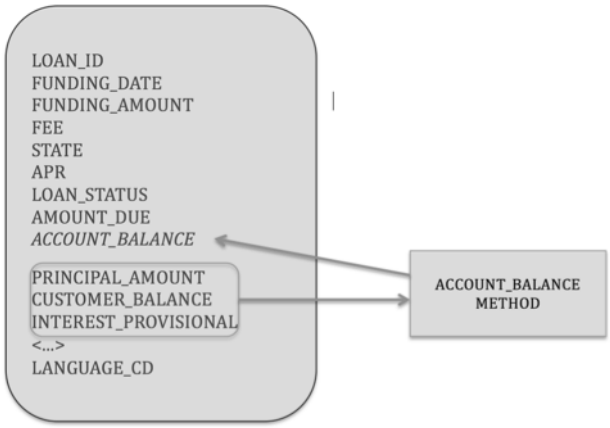

Figure 10. Using Account_Balance as a Callback

5. EXECUTION STATISTICS FOR NEW METHODS

When we modified existing methods utilizing our “logic split” methodology, we had to perform extended testing to make sure the new methods would produce the same results. This is especially important when money is involved. For most of the new methods we conduct extended “dark testing”: for a predefined percent of executions old and new methods are executed simalteneously, and when the produced results are different, the difference is logged. We usually run such testing for a couple of weeks, and for each of the discrepancies we analyze it’s cause. In some cases we where able to identify the problems with new method implementation, but in the other cases the business users would confirm, that new methods are more accurate. Since our modified methods are more declarative, they

would allow us to avoid some issues, which resulted from the imperative nature of original methods.

The same "dark testing" allowed us to obtain some valuable execution statistics. When we started to test our first new method, described in Section 3, we performed full parellel testing on a smaller production database where we could afford the additional load. Both old and new methods were always executed synchroniously to make certain the inputs and outputs would remain identical. This allowed us to measure the advantage of the new method in terms of database load.

Figures 11 and 12 repersent the parts of the hourly execution logs of a smaller database, which show the difference in the total execution time:

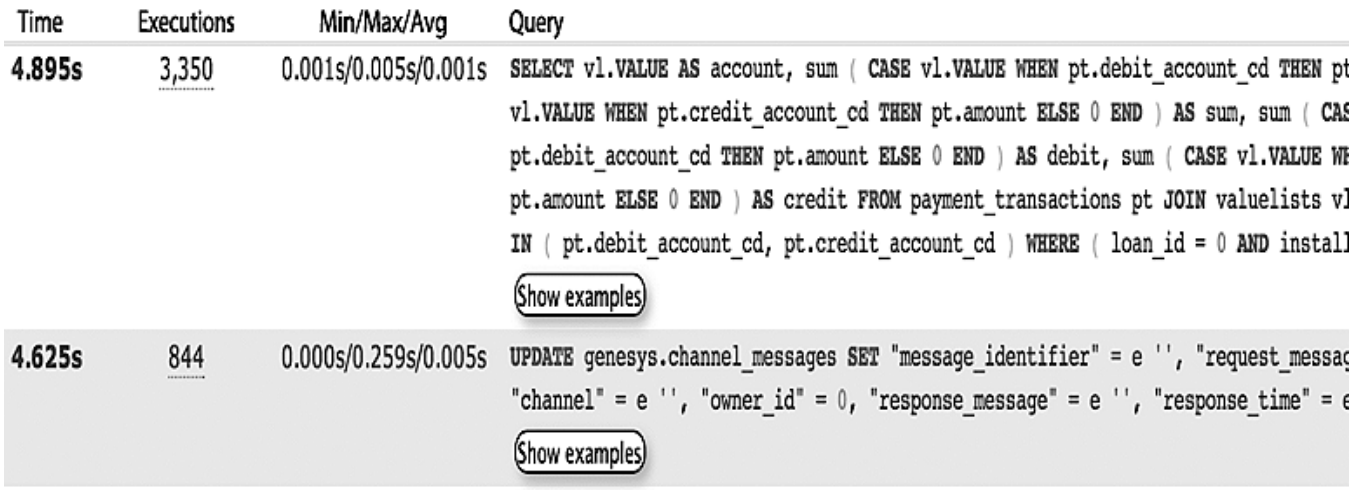

Figure 11. Execution of the select statement from the original account_outstanding method

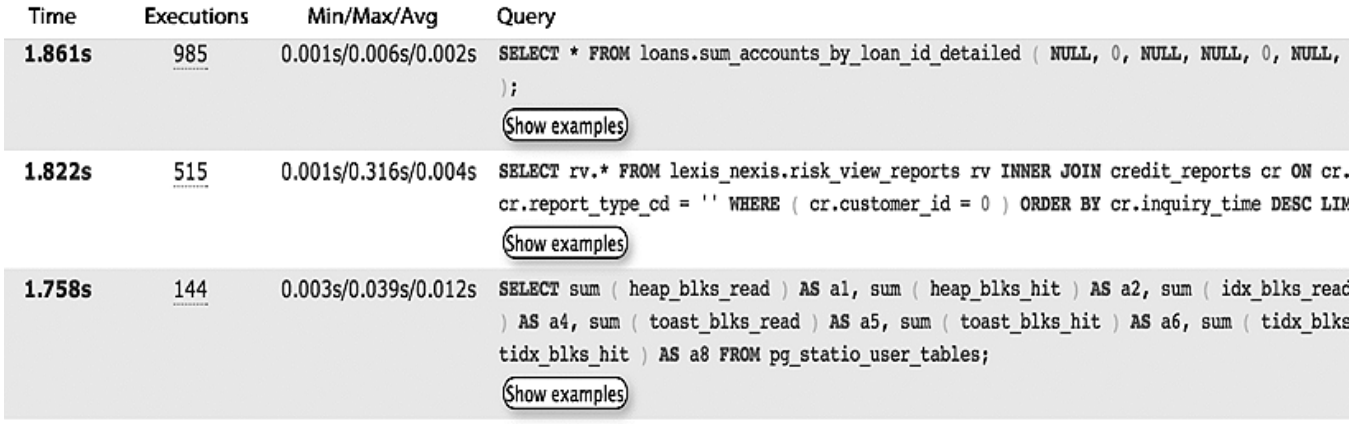

Figure 12. Execution of the select statements from the optimized account_outstanding method

These two screenshots show us that the optimized method has improved efficiency, not because the SELECT execution is faster, since – it is actually slightly slower - on average, but because the number of executed SELECT statements is reduced.

We also compared the number of SELECT executions per method invocation using our internal statistics-gathering program and obtained similar results. We are getting approximately half of the original number of database calls with the optimized method. Note that the difference is not as large as it could be because our application is caching the results of SELECT statements.

After our success with the simple rewriting, we proceeded with implementing the logic split on a larger scale. We developed a couple of classes, which used one or several database calls per screen rendering. Specifically, we developed a method that returns summaries of all loans for which a customer has applied. We then ran our internal statistics gathering scripts to see the difference in execution.

Table 1 presents the average count of database queries and average total execution time required to retrieve loan summary information for all loans for a specified customer. This comparison shows that, in the old version of the application, both

of these numbers depend on a total number of loans per customer. However, for the new version it takes approximately the same time regardless of the number of loans. The graphs, presenting this analysis are shown on Figures 13 and 14.

Table 1. Number of DB calls and DB time per customer based on #loans

# loans/ cust	New App avg_calls	New App avg_time	Old App avg_calls	Old App avg_time
1	4	0.685	121	1.962
2	4	0.399	196	2.72
3	4	0.435	373	5.227
4	4	0.455	449	6.001
5	4	0.393	632	4.994
6	4	0.464	765	5.624
7	4	0.462	819	6.87
8	4	0.458	923	6.52
9	4	0.481	1129	8.932

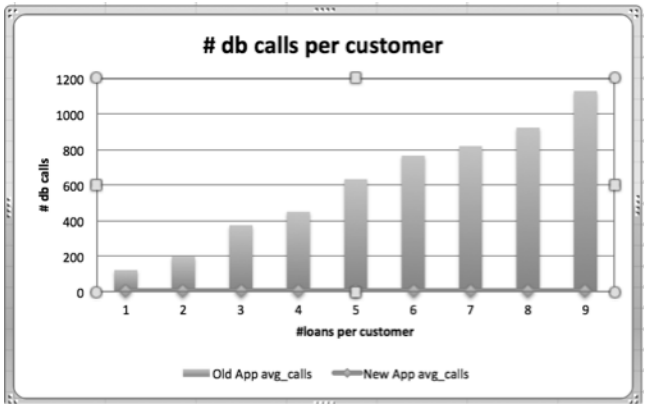

Figure 13. Average # of DB calls per customer

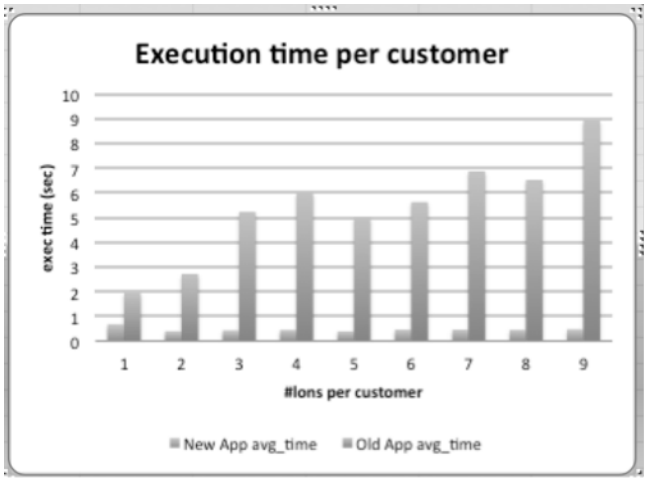

Figure 14. Average DB time per customer

Table 2 presents some production statistical data comparing the execution of some controllers of old and new applications, which perform similar functions. Note that these are not test run results but actual production execution statistics over a 24 hour period. We run statistical reports daily to ensure the new application is performing well and to watch for the next possible areas of improvement. The old application is still used by some of the customer service representatives.

Table 2. Execution Statistics: Old App vs. New App by Controller

Controller action	Old Avg # DB calls	New Avg # DB calls	Old Avg Time (sec)	New Avg Time (sec)
Customer Summary	167	39	1.08	0.19
Loan Summary	506	50	4.5	0.44
Loan Payments	36	3	0.11	0.04
Installments	130	3	0.72	0.018

These statistics show that, even when the methods are not completely optimized, applying the Logic Split consistently gives us a significant performance improvement.

6. RELATED WORK

The problem of object-relational impedance mismatch is a constant discussion topic when it comes to developing an efficient application. In recent years multiple attempts were made to try to resolve this issue. Many authors indicate that the ORIM is not a single problem, but rather identify different *types* of ORIM. For example, Ireland et al [18, 19] identify conceptual, representation, emphasis, and instance impedance mismatches.

The Hybrid Object-Relational Architecture (HORA) approach was first introduced in 1993 [20] and became a foundation for multiple ORM-based systems; to name a few: Java Persistence [21], ActiveJDBC [22], ADO.NET [23], and Ruby on Rails ActiveRecord.

Hybernate [24] is a high-performance Object/Relational persistence and query service. It is considered one of the most flexible and powerful Object/Relational solutions on the market. It takes care of the mapping from Java classes to database tables and from Java data types to SQL data types. Hybernate definitely can be credited for significantly reducing development time, allowing application developers to concentrate on the business side of the project.

The Hybernate developers claim that, in contrast to other solutions, it does not hide “the power of SQL” from developers. This claim is true in some sense since the solution indeed allows us to write queries similar to SQL queries. However, creation of complex queries using Hybernate is not an easy task. Similar to other ORM systems, Hybernate prompts for solutions that seem more natural for application developers.

Agile Data [25] technology acknowledges the existence of ORIM and, what is more important, it also acknowledges a cultural impedance mismatch, citing the major difference as: “The object-oriented paradigm is based on proven software engineering principles. The relational paradigm, however, is based on proven mathematical principles.” While raising awareness of the problem, the solutions proposed by Agile Data technology refer mostly to database schema changes and/or more careful design

and data refactoring; it offers little in terms of dealing with inefficient queries.

The AppSlueth [8] tool for application tuning is designed to achieve a goal very similar to what we are trying to achieve. It parses the application code and identifies "delinquent design patterns", such as fetching and/or processing one record at a time. The tool helps to identify critical pieces of the code, which require rewriting, but in general does not allow to stay within ORM, or to reuse existing methods the way we are trying to reuse them with the Logic Split methodology. This paper also provides a comprehensive overview of different research in the field of optimization.

SQLAlchemy [26] considers the database to be a relational algebra engine, not just a collection of tables. Rows can be selected from not only tables but also joins and other select statements. Any of these units can be composed into a larger structure. SQLAlchemy's expression language builds on this concept from its core.

SQLAlchemy is most famous for its object-relational mapper (ORM), an optional component, which provides the data mapper pattern where classes can be mapped to the database in open ended, multiple ways; allowing the object model and database schema to develop in a cleanly decoupled way from the beginning.

The only problem with this tool is that it implies that an application developer is at the same time a database developer, is aware of the best data access paths, and can divert from the OO design standards.

The DBridge [27] project explores different methods of holistic application optimization, including analysis of the source code, suggesting some improvements of its structure. These possible changes include pre-fetching and "loop splitting". The latter technique is similar to our logic split methodology but is limited to only changing loop processing from "one by one" execution to "batch" execution. Although it allows IF THEN ELSE constructs within the loop and the nested loop, it does not appear to be able to process loops which manifest themselves due to calling methods from different classes.

7. CONCLUSION AND FUTURE WORK

After we had several successful use cases of our new methodology, we pursued a larger task: rewriting a whole application using our new methodology for splitting logic between an application and a database.

At this time we are in the middle of this process. We are developing a new version of one of our legacy applications; carefully assembling new pieces while preserving the existing functionality.

One of the most important problems we are facing is the large amount of legacy code where most of the business logic is embedded in Ruby classes. There are virtually no business specifications, which means that we have to extract the business logic from the existing code. On the other hand, the legacy application is evolving and existing models are being modified all the time. We also need to consider the human factor, i.e. to take into account the current development practices.

Another problem started to emerge, when we made an effort to extract the business logic from the existing legacy code. In some cases our approach helped to clear some existing issues, thus making our results "better" than legacy application results. This lead to some compliance issues – all applications should display the same values for the same objects (like account balance or available credit). Which, in turn led to the necessity of closer integration and simultaneous changes in different applications.

Having said this, our future work goes in two different but related directions. First: we continue to rewrite the larger parts of our applications, shooting for having only a couple of database queries per screen rendering. Second: while doing this, we are clarifying our technology, making it more transparent and easier for application developers to use.

8. ACKNOWLEDGEMENTS

The authors would like to thank the Enova Tech Leadership Team and all their co-workers for their continuous support, without which this research and development would not be possible.

9. REFERENCES

[1] Papadomanolakis, S., Dash, D., Ailamaki, A., Efficient use of the query optimizer for automated physical design. *In Proceedings of the 33th International Conference on Very Large Data Bases (VLDB '07)* (University of Vienna, Austria, September 23 – 27, 2007). ACM Press, New York, NY, 2008, pp 1093 – 1104.

[2] Zilio, D., Rao, J., Lightstone, S., Lohman, G., Storm, A. J., Garcia-Arellano, C., and Fadden, S. DB2 Design Advisor: Integrated automatic physical database design. *In Proceedings of the 30th International Conference on Very Large Data Bases (VLDB '04)* (Toronto, Canada, August 31 – September 3, 2004). Morgan Kaufmann, San Francisco, CA, 2004, pp 1110 – 1121.

[3] Agrawal, S., Chaudhuri, S., Narasayya, V. R. Automated selection of materialized views and indexes in SQL databases. *In Proceedings of the 26nd International Conference on Very Large Data Bases (VLDB'00)* (Cairo, Egypt, September 10 – 14, 2000). Morgan Kaufmann, San Francisco, CA, 2000, pp 496 – 505.

[4] Surajit Chaudhuri, Vivek Narasayya, and Manoj Syamala, Bridging the Application and DBMS Profiling Divide for Database Application Developers, *In Proceedings of the 33th International Conference on Very Large Data Bases (VLDB '07)* (University of Vienna, Austria, September 23 – 27, 2007). pp 1252-1262

[5] Oracle Corporation. Performance tuning using the SQL Access Advisor. *Oracle White Paper*. (2007), DOI=http://otn.oracle.com

[6] Microsoft Corporation. SQL Server 2005 books online:Automating administrative tasks. *SQL Server product documentation.* (September 2007), DOI =http://msdn.microsoft.com/enus/library/ms187061(SQL.90) .aspx.

[7] Quest Software. Toad: SQL Tuning, Database Development & Administration Software. (2012), DOI = http://www.quest.com/toad/, 2012

[8] AppSleuth: a Tool for Database Tuning at the Application Level Wei Cao and Dennis Shasha. *In Proceedings of IBDT/ICDT '13*

[9] The Committee for Advanced DBMS Function Corporate. Third-generation Database System Manifesto, *ACM SIGMOD Record*, 1990, vol. 19, no. 3, pp. 31–44

[10] Corporate Act-Net Consortium. The Active Database Management System Manifesto: A Rulebase of ADBMS Features, *ACM SIGMOD Record*, 1996, vol. 25, no. 3, 40–49.

[11] Booch,G., Maksimchuk, R., Engel, M., Young, B., Conallen, J., Houston, K. *Object Oriented Design with Applications,* Addison-Wesley, 2007

[12] Fowler, Martin. *Patterns of enterprise application architecture*. Addison-Wesley, 2003 ISBN 978-0-321-12742-6

[13] Ambler, S., *Agile Database Techniques: Effective Strategies for the Agile Software Developer*, New-York: Wiley, 2003

[14] PG Open 2013 Conference Chicago IL September 16-18 2013; abstract at http://postgresopen.org/2013/schedule/presentations/312/; presentation at http://www.youtube.com/watch?v=hamQIe9YZJw&feature=youtu.be

[15] 4.Shasha, D. and Bonnet, Ph., *Database Tuning: Principles, Experiments, and Troubleshooting Techniques* Morgan Kaufmann, 2002.

[16] Celko, J., *Thinking in Sets: Auxiliary, Temporal, and Virtual Tables in SQL*, The Morgan Kaufmann Series in Data Management Systems, MK, 2008

[17] Ruby on Rails Guide http://guides.rubyonrails.org/active_record_querying.html

[18] Ireland, C., Bowers, D., Newton, M., Waugh, K. A Classification of Object-Relational Impedance Mismatch, *First International Conference on Advances in Databases, Knowledge, and Data Applications DBKDA '09.* (Gosier, Guadeloupe/France, March 01-06, 2009), IEEE, 36- 43. DOI= 10.1109/DBKDA.2009.11

[19] Ireland, C., Bowers, D., Newton, M., Waugh, K. Exploring the Essence of an Object-Relational Impedance Mismatch - A novel technique based on Equivalence in the context of a Framework, *The Third International Conference on Advances in Databases, Knowledge, and Data Applications DBKDA 201*

[20] Sutherland, J., Pope, M., and Rugg, K., The Hybrid Object-Relational Architecture (HORA): An Integration of Object-Oriented and Relational Technology, *Proc. of the 1993 ACM/SIGAPP Symp. on Applied Computing: States of the Art and Practice*, Indianapolis, 1993, 326–333

[21] Java Persistence http://jcp.org/aboutJava/communityprocess/final/jsr317/index.html

[22] ActiveJBDC https://code.google.com/p/activejdbc/

[23] ADO.NET http://msdn.microsoft.com/en-us/library/aa286484.aspx

[24] Hybernate http://www.hibernate.org/about

[25] Agile: http://www.agiledata.org/essays/culturalImpedanceMismatch.html

[26] SQLAlchemy: http://www.sqlalchemy.org/

[27] Dbridge: http://www.cse.iitb.ac.in/dbms/dbridge/