PL/SQL Advisor: a Static Analysis-based Tool to Suggest Improvements for Stored Procedures

Dimas C. Nascimento, Carlos Eduardo Pires, Tiago Massoni

[1] Department of Computer Science – Federal University of Campina Grande (UFCG)
Campina Grande – PB – Brazil

dimascnf@copin.ufcg.edu.br, {cesp, massoni}@dsc.ufcg.edu.br

Abstract. *Stored procedures are commonly used to provide access and manipulation of database data for information systems and other applications. If procedures present inefficient programming logic or data manipulation, excessive delays are provided to the client applications. Such delays can cause, among other problems, expressive financial losses to enterprises. In additon, if procedures are developed using bad programming practices, they may become complex to maintain and evolve. In general, attempts to minimize these problems using manual analysis of source code are labor- and time-consuming. In this work, we present PL/SQL Advisor, a static analysis-based tool, which automatically detects potential improvements on database stored procedures written in PL/SQL. The results of a case study, using real open source projects, show that our tool is able to suggest a reasonable amount of code improvements with low cost.*

1. Introduction

Many DataBase Management Systems (DBMSs) have introduced platform dependent database programming languages. These programming languages enable programmers to develop and store part of the business logic of information systems as stored procedures (for convenience, in this paper we use the term procedure for both procedure and function). As presented by [Berkovic et al. 2010, Feuerstein 2007], stored procedures can bring several benefits to the whole system functioning, such as performance improvements, security and simpler maintenance. Due to these advantages, the usage of stored procedures became an effective technique on the architecture of information systems and general database applications. However, these advantages only make sense if stored procedures are developed concerning acceptable performance and good programming practices.

There are circumstances in which stored procedures can consume CPU resources, without any database access, at an excessive rate [Harrison 2000]. Inefficient processing performed by stored procedures can also generate unexpected delays for client applications. Thereby, stored procedures must follow good programming practices and use the available resources efficiently. Moreover, since part of the application logic of a system can be implemented as stored procedures, efforts to develop procedures using good programming practices and standards will decrease further costs to evolve and maintain them. In practice, programmers can eventually use bad programming practices or implement inefficient code while developing stored procedures. In order to minimize these problems, stored procedures must undergo manual inspections. However, due to the high

costs [Sommerville 2010; Young and Pezze 2005], it usually becomes too expensive to perform manual analysis on large business logic implementations available as stored procedures.

In general, stored procedures contain both SQL commands and procedural code. Although SQL optimizations provide a greater performance impact, procedural code optimizations can also bring performance gains [Feuerstein 2007; Berkovic et al. 2010]. In this paper, we present PL/SQL Advivsor, a tool based on static analysis of source code, which automatically detects efficiency and quality potential improvements on stored procedures written in PL/SQL [Sheila Moore 2009], the Oracle programming language. The tool focus on the analysis of the procedural code within stored procedures. We investigate the efficacy and costs of analyzing stored procedures using the proposed tool. For doing so, we describe a case study, with real open source projects, that we performed to evaluate the tool capability of detecting improvements on stored procedures. The results indicate that the tool is able to perform automatic detection of a variety of improvements on stored procedures and it is able to analyze a reasonable amount of source code with a low cost. In practice, these improvements can help the development and maintenance of information systems or other database applications by: i) decreasing the time required to perform manual inspections on stored procedures; ii) saving maintenance and evolution costs; and iii) optimizing the time spent by the applications to access and manipulate data on databases using stored procedures.

2. Improvements of Stored Procedures

In this section, we discuss potential improvements (efficiency and code quality) on the source code of stored procedures and point out the problems with manually identifying these improvements.

2.1. Efficiency Improvements

Several approaches in the literature [Berkovic et al. 2010; Feuerstein 2007; Hall 2006; Harrison 2000] focus efforts on identifying good practices for the development of efficient stored procedures. Efficiency improvements are related to changes that can enhance the execution time or memory utilization of the procedures. Regarding **efficiency**, we investigate the following subclasses of improvements:

Data type changes (DTC). Some data types available on database programming languages are stored as a lower level representation and use machine arithmetic. The usage of these data types result in better performance to execute arithmetic operations and require less storage. Another possible improvement related to this subclass is to optimize the size of declared variables in order to minimize memory consumption;

Changes in the expression evaluation order (EO). Ordering conditional structures or logical expressions such that the most frequently chosen branches/expressions are placed in the first options of the list to be evaluated will avoid some logic expressions evaluations, and thus improve performance;

Reduction of parameter copying (PC). The process of copying large parameters, such as records, collections, and objects requires both time and memory utilization, which affects performance. Thereby, passing parameters by reference decreases the time and memory required to copy parameters;

Optimization of database-related operations (DB-OP). Regarding a database context, some specific resource utilization may be optimized in order to save memory and improve performance: reduce context switches, decrease the frequency of commit executions and use implicit cursors to iterate over data stored in databases;

Utilization of native functions (NF). Attempts to overwrite built-in functions available on the database programming languages will result in lower performance, since the overwritten implementation may perform inefficient computations or add additional overheads to native functions;

Removal of useless declarations (UD). Removing unused variables and parameters from the stored procedures will result in storage saving. Similarly, removing declarations inside loop statements will cause the same effect. Moreover, the removal of unused parameters decreases the overhead of the copying effort, and thus improves performance.

Several performance experiments which investigate the impact of these kind of efficiency improvements are available in the related literature [Berkovic et al. 2010; Hall 2006; Harrison 2000]. In order to illustrate some of efficiency problems, Code 1 is used as an example. It illustrates an excerpt from a PL/SQL procedure which aims to persist the employees of an enterprise that are eligible to retire. Some problems have been purposely added to Code 1, e.g. usage of inefficient data types on a variable used for arithmetic operations (line 2), unused parameter (line 1), unused variable (line 3), inefficient logic expressions order (line 6) and execution of a *commit* statement for each iteration of a loop (line 9). These kinds of problems can potentially reduce the performance of an information system or general database applications.

Code 1. Inefficient procedure which persists the employees of an enterprise that are eligible to retire.

```
1. CREATE PROCEDURE ELIGIBLE_EMPLOYEES(emp_limit IN INTEGER) AS
2.    count_employees INTEGER := 0;
3.    emp employee_table%ROWTYPE;
4. BEGIN
5.    FOR emp IN (SELECT ID, NAME, YEARS_WORKED FROM EMPLOYEE_TABLE) LOOP
6.      IF (employee_score(emp.id) > 5 AND emp.years_worked > 30) THEN
7.         INSERT INTO ELIGIBLE_EMPLOYEES VALUES (emp.id, emp.name, emp.years_worked);
8.         count := count + 1;
9.         COMMIT;
10.        . . .
11.  DBMS_OUTPUT.PUT_LINE('# Eligible Employees:' || count_employees);
12. END ELIGIBLE_EMPLOYEES;
```

2.2. Code Quality Improvements

Maintenance costs of business applications are as expensive as development costs [Sommerville 2010] and the authors of [Erlikh 2000] suggest that 90% of a software cost is related to evolution tasks. In this context, it is important that programmers follow good programming practices when developing database stored procedures. Otherwise, bad smells may drastically reduce code readability and lead to high maintenance and evolution costs. Quality improvements are related to changes that can enhance code readability or instill patterns throughout the procedures. Regarding **code quality**, we investigate the following subclasses of improvements:

Simplify code control flow (CCF). The removal of scape commands as well as of multiple exit points on the code simplifies its control flow, and thus facilitates its understanding;

Avoid dodgy code (DC). These improvements refer to avoid using constructs which drastically difficult code readability (and increase the probability of fault insertions) or which are unreliable and may cause runtime errors;

Avoid collateral effects (CE). Understanding a program with side effects requires knowledge about its context and its possible states; therefore, the code becomes hard to read, understand, and debug;

Improve the meaning of identifiers (MI). Spreading meaningful identifiers for non-obvious parts of the source code significantly increases program readability;

Usage of programming styles (PS). Using code patterns, e.g., naming conventions and proper indentation, usually facilitates code understanding.

2.3. Manual Inspections

The task of identifying parts of stored procedures source code in which the improvements presented in Sections 2.1 and 2.2 may be performed by manual inspections. However, it is difficult to introduce formal inspections into many software development organizations and they appear to slow down the development process [Sommerville 2010]. Moreover, manual inspections are time-consuming tasks and may require meetings, which can become a scheduling bottleneck [Young and Pezze 2005]. For these reasons, we believe that an automatic approach is more suitable to analyze the source code of stored procedures.

3. PL/SQL Advisor - A Static Analysis-Based Tool

In this section, we present a tool, based on static analysis of source code, to perform automatic detection of potential improvements on stored procedures. Automated static analysis is more limited in applicability compared to manual inspection, but is selected when available because substituting machine cycles for human effort makes them particularly cost-effective [Young and Pezze 2005].

Automated static analysis uses a suitable representation of a program's source code to achieve its goal. This representation can be quite different depending on the type of analysis that is performed and on the underlying formalism. We use a Control Dependence Tree (CDT) [Young and Pezze 2005, Lengauer and Tarjan 1979], which derives from the definition of Control Flow Graph [Young and Pezze 2005] [Allen 1970], to represent a stored procedure source code. A CDT provides a hierarchical representation of the source code using the definition of control dependency between the statements of the source code. The related literature [Lengauer and Tarjan 1979, Young and Pezze 2005] presents formal steps required to turn a procedural code into an equivalent representation as a Control Dependence Tree. For example, Figure 1 shows the CDT representation of the Code 1. The statements of the Code 1 are represented as their respective lines in Figure 1.

We implemented a tool, called PL/SQL Advisor[1], to perform automatic detection of potential improvements on stored procedures written in PL/SQL. The proposed tool is a implementation of a generalizable approach [Nascimento 2013] to analyze stored procedures and is fully implemented in JAVA. Figure 2 illustrates an overview of the process performed by the proposed tool. In the first step, a PL/SQL stored procedure source code

[1]http://sites.google.com/plsqladvisor

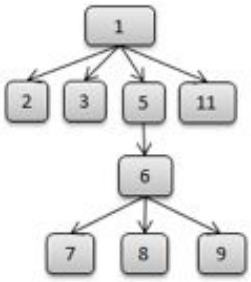

Figure 1. CDT representation of Code 1.

is parsed by a suitable parser for the PL/SQL programming language. This process is supported by a PL/SQL Lexer. The generation of the PL/SQL Lexer and Parser was carried out by ANTLR [Parr and Quong 1995]. In the second step, a CDT is created for the PL/SQL stored procedure after the parse operation. Since we do not intend to create a compiler, we require that the source code must be syntactically correct. Otherwise, the CDT analysis will lead to a mistaken static analysis.

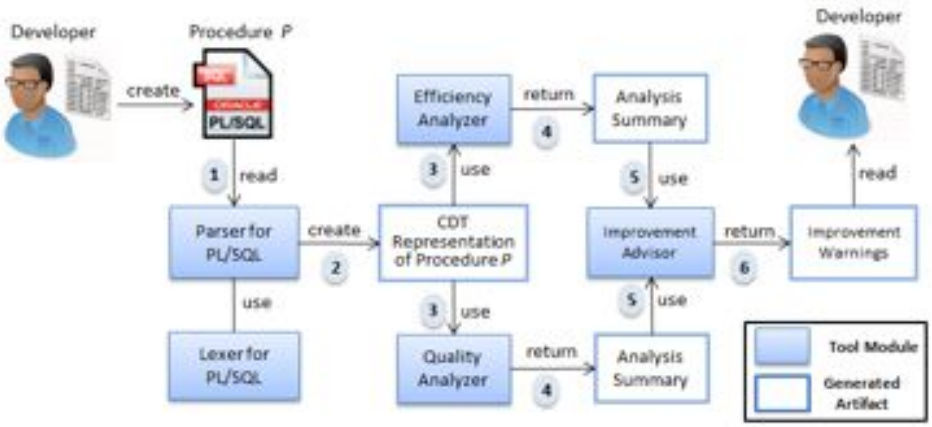

Figure 2. Process executed by PL/SQL Advisor.

In the third step, once the CDT is properly structured, the CDT Analyzers (Efficiency Analyzer and Quality Analyzer) search for predefined patterns on the statements represented as a CDT. The Efficiency Analyzer is responsible for analyzing the CDT structure in order to detect patterns of the source code in which procedural efficiency improvements can be applied. Analogously, the Quality Analyzer is responsible for detecting code smells throughout the CDT structure. To implement the third step, classic search algorithms like DFS (Depth First Search) and BFS (Breadth First Search) can be executed for finding useful information.

In the fourth step, the CDT Analyzers return an analysis result summary, which contains all the identified patterns and their respective positions in the CDT. In the fifth step, the Improvement Advisor evaluates the analysis summary and decides which improvements will be reported on the output and in which order they will appear. This step is important since sometimes more than one problem may be related to the same statement

and there might be a single warning which suggests a better solution for the problems. For example, if a variable is declared using an inefficient data type and the variable is not used in the procedure, suggesting the variable removal is the most suitable warning in this case. Besides, ordering the warnings, according to a certain relevance criteria, is important to improve the output readability, and thus save time in the tuning process. Some possible ordering criteria are: warning frequency, warning subclass or warning detection difficulty (i.e., how hard it is to identify the improvement using manual inspection). In the sixth step, the Improvement Advisor selects the suitable warnings related to possible improvements on the stored procedure source code and redirects them to the output. Note that the process executed by the tool can be easily extensible to add new analysis. For doing so, only the components Efficiency Analyzer, Quality Analyzer and Improvement Advisor need to be evolved.

In Table 1 we show all the improvements that are detected by PL/SQL Advisor. The improvements are classified according to the subclasses defined in Section 2. In order to illustrate how the CDT structure can be analyzed by the proposed tool, Figure 3 shows an example of a pseudo-code to detect the improvement E9, shown in Table I. The pseudo-code searches for *commit* statements that are dominated by a loop statement. Additionally, it is checked if the amount of iterations executed by the loop is above a threshold received as parameter (the parameter can be configured by the developer). This checking can be statically analyzed in the cases that the range of the loop is fixed. If these conditions are verified, the Efficiency Analyzer reports this fact in the analysis summary, which will lead the tool to report a warning related to the efficiency improvement E9.

```
search_commits_within_loops(Statement stmt, threshold int)
    IF (TYPE(stmt) = LOOP AND RANGE(stmt) > threshold)
        FOR child in children(stmt)
            search_commit(child)
    FOR child IN children(stmt)
        search_commits_whithin_loops(child, threshold)
END
search_commit(Statement stmt)
    IF TYPE(stmt) = COMMIT
        analysis_summary.add('commit whithin loop statement',
                                line(stmt))
    FOR child IN children(stmt)
        search_commit(child)
END
```

Figure 3. Example of a search in the CDT representation for a stored procedure.

In order to illustrate an output of PL/SQL Advisor, Code 1 is used as example. A SQL file containing the source of Code 1 was selected on the PL/SQL Advisor GUI (Figure 4) and the process shown in Figure 2 was executed by the tool. In Figure 5 it is shown the warnings reported by PL/SQL Advisor after the analysis of Code 1. To simplify, only the efficiency warnings are shown. Note that the tool is able to create suitable warnings related to all efficiency problems of Code 1 discussed in Section 2.1. According to the related literature [Berkovic et al. 2010; Feuerstein 2007; Hall 2006; Harrison 2000], the appliance of these warnings can significantly improve the performance of a stored procedure.

Table 1. Improvements reported by PL/SQL Advisor.

Efficiency Improvements	
Short Description	*Subclass*
E1: Use native INTEGER types	DTC
E2: Use native NUMBER types	DTC
E3: Avoid using constrained datatypes (e.g. POSITIVE, POSITIVEN, NATURAL)	DTC
E4: Optimize the order of logic expressions	EO
E5: Optimize the order of conditional commands	EO
E6: Pass parameters by reference	PC
E7: Optimize the length of variables	DTC
E8: Reduce context switches [Berkovic et al. 2010; Feuerstein 2007]	DB-OP
E9: Decrease commit frequency	DB-OP
E10: Remove declarations inside loops	UD
E11: Use built-in string functions	NF
E12: Iterate over rows implicitly	DB-OP
E13: Remove unused variables	UD
E14: Remove unused parameters	UD
Code Quality Improvements	
Short Description	*Subclass*
Q1: Avoid using GOTO commands	CCF
Q2: Avoid declaring a variable using the same identifier of a loop for counter	DC
Q3: Avoid using escape commands (e.g. EXIT, CONTINUE) inside loops	CCF
Q4: Avoid using return commands inside loops	CCF
Q5: Avoid using multiple RETURN clauses in a function	CCF
Q6: Avoid using result parameters on functions signatures	CE
Q7: Turn a procedure which has only one result parameter into a function	PS
Q8: Encapsulate complex logic expressions using functions	MI
Q9: Use naming conventions to indicate the parameter passing types	PS
Q10: Declare one variable by line	PS
Q11: Place one parameter by line	PS
Q12: Declare named constants for literal values	MI
Q13: Use %type or %rowtype for data types used on *fetch* or *select into* statements	DC
Q14: Verify if an explicit cursor is not already opened before opening it	DC
Q15: Use an ELSE clause on CASE commands	DC
Q16: Use named (instead of positional) notation to pass parameters on procedure calls	DC
Q17: Avoid returning NULL values on boolean functions	DC

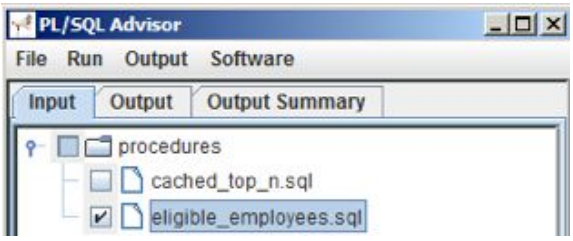

Figure 4. Selection of the eligible_employees.sql file on PL/SQL Advisor GUI.

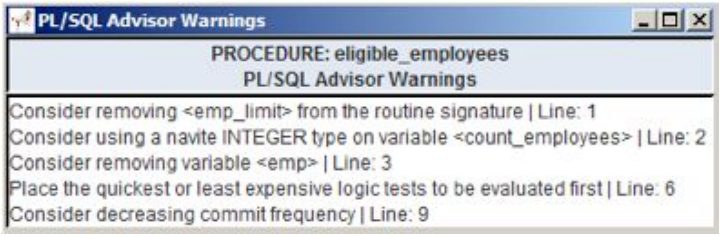

Figure 5. The PL/SQL Advisor output.

4. Evaluation

Our evaluation methodology addresses three questions: **(i)** *Is the proposed tool able to detect possibilities of improvements on the source code of stored procedures*?; **(ii)** *Are the efficiency and quality mistakes analyzed by the tool present on real stored procedures*?; and **(iii)** *Does the proposed tool perform an efficient detection of code improvements*?

To address the first question, we measured the total amount of warnings reported by PL/SQL Advisor in our case study. To answer the second question, we discuss the frequency of warnings reported by the tool. Finally, to answer the third question, we measured the elapsed time to execute PL/SQL Advisor. The obtained results are discussed in Section 4.2.

4.1. Case Study

We performed a case study using real projects written in PL/SQL. We selected four projects from an online repository[2]. Among the available options, we selected the projects which contained a reasonable amount of lines of code and a good variety of programming constructs. The selected projects are the following:

- **JSON Library**: a set of tools for generating JSON from PL/SQL;
- **CodeBrew**: a framework for PL/SQL Gateway developers;
- **DBLens**: an Oracle-based toolkit for performing collaborative filtering;
- **STR**: a set of tools for string manipulations.

For each project, we selected 10 stored procedures to be analyzed by PL/SQL Advisor. We prioritized the procedures that presented more lines of procedural code. Each procedure was passed as argument to PL/SQL Advisor and we measured the total amount of reported warnings and the elapsed time of each execution. Table 2 presents information about the stored procedures selected in the case study, e.g. number of Lines of Code (LoC). We also show the number of Efficiency (EW) and Code Quality (CQW) Warnings reported by PL/SQL Advisor after analyzing the selected procedures of each project.

The individual warnings count on PL/SQL Advisor output, related to the improvements presented in Section 2, is shown in Table 3. The improvements which are not contained in Table 3 were not reported by PL/SQL Advisor on the case study.

[2]The projects are available at http://plnet.org

Table 2. Warnings occurrences on PL/SQL Advisor output.

Project	SP Analyzed	Analysis Time	LoC Analyzed	EW	CQW
JSON Library	10	2,33 s	268	16	45
CodeBrew	10	1,35 s	473	22	34
DBLens	10	1,87 s	829	80	40
STR	10	1,26 s	314	27	35

Table 3. Individual warnings occurrences on PL/SQL Advisor output.

Efficiency Improvements													
Project	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E12	E13	E14
JSON	0	2	0	2	9	0	3	0	0	0	0	0	0
CodeBrew	1	8	0	2	3	0	5	3	0	0	0	0	0
DBLens	22	13	0	2	17	0	9	0	4	0	4	4	5
STR	12	0	0	0	10	0	1	1	0	0	0	0	3
Code Quality Improvements													
Project	Q1	Q2	Q3	Q4	Q5	Q7	Q8	Q9	Q11	Q12	Q13	Q15	CQ16
JSON	0	0	6	0	8	1	0	10	6	12	0	0	2
CodeBrew	0	0	3	1	6	1	0	7	0	16	0	0	0
DBLens	3	0	5	0	1	2	1	8	0	18	2	0	0
STR	0	4	4	0	8	0	0	4	0	14	0	0	1

4.2. Discussion

This section discusses the methodology used to validate our work as well as the obtained results. Basically, we address the raised questions in the evaluation planning.

(i) *Is the proposed tool able to detect possibilities of improvements on the source code of stored procedures*?

PL/SQL Advisor reported 299 efficiency warnings after analyzing 1884 lines of code of all 40 stored procedures. The tool reported general programming improvements as well as specific improvements on a database context. The amount of reported warnings is a good evidence that the investigated coding mistakes are present on the development of real stored procedures. Moreover, it shows that simple static analysis techniques seem to effectively identify common programming mistakes within a database context.

(ii) *Are the efficiency and quality mistakes analyzed by the tool present on real stored procedures*?

We now discuss the data shown in Table 3. PL/SQL Advisor reported 145 efficiency warnings. The warnings related to data type changes (E1, E2, E3, and E7) summed together 58 hits. These improvements may cause meaningful impact on procedures that performs intensive arithmetic operations [Feuerstein 2007; Hall 2006; Harrison 2000]. We attribute the excessive quantity of these warnings to two reasons. First, some warnings can be false positives since there are procedures that do not perform enough arithmetic operations to justify the data type changes. Second, we believe that the developers of the

DBLens project are not aware of the improvements E1 and E2.

The improvements related to changes on the expression evaluation order (E4 and E5) summed 45 hits. Although these warnings are in fact opportunities for improvements, we believe that part of them is useful to tune the analyzed stored procedures. The warnings related to optimization of database-related operations (E8, E9, and E12) appeared together 12 times. These improvements may cause significant impact [Berkovic et al. 2010; Hall 2006; Harrison 2000] on the memory consumption and execution time of specific stored procedures in the investigated projects. PL/SQL Advisor reported 12 warnings related to the removal of useless declarations (E10, E13, and E14). These warnings are surely useful to minimize memory usage during the execution of the analyzed stored procedures. We believe that they were frequently reported because as the number of lines of code increases, it becomes harder to notice unused variable declarations and unused parameters on the stored procedures signatures.

PL/SQL Advisor reported 154 warnings related to code quality. The warnings related to control flow complexity (Q1, Q3, Q4, and Q5) appeared 45 times. This amount of warnings is a significant part (nearly 30%) of the reported code quality improvements. These warnings can substantially simplify the control flow of some of the investigated stored procedures and greatly improve their readability. PL/SQL Advisor reported 9 warnings related to the improvements Q2, Q13, Q14, Q15, Q16, and Q17. Although this quantity of warnings is not a significant part of the reported code quality improvements, they are useful to prevent possible runtime errors and remove obscure pieces of code.

The warnings related to programming style (Q7, Q9, Q10, and Q11) improvements summed together 49 hits. They are quite representative (nearly 32% of the reported quality improvements) and useful to establish patterns and conventions throughout the code. The improvements related to code semantics (Q8 and Q12) appeared 61 times. The improvement Q12 was the most recurrent improvement on the case study (60 occurrences). Apparently, the developers of the investigated projects are not concerned on creating named constants for the literal values (many of them are quite meaningless) used on the stored procedures. In general, the occurrence of improvements is not uniform. Some of the improvements (E1, E2, E5, E7, Q3, Q5, Q9, and Q12) occurred very frequently, whereas other improvements (E3, E6, E10, E11, Q6, Q8, Q10, Q15, and Q17) had very few occurrences or any occurrences at all. Some of the coding mistakes presented in this paper seem more likely to be remembered and properly avoided/corrected on the development of stored procedures.

(iii) *Does the proposed tool perform an efficient detection of code improvements*?

PL/SQL Advisor took less than 7 seconds to analyze and create suitable warnings for 1884 lines of code. Clearly, the automatic analysis time is much faster than an equivalent time to perform a manual inspection in all the 40 stored procedures. Besides, manual inspection is an error prone process and may occasionally fail to detect potential code improvements. On the other hand, once the tool implementation is correct, it is guaranteed that all implemented improvement detections will be properly reported on the analysis output.

According to the obtained results, the proposed tool neither seem to be sensitive to

the complexity of stored procedures nor to the amount of analyzed lines of code. Thereby, the tool can be used to analyze a large amount of stored procedures with low cost. The tool can be executed repeatedly during the development phase (after subsequent source code changes) and its execution will not represent a significant overhead to the development process. This characteristic can potentially reduce costs of developing database applications that use stored procedures to manipulate data in databases.

5. Related Work

SQL and PL/SQL tuning. The *Quest SQL Optimizer for Oracle* tool identifies potential performance issues and automates SQL optimization by scanning and analyzing running SQL statements, PL/SQL, or other source code. The tool identifies problematic SQL, rewrites statements, locates fastest alternatives, and generates optimized code automatically. Although the *Quest SQL Optimizer for Oracle* is able to analyze PL/SQL code, it focuses on optimizing SQL statements within the procedural code. The *SQL Tuning Advisor* tool allows the Oracle query optimizer to run in tuning mode where it can gather additional information and make recommendations about how specific statements can be tuned. The *SQL Tuning Advisor* also focuses on tuning SQL statements. Our work is different from these tools since the proposed tool focuses on improving the procedural code of stored procedures.

Automatic Analysis of Stored Procedures. The *SQL Enlight* [Ubisoft 2007] tool aims to check the conformity of stored procedures written in Transact-SQL with aspects such as naming conventions, design rules, performance and readability of the source code. The *Sonar Source* [SonarSource 2008] tool is able to check the conformity of PL/SQL procedures with pre-defined guidelines concerning performance, good programming practices, naming conventions and comments. PL/SQL Advisor is different from the copyrighted tools offered by [SonarSource 2008, Ubisoft 2007], since no information is published about the internal details of these tools. On the other hand, we provide details about the design and implementation of the proposed tool. Moreover, we also discuss the results of an experimental evaluation of PL/SQL Advisor in the context of open source stored procedures.

6. Conclusions

In this paper, we present PL/SQL Advisor, a tool designed to reduce the costs of the development and maintenance of database applications that manipulate data using stored procedures. We presented a list of efficiency and code quality improvements on database stored procedures discussed in the related literature. We then pointed out the practical drawbacks related to the usage of manual inspections to identify these improvements.

PL/SQL Advisor performs automatic analysis on database stored procedures. To this end, the tool employs a simplified source code representation (CDT) and is able to identify well-known source code improvements as well as specific improvements on a database context.

We performed a case study, using real open source projects written in PL/SQL, in order to evaluate the proposed tool. The tool was able to report warnings related to both general programming languages improvements as well as to specific improvements within a database context. We discussed the most frequently reported warnings, their

implications on the stored procedures and possible explanations for their frequency of occurrence. The subclasses of improvements more frequently reported on the case study were: data type changes, changes on the expressions evaluation order, optimization of database-related operations, simplification of code control flow, avoid dodgy code, improvement of the meaning of identifiers and usage of programming styles. The appliance of these kind of improvements on stored procedures used to manipulate real world data can lead to several benefits, such as: decreasing of application delays, improvement of source code readability and reduction of costs to understand and analyze stored procedures. For future work, we intend to evolve PL/SQL Advisor in order to detect other efficiency improvements which are not covered in this paper. For example, minimizing data type conversions, optimizing loops and avoiding recursion.

References

Allen, F. E. (1970). Control flow analysis. *SIGPLAN Not.*, 5(7):1–19.

Berkovic, I., Ivankovic, Z., Markoski, B., Radosav, D., and M., I. (2010). Optimization of bulk operation performances within oracle database. In *8th International Symposium on Intelligent Systems and Informatics (SISY), 2010*, pages 163 –167.

Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT Professional*, 2:17–23.

Feuerstein, S. (2007). *Oracle pl/sql best practices, 2nd edition*. O'Reilly, second edition.

Harrison, G. (2000). *Oracle SQL High-Performance Tuning, Second Edition*. Prentice Hall Professional Technical Reference, 2nd edition.

Lengauer, T. and Tarjan, R. E. (1979). A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141.

Nascimento, D. C. (2013). Uma Abordagem para Análise Estática Automática de Procedimentos Armazenados em Bancos de Dados. Master's thesis, Universidade Federal de Campina Grande, Campina Grande, Brasil.

Parr, T. J. and Quong, R. W. (1995). Antlr: A predicated-ll(k) parser generator. *Software: Practice and Experience*, 25(7):789–810.

Sheila Moore, E. B. (2009). *Oracle Database PL/SQL Language Reference, 11g Release 2 (11.2)*. Oracle.

Sommerville, I. (2010). *Software Engineering*. Addison-Wesley, Harlow, England, 9. edition.

SonarSource (2008). Sonar for pl/sql: http://www.sonarsource.com/products/plugins/.

Ubisoft (2007). Sql enlight for t-sql: http://www.ubitsoft.com/index.php.

Young, M. and Pezze, M. (2005). *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons.