Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Uma Abordagem para Análise Estática Automática de Procedimentos Armazenados em Bancos de Dados

Dimas Cassimiro do Nascimento Filho

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linhas de Pesquisa: Banco de Dados, Engenharia de Software

Prof. Dr. Carlos Eduardo Santos Pires e Prof. Dr. Tiago Lima Massoni
(Orientadores)

Campina Grande, Paraíba, Brasil

Resumo

Os procedimentos armazenados em bancos de dados são comumente utilizados por aplicações para acessar e manipular dados em bancos de dados. Se os procedimentos forem implementados de maneira ineficiente, esperas demasiadas podem ser repassadas para a camada de aplicação e acarretar perdas financeiras significativas para as empresas. Caso sejam implementados usando más práticas de programação, os procedimentos podem apresentar difícil legibilidade e entendimento. Considerando que uma parte considerável da lógica de negócio das aplicações pode ser desenvolvida na forma de procedimentos, a redução destes problemas utilizando inspeções manuais pode tornar-se um processo custoso, propenso a erros e desestimulante para os desenvolvedores. Neste trabalho, é proposta uma abordagem, baseada em análise estática automática de código-fonte, para verificação da conformidade de procedimentos de banco de dados com diretrizes pré-definidas de eficiência e qualidade. A abordagem proposta foi instanciada para uma linguagem de programação de banco de dados específica e avaliada de três maneiras. Primeiramente, foi medido o impacto da aplicação de diretrizes de eficiência no tempo de execução de procedimentos de banco de dados. Para as maiores cargas de trabalho testadas, as diretrizes investigadas acarretaram em melhorias superiores a 80%. Além disso, foi realizado um estudo de caso, utilizando procedimentos de banco de dados reais, no intuito de avaliar a eficiência (tempo de análise automática) e eficácia (quantidade de advertências reportadas) da abordagem proposta. Neste processo, foram reportadas 299 advertências após a análise dos procedimentos ($\sim$2KLoC) selecionados e, para tal, foram necessários menos de 7 segundos para a realização da análise automática. Por fim, foram realizados experimentos, no contexto de um projeto industrial real, no intuito de comparar as características da análise realizada pela abordagem proposta com as realizadas de forma manual por desenvolvedores. Nestes experimentos, a abordagem automática apresentou eficiência e eficácia superiores em todos os cenários observados.

palavras-chave: procedimentos de banco de dados, análise estática, análise de desempenho, eficiência, qualidade de código.

Abstract

Stored Procedures are commonly used by applications to access and manipulate data in databases. If procedures are inefficiently implemented, excessive delays are passed to the application layer and can bring significant financial losses to enterprises. If procedures are implemented using bad programming practices, it may become hard for a developer to read and understand them. Considering that a significant part of the business logic of the applications can be implemented as stored procedures, the reduction of these problems using manual inspections may become a costly, error-prone and discouraging process to the developers. In this work, we propose an approach, based on static analysis of source code, to check the conformity of stored procedures with predefined efficiency and quality guidelines. The approach was instantiated to a specific database programming language and evaluated in three different ways. First, we measured the impact of the appliance of efficiency guidelines on the execution time of stored procedures. Considering the highest processing loads under test, the investigated guidelines led to optimization gains greater than 80%. Moreover, a case study, using real open source projects, was conducted to evaluate the efficacy (amount of reported warnings) and efficiency (automatic analysis time) of the proposed approach. In this process, 299 warnings were reported after the analysis of the selected procedures ($\sim$2KLoC) and, for doing so, less than 7 seconds of automatic analysis were required. Finally, we performed experiments, in the context of an industrial project, in order to compare the automatic analysis with manual inspections performed by developers. In these experiments, the automatic approach presented a superior efficacy and efficiency in all observed treatments.

keywords: stored procedures, static analysis, performance analysis, efficiency, code quality.

Agradecimentos

A Deus, por todas as alegrias e dificuldades pelas quais me fez passar. Sem estas dificuldades não teria me tornado mais forte e nunca teria chegado até aqui.

À minha Mãe, Teresinha, por todo o apoio durante minha vida e por ter me ensinado a manter sempre a cabeça erguida diante das dificuldades. Ao meu Pai, Dimas, por todo o aconselhamento, palavras sábias e apoio incondicional que me proporcionou. Aos meus irmãos, Lucila e Diógenes, pelos ótimos momentos que vivemos.

Ao meu professor orientador Carlos Eduardo, por toda a ajuda, pelas aulas com as quais tanto aprendi e pelas excelentes (e essenciais) revisões neste trabalho. Também agradeço por ter sempre mantido a serenidade e um sorriso amigo não importando a situação.

Ao meu professor orientador Tiago Massoni, por toda a ajuda e pelas dicas geniais para a execução deste trabalho. Em especial, por ter acreditado no meu potencial desde o início, apesar da cara de sono que eu estava no primeiro dia em que conversamos sobre a ideia deste trabalho.

À minha namorada, Stênia, pelo carinho, atenção, paciência e deliciosos brigadeiros.

A alguns professores que contribuíram de maneira marcante para a minha formação acadêmica e cujos ensinamentos foram absolutamente essenciais para a concretização deste trabalho: Bernardo Lula, Carlos Eduardo, Cláudio Baptista, Franklin Ramalho, Ianna Sodré, Hyggo Almeida, Jacques Sauvé, Marcus Salerno, Raquel Lopes e Tiago Massoni. Agradeço por terem compartilhado seus conhecimentos de maneira tão brilhante.

Ao pessoal do projeto MAS-SCM: Pryscilla Dora, Fernando, Ramon e Laura. Pelos ótimos momentos de convivência e pelas contribuições nos experimentos deste trabalho.

Ao pessoal do projeto Icuriã, pelo ótimo convívio profissional. Aos meus grandes (alguns atualmente "ex") amigos do tempo de graduação: Vicente, Vinicius, Zé Filho e Fabiano, por tornarem minha estadia em Campina Grande mais alegre. Ao pessoal do Apt 04: Fernando, Arnaldo e Daniel; pelos ótimos anos de convivência. A Renato Miceli, por algumas contribuições técnicas para a ideia inicial deste trabalho.

Conteúdo

Lista de Acrônimos e Abreviações

ACD - *Árvore de Controle de Dependência*

C_D - *Código da Diretriz*

GFC - *Grafo de Fluxo de Controle*

GFD - *Grafo de Fluxo de Dados*

P_D - *Pontuação da Diretriz*

R_A - *Resumo da Análise*

SGBD - *Sistema Gerenciador de Banco de Dados*

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Nos últimos anos, as aplicações de banco de dados tem enfrentado muitos desafios. Fatores como o frequente aumento da quantidade de dados processados, predominância da arquitetura cliente-servidor ou distribuída dos sistemas, restrições de segurança bastante rígidas e restrições de tempo cada vez menores demandaram novas estratégias para permitir o acesso e a manipulação de dados em banco de dados. Um dos grandes problemas no desenvolvimento de sistemas de banco de dados está relacionado à complexidade dos esquemas e dos dados armazenados. Uma vez que programadores de aplicações muitas vezes não possuem sólidos conhecimentos em banco de dados [12], as tarefas de desenvolvimento e manutenção de sistemas que utilizam complexos esquemas de banco de dados podem tornar-se demasiadamente complicadas e caras. Sendo assim, foi necessário criar uma maneira de permitir que a camada de aplicação pudesse acessar e manipular dados armazenados em banco de dados sem que precisasse lidar diretamente com a complexidade inerente aos dados.

Com este fim, muitos Sistemas Gerenciadores de Banco de Dados (SGBD) introduziram linguagens de programação dependentes de plataforma. Estas linguagens de programação permitem que programadores armazenem parte da lógica de negócio das aplicações diretamente no banco de dados na forma de procedimentos (neste trabalho, o termo procedimento é utilizado para denotar procedimentos ou funções). Tais procedimentos podem ser chamados por outros procedimentos, por outros esquemas ou diretamente pela camada de aplicação. Desse modo, ao invés de enviar instruções diretamente para o SGBD, a camada de aplicação pode simplesmente realizar chamadas a tais procedimentos que, por sua vez, realizam o processamento desejado na própria camada de dados.

Além de permitir que a camada de aplicação manipule dados em bancos de dados sem ter que lidar diretamente com a complexidade dos dados, existem diversos outros benefícios relacionados à utilização de procedimentos de banco de dados [4] [9] [12]. Dentre estes benefícios, destacam-se: **melhorias de desempenho** (devido a diminuição do tráfego de comandos SQL por canais de comunicação e do tempo de compilação dos mesmos), **segurança** (sem que a aplicação execute comandos SQL diretamente no banco de dados, diminui-se as chances da ocorrência de ataques do tipo *SQL Injection*) e **simplificação da manutenção do código-fonte do procedimento** (diversas aplicações podem realizar chamadas ao mesmo código-fonte dos procedimentos).

Levando em consideração as vantagens apresentadas, a utilização de procedimentos de banco de dados tornou-se uma técnica bastante efetiva em sistemas de banco de dados [4] [12]. Porém, a utilização de tais procedimentos pode também acarretar em problemas desafiadores de serem solucionados.

1.1 Desafios no Uso de Procedimentos de Banco de Dados

Muitos trabalhos [13] [12] [9] [11] [4] realizam esforços em identificar boas práticas a serem seguidas por programadores de procedimentos de banco de dados. A literatura também discute uma grande quantidade de más práticas de programação que podem reduzir drasticamente a eficiência e dificultar a manutenção de procedimentos de banco de dados. Porém, mesmo com uma relativamente vasta literatura na área, nem todos os programadores possuem sólidos conhecimentos no desenvolvimento de procedimentos de banco de dados eficientes e seguem boas práticas de programação. Sendo assim, programadores inexperientes podem potencialmente escrever procedimentos de baixa qualidade que acarretem problemas como: baixo desempenho, incorretude, pouca legibilidade e difícil manutenção e evolução dos procedimentos. Além disso, tais problemas não estão apenas relacionados a programadores inexperientes. Mesmo um programador experiente pode cometer erros e ocasionalmente implementar código-fonte ineficiente, inserir falhas ou utilizar más práticas de programação.

1.1.1 Implicações de Implementações Ineficientes

Usualmente, programadores consideram procedimentos de banco de dados como uma forma de acessar o banco de dados e concentram-se em tentar otimizar comandos SQL contidos nos procedimentos [13]. Contudo, a parte procedimental dos procedimentos de banco de dados também está sujeita a otimizações semelhantes a outras linguagens de programação. Há circunstâncias em que procedimentos de banco de dados podem consumir recursos de CPU, mesmo sem realizar acessos a banco de dados, a uma taxa excessiva [13].

A utilização de procedimentos de banco de dados faz sentido apenas se a execução dos mesmos apresentar um desempenho aceitável para o escopo das aplicações que os utilizam. Em muitas situações, programadores podem implementar um procedimento de banco de dados para realizar determinado processamento de várias maneiras distintas. Porém, algumas implementações podem apresentar lógica de programação e manipular dados de forma mais eficiente que outras. Claramente, implementações eficientes irão apresentar melhor desempenho. Ainda que implementações ineficientes possam ser detectadas durante a fase de desenvolvimento, não é o que usualmente ocorre na prática. Como apresentado em [13], problemas de desempenho são geralmente detectados apenas quando a aplicação passar a ser utilizada com dados e cargas de processamento reais.

Uma consequência crítica relacionada ao uso de procedimentos de banco de dados ineficientes são as esperas repassadas para a camada de aplicação. Considerando o contexto de um sistema de comércio eletrônico, tal sistema poderia apresentar atrasos intoleráveis aos usuários finais. Uma vez que seres humanos são sensíveis a tempos de resposta grandes, tais esperas podem levar potenciais clientes a abandonar o sistema e, consequentemente, gerar perdas financeiras. Outro exemplo no qual a implicação de esperas pode também ser crítica é no contexto de um sistema de tempo real. Se esperas demasiadas relacionadas à execução de procedimentos de banco de dados ineficientes (ou, obviamente, relacionadas a qualquer outro motivo) ocorrerem, um sistema de tempo real pode falhar em garantir suas restrições de tempo. Tais falhas podem causar perdas financeiras significativas ou até mesmo pôr em risco vidas humanas.

1.1.2 Implicações do Uso de Más Práticas de Programação na Manutenção de Software

Mesmo após os sistemas serem devidamente implantados no ambiente de produção, é comum que os mesmos ainda precisem sofrer mudanças no intuito de permanecerem úteis [28]. Uma vez que um software é colocado em uso, novos requisitos podem surgir e requisitos existentes podem ser modificados. Além disso, partes do software podem precisar ser modificadas para corrigir falhas, adaptar o software para uma nova plataforma, realizar refatoramentos, ou melhorar o desempenho ou outras características não funcionais do mesmo [28]. Uma vez que é possível armazenar parte da lógica de negócio de sistemas na forma de procedimentos de banco de dados, é comum que tais procedimentos também necessitem ser frequentemente modificados.

Manutenção de software é o processo de modificar um sistema após o mesmo ter sido colocado em funcionamento. Este processo é importante principalmente porque muitas organizações são completamente dependentes de seus sistemas [28]. Os autores de [18] sugerem uma série de leis que regem o processo de mudanças em software. Uma destas leis afirma que a manutenção de software é um processo inevitável.

Manutenção de software é geralmente um processo bastante custoso. Os custos de manutenção para aplicações de negócio são comparados aos custos de desenvolvimento [10]. Em [8], os autores sugerem que 90% dos custos de um software estão relacionados à tarefas de manutenção. Existem muitos fatores que contribuem para o alto custo da manutenção de sistemas. Um dos principais fatores é o fato de que adicionar funcionalidades após o sistema entrar em produção é mais caro que implementá-las durante a fase de desenvolvimento [28]. Muitas vezes, a manutenção de sistemas é realizada por empresas ou desenvolvedores que não participaram da fase de desenvolvimento. Desse modo, modificações no software após a implantação podem requerer um tempo considerável para o entendimento do mesmo [28]. Além disso, se um software é difícil de ser entendido é mais provável que desenvolvedores introduzam falhas no mesmo [18].

Esforços feitos durante a fase de desenvolvimento para tornar o software mais simples de ser entendido e modificado resultam em prováveis reduções nos custos de manutenção [28]. Neste contexto, é importante que procedimentos de banco de dados sejam desenvolvidos

seguindo boas práticas de programação e prezando pela legibilidade e facilidade no entendimento dos mesmos. Caso contrário, a manutenção de tais procedimentos pode tornar-se um processo complexo e, consequentemente, custoso.

1.2 Soluções Baseadas em Inspeções Manuais

A partir da década de 70, revisões formais e inspeções foram reconhecidas como fatores importantes para aumentar a produtividade e a qualidade dos produtos desenvolvidos. Desde então, foram largamente adotadas do desenvolvimento de projetos [17]. Assim como procurar defeitos em programas, inspeções podem também ser utilizadas para ampliar atributos de qualidade nos programas, tais como conformidade com padrões, portabilidade e legibilidade. As inspeções podem procurar por ineficiências, algoritmos inapropriados e estilos de programação não adequados que possam dificultar a manutenção e evolução de sistemas [28].

Tentativas de resolução dos problemas discutidos nas Seções 1.1.1 e 1.1.2 utilizando inspeções manuais irão acarretar, além de outras implicações, em um maior custo no desenvolvimento e manutenção de sistemas. Isto ocorre porque o processo de inspeção manual usualmente envolve as tarefas de: i) identificação de código-fonte que permita melhorias; ii) proposição de possíveis melhorias; iii) implementação das melhorias propostas; iv) teste e análise do impacto das melhorias implementadas; e v) checagem de possíveis mudanças semânticas indesejadas realizadas no passo iii). Realizar todas estas tarefas de forma manual pode tornar-se um processo demorado e custoso.

Ainda que inspeções mostrem-se como uma técnica efetiva, é difícil introduzi-las no processo de desenvolvimento de muitas organizações [28]. De acordo com [30], re-inspeções manuais podem ser tão caras quanto as inspeções iniciais. Inspeções manuais necessitam de tempo para serem planejadas e parecem atrasar o processo de desenvolvimento [28]. Além disso, inspeções podem requerer encontros presenciais, o que pode gerar gargalos de escalonamento de pessoas [30].

Realizar inspeções manuais em procedimentos de banco de dados quando grande parte da lógica de negócio de um sistema é implementada usando tais procedimentos pode tornar-se uma tarefa cansativa e desestimulante para os programadores. É também importante destacar

que atividades manuais são normalmente propensas a erros. Nesse contexto, é importante que sejam desenvolvidas técnicas para reduzir eventuais custos adicionais que implementações inadequadas de procedimentos armazenados em banco de dados possam gerar para o desenvolvimento, manutenção ou evolução de sistemas.

1.3 Objetivo Geral

O objetivo geral do trabalho é investigar como utilizar técnicas de análise estática automática de código-fonte para diminuir os custos relacionados às tarefas de ajuste de desempenho e manutenção de procedimentos armazenados em banco de dados. Para tal, é proposta uma abordagem para o desenvolvimento de analisadores estáticos automáticos para a checagem de conformidade do código-fonte de procedimentos de banco de dados com diretrizes de eficiência e qualidade.

1.4 Objetivo Específicos

Neste trabalho, propõe-se investigar e propor técnicas que facilitem as tarefas relacionadas ao ajuste de desempenho e à manutenção código-fonte de procedimentos armazenados em banco de dados. É fato que diferentes diretrizes de eficiência podem apresentar diferentes impactos no código-fonte de procedimentos de banco de dados. Além disso, existem várias linguagens de programação disponíveis para implementação de tais procedimentos. Dado esses fatores, propõe-se realizar os seguintes objetivos específicos:

- investigar na literatura trabalhos relacionados ao desempenho e à manutenção de código-fonte de procedimentos armazenados em banco de dados;
- especificar análises estáticas para a checagem da conformidade do código-fonte de procedimentos de banco de dados com diretrizes de eficiência e qualidade;
- propor uma abordagem que forneça a estrutura necessária para o desenvolvimento de analisadores estáticos automáticos para linguagens de procedimentos de banco de dados;

- realizar experimentos para medir o impacto das diretrizes de eficiência investigadas no trabalho;
- desenvolver um analisador estático para uma linguagem de procedimentos de banco de dados específica com base na abordagem proposta;
- avaliar a abordagem proposta no contexto de projetos reais que utilizem procedimentos de banco de dados.

1.5 Solução

Neste trabalho, é proposta uma abordagem para análise automática da conformidade do código-fonte de procedimentos armazenados em banco de dados com diretrizes pré-definidas de eficiência e qualidade. A abordagem baseia-se em análise estática de código-fonte; de maneira geral, a técnica realiza análise sintática do código-fonte de um procedimento e transforma-o em um estrutura de dados que permita a realização de buscas hierárquicas nos comandos presentes no procedimento. Em posse desta estrutura, a abordagem realiza análises no intuito de detectar padrões no código-fonte que não estejam de acordo com as regras definidas pelas diretrizes de eficiência e qualidade. O resultado desta etapa resulta em resumos de análises, os quais armazenam possibilidades de aplicação de diretrizes e suas respectivas localizações no código-fonte. Por fim, estes resumos são processados e mapeados em advertências de melhorias que são repassadas para o usuário da abordagem (um desenvolvedor de procedimentos de banco de dados). Um comparativo entre a abordagem proposta com outros trabalhos disponíveis na literatura relacionada é discutido no Capítulo 6.

Devido a algumas limitações da abordagem e dos recursos disponíveis em análises puramente estáticas, a abordagem pode reportar advertências falso-positivas, ou seja, cujas melhorias não são válidas no contexto da aplicação ou cujo impacto não justifique uma mudança no código-fonte. Neste sentido, este trabalho também propõe heurísticas automáticas no intuito de diminuir as chances de serem criadas advertências falso-positivas durante o processo realizado pela abordagem.

São discutidas diversas diretrizes de qualidade e eficiência para procedimentos de banco

de dados apresentadas na literatura relacionada [4] [5] [12] [9] [13]. Adicionalmente, são criadas possíveis estratégias para a checagem automática destas diretrizes no código-fonte de procedimentos de banco de dados. Estas estratégias são apresentadas na forma de análises estáticas, cujas condições são especificadas neste trabalho utilizando proposições da lógica de predicados.

Como prova de conceito, foi desenvolvida uma ferramenta, denominada PL/SQL Advisor, que instancia a abordagem proposta no contexto de uma linguagem de programação de banco de dados específica.

1.6 Avaliação

Primeiramente, foram realizados experimentos de análise de desempenho com o objetivo de mensurar o impacto individual de algumas das diretrizes de eficiência discutidas neste trabalho na execução de procedimentos de banco de dados. O planejamento experimental da análise de desempenho possui características complementares aos realizados por alguns trabalhos [4] [5] [12] [9] [13], uma vez que o mesmo considerou fatores como a quantidade de chamadas aos procedimentos e repetições dos tratamentos. Para as maiores cargas de trabalho testadas, algumas diretrizes de eficiência acarretaram otimizações superiores a 80% no tempo de execução dos procedimentos.

Em seguida, foi conduzido um estudo de caso utilizando procedimentos de banco de dados de 4 projetos de código-fonte aberto reais. A instanciação (ferramenta PL/SQL Advisor) da abordagem proposta foi utilizada para analisar automaticamente 10 procedimentos de cada projeto selecionado. Neste processo, a ferramenta reportou 299 advertências, tanto relacionadas a linguagens de programação em geral, quanto específicas para o contexto de banco de dados. No estudo de caso, a ferramenta desenvolvida necessitou de menos de 7 segundos para analisar o código-fonte de todos os procedimentos de banco de dados.

Por último, foram realizados experimentos para comparar a eficácia e a eficiência da abordagem proposta para análise de procedimentos de banco de dados com a eficácia e eficiência de inspeções manuais realizadas por desenvolvedores. Para tal, foram utilizados procedimentos de banco de dados de um projeto industrial real (projeto *Malha Brasil*). Os mesmos procedimentos de banco de dados selecionados do projeto foram analisados pe-

los desenvolvedores e pela ferramenta desenvolvida. Nestes experimentos, a abordagem proposta apresentou eficácia e eficiência superiores a todos os desenvolvedores do projeto; tanto em média quando em comparações individuais. Além disso, apenas uma parte consideravelmente pequena (13% em média) das advertências reportadas pela ferramenta foram consideradas falso-positivas pelos desenvolvedores do projeto industrial.

1.7 Principais Contribuições

Em resumo, as principais contribuições deste trabalho são:

- uma abordagem [22] [21], baseada em análise estática de código-fonte, para automatizar a checagem de conformidade de procedimentos de banco de dados com diretrizes pré-definidas de eficiência e qualidade, apresentada no Capítulo 3;
- descrição de estratégias, baseadas em análise estática automática de código-fonte, para detecção de oportunidades para aplicação de diretrizes de eficiência e qualidade específicas para o contexto de banco de dados; apresentadas no Capítulo 4 e Apêndice A;
- proposição de heurísticas para diminuição das chances de criação de advertências falso-positivas; apresentadas no Capítulo 4 e Apêndice A;
- uma ferramenta para diminuir os custos de análise de código-fonte de procedimentos de banco de dados escritos em PL/SQL [27];
- resultados da análise de impacto de diretrizes de eficiência específicas no tempo de execução de procedimentos de banco de dados, apresentados na Seção 5.2;
- avaliação da eficácia e eficiência da abordagem proposta utilizando procedimentos de projetos de código-fonte aberto reais (Seção 5.3);
- comparação da eficácia e eficiência da abordagem proposta com a eficácia e eficiência de inspeções manuais no contexto procedimentos de banco de dados de um projeto industrial (Seção 5.4);

- discussão sobre as classes de problemas mais presentes em procedimentos de banco de dados de código-fonte aberto e industriais; como também, possíveis explicações para a ocorrência das classes de problemas.

1.8 Organização

No Capítulo 2, é apresentada a fundamentação teórica necessária para o entendimento do trabalho. No Capítulo 3, é proposta uma abordagem para realizar análise estática automática do código-fonte de procedimentos armazenados em banco de dados.

As diretrizes de eficiência e qualidade para procedimentos de banco de dados investigadas neste trabalho são apresentadas no Capítulo 4 e Apêndice A. Em seguida (Capítulo 5), são descritas a implementação de uma instanciação da abordagem proposta e a avaliação da mesma. Por fim (Capítulo 6), são apresentadas as principais conclusões do trabalho, além de perspectivas para trabalhos futuros e um resumo de trabalhos relacionados.

Capítulo 2

Fundamentação Teórica

Nesse capítulo, é apresentada a fundamentação teórica necessária para o entendimento do trabalho desenvolvido. Inicialmente, é discutida uma visão geral de procedimentos armazenados em bancos de dados (Seção 2.1). Na Seção 2.2, é apresentada a definição de analisadores estáticos automáticos e conceitos de estruturas de dados relacionados à análise estática, tais como Grafo de Fluxo de Controle e Árvore de Controle de Dependência.

2.1 Stored Procedures

Stored Procedures representam subrotinas que são armazenadas na própria camada de dados. Estas subrotinas podem armazenar parte da lógica de negócio das aplicações e são usualmente utilizadas para tarefas como validação de dados e mecanismos de controle de acesso a dados [1]. Dessa maneira, processamentos que exigem muitos acessos a bancos de dados podem ser executados diretamente na camada de dados, podendo trazer diversos benefícios para as aplicações (Capítulo 1). *Stored Procedures* podem conter tanto comandos de linguagens de consulta e manipulação de dados (*DML statements*) quanto comandos procedimentais, tais como IF, WHILE, LOOP, REPEAT, CASE, dentre outros. A sintaxe destes comandos depende da linguagem de programação em que os procedimentos de banco de dados são escritos, uma vez que as linguagens variam de acordo com o SGBD. Por exemplo, algumas linguagens bastante conhecidas para o desenvolvimento de procedimentos de banco de dados são: PL/SQL [27] (SGBD Oracle), Transact-SQL [15] (SGBD SQL Server) e PL/pgSQL [1] (SGBD PostgreSQL).

Stored Procedures podem ser armazenadas na forma de procedimentos ou funções. A principal diferença entre estas duas formas de armazenamento é devido ao fato de funções sempre retornarem um valor. Desse modo, funções de banco de dados podem ser utilizadas como quaisquer outras expressões em comandos SQL. Por outro lado, um procedimento de banco de dados não retorna um valor e sua chamada é usualmente realizada utilizando uma das seguintes sintaxes:

- CALL *procedure_name*(parameter_list..);
- EXEC[UTE] *procedure_name*(parameter_list..);
- *procedure_name*(parameter_list..);

Por questões de simplificação, o termo procedimento é utilizado neste trabalho para denominar um procedimento ou função armazenada em banco de dados. Procedimentos de banco de dados podem retornar, por exemplo, múltiplos resultados provenientes de um comando SELECT. Estes resultados podem ser processados por cursores (exemplo mostrado no Código-fonte 2.1), por outros procedimentos ou pela camada de aplicação. Procedimentos de banco de dados também podem declarar variáveis para armazenar temporariamente os dados processados e receber parâmetros (de entrada e/ou saída).

Procedimentos de banco de dados podem ser utilizados de duas maneiras distintas: *procedimentos nomeados* e *blocos anônimos*. Da primeira maneira, é necessário que o procedimento seja criado armazenado fisicamente no dicionário de dados do SGBD e, consequentemente, sejam associadas informações específicas ao mesmo, tais como nome, esquema de banco de dados e permissões de acesso. Da segunda maneira, o procedimento pode ser executado, mas não é armazenado de fato no banco de dados; como consequência, um bloco anônimo não pode ser chamado por outros procedimentos ou pela camada de aplicação. Por exemplo, o Código-fonte 2.1 representa um bloco anônimo na linguagem T-SQL [15]. Analogamente, o Código-fonte 2.2 representa um bloco anônimo na linguagem PL/SQL.

```
1 DECLARE @Site NVARCHAR(128)
2     SET @Site='mysite'
3     SELECT      i_SiteID
4     FROM        Sites
5     WHERE       s_Name=@Site
```

Código-fonte 2.1: Exemplo de bloco anônimo na linguagem T-SQL.

```
1 DECLARE
2   CURSOR square_cur IS SELECT side1, side2 FROM square_table;
3   BEGIN
4     FOR square_rec IN square_cur LOOP
5       dbms_output.put_line('square area: ' || square_rec.side1 *
            square_rec.side2 );
6     END LOOP;
7   END;
```

Código-fonte 2.2.: Exemplo de bloco anônimo na linguagem PL/SQL.

2.2 Análise Estática de Código-fonte

Nesta seção, são apresentados alguns conceitos relacionados à análise estática de código-fonte.

2.2.1 Analisadores Estáticos

Neste trabalho, propõe-se a utilização de técnicas automáticas de análise estática de código-fonte. Um analisador estático é um software que analisa o código-fonte de programas, sem de fato executá-lo, no intuito de identificar defeitos, anomalias, inconformidade com padrões, más práticas de programação e ineficiências. A intenção da análise estática automática é chamar a atenção dos inspetores de código-fonte para fatores como: variáveis utilizadas sem serem inicializadas, variáveis não utilizadas, variáveis assumindo valores além do limite permitido, código inalcançável, dentre outros problemas [28]. Analisadores estáticos não são capazes de substituir completamente inspeções manuais. Isso ocorre porque tais analisadores podem identificar apenas um conjunto fixo de padrões ou regras no código-fonte. No entanto, acredita-se que o processo de inspeção de procedimentos de banco de dados pode ser bastante otimizado se assistido por técnicas automáticas de análise estática.

2.2.2 Grafo de Fluxo de Controle

Um *Grafo de Fluxo de Control* (GFC) [20] [30] é um grafo direcionado que representa as entidades contidas no código-fonte na forma de nodos e os possíveis caminhos da execução do código-fonte na forma de arestas. Em geral, um GFC representa um conjunto de localidades de um programa em um único estado abstrato (nodo). As arestas do GFC representam a possibilidade da execução do programa seguir do fim de uma região para o início de outra, sendo esta sequência realizada por meio de uma execução sequencial ou um desvio (condicional ou não condicional) no código-fonte [30]. Um nodo em um GFC pode representar um comando único ou mesmo uma operação única de baixo nível. No entanto, por questões de simplificação, os nodos em um GFC usualmente representam um bloco básico, o qual representa toda uma região com uma única possibilidade de entrada e uma única possibilidade de saída na execução do programa. Um ponto importante a ser destacado é o fato de que, uma vez que o código-fonte de um programa é finito, o número de nodos de um GFC é também sempre finito.

A representação de código-fonte na forma de um Grafo de GFC possui inúmeras aplicações na área de compiladores e análise estática. Por exemplo, o GFC permite a verificação de propriedades de alcançabilidade e dominação das regiões do código-fonte [30]. Além disso, muitas representações de código-fonte derivam da definição de GFC, por exemplo: *Grafo de Dependência de Dados*, *Grafo de Controle Inter Procedural* e *Árvore de Controle de Dependência*. A seguir, descrevemos a estrutura de dados *Árvore de Controle de Dependência*.

2.2.3 Árvore de Controle de Dependência

A definição de *Árvore de Controle de Dependência* (ACD) deriva da definição de *dominância* (ou *Dom*) [19] [30] [20] em um GFC. Em um GFC cujo ponto de entrada é o nodo $\mathbf{N_0}$, a definição de dominância pode ser organizada da seguinte forma:

- Um nodo $\mathbf{N}$ que é atingível por todos os caminhos possíveis de $\mathbf{N_0}$ até $\mathbf{N}$ é dominado por $\mathbf{N_0}$;
- Para qualquer outro nodo $\mathbf{M}$ que seja atingível por apenas parte dos caminhos possíveis de $\mathbf{N_0}$ até $\mathbf{M}$, existe um nodo $\mathbf{N}$ que controla execução de $\mathbf{M}$ no sentido que a

execução de $\mathbf{M}$ depende do resultado da execução de $\mathbf{N}$. Neste caso, diz-se que $\mathbf{N}$ domina $\mathbf{M}$ ou $\mathbf{N} \in \mathit{Dom}(\mathbf{M})$;

- Cada nodo $\mathbf{N}$ domina a si próprio, ou seja, $\mathbf{N} \in \mathit{Dom}(\mathbf{N})$;

Com base na definição de dominância, derivam-se as seguintes definições:

- Um nodo $\mathbf{A}$ *domina estritamente* (ou *eDom*) um nodo $\mathbf{M}$ se: $\mathbf{A} \in \mathit{Dom}(\mathbf{M})$ e $\mathbf{A} \neq \mathbf{M}$;
- O *dominador imediato* ou *iDom* de um nodo $\mathbf{M}$ é o único nodo que domina estritamente $\mathbf{M}$ mas não domina estritamente nenhum outro nodo que domine estritamente $\mathbf{M}$; ou seja: $\mathit{iDom}(\mathbf{M}) = \mathbf{N} \Leftrightarrow \mathbf{N} \in \mathit{eDom}(\mathbf{M})$ e $\nexists\ \mathbf{K}\ /\ \mathbf{K} \in \mathit{eDom}(\mathbf{M})$ e $\mathbf{N} \in \mathit{eDom}(\mathbf{K})$;

Sendo $\mathbf{N_0}$ o nodo que representa o ponto de entrada de um GFC $\mathbf{G_1}$, a estrutura hierárquica da ACD gerada a partir de $\mathbf{G_1}$ pode ser definida da seguinte forma:

- $\mathit{parent}(\mathbf{N_0}) = \emptyset$
- $\mathit{iDom}(\mathbf{M}) = \mathbf{N} \Rightarrow \mathbf{N} = \mathit{parent}(\mathbf{M})$;

Uma vez que cada nodo, com exceção da raiz, tem um único dominador imediato, a relação de dominação imediata entre os nodos de um GFC irá sempre formar uma árvore, a qual é denominada Árvore de Controle de Dependência.

No Pseudocódigo 2.1 é apresentado um exemplo prático de um processamento realizado por um procedimento de banco de dados. O Pseudocódigo 2.1 itera, utilizando um cursor explícito, sobre os dados da tabela *employees* no intuito de inserir em outra tabela (*retired_employees*) os funcionários que estão legitimados a se aposentarem. Na Figura 2.1, é apresentado o *GFC* que representa o Pseudocódigo 2.1. Na Figura 2.2 é apresentada a *ACD* que representa o Pseudocódigo 2.1. Para simplificar, os comandos do Pseudocódigo 2.1 são referenciados pelas linhas do código-fonte nas quais estão inseridos.

```
1  PROCEDURE ELIGIBLE_EMPLOYEES
2
3      count_employees INTEGER = 0
4      emp_id INTENGER
```

```
5       emp_name CHAR(80)
6       years_worked INTEGER
7       employees_cursor = SELECT ID, NAME, YEARS_WORKED FROM EMPLOYEE_TABLE
8
9       OPEN employees_cursor
10      WHILE HAS_NEXT(employees_cursor)
11        SELECT NEXT(employees_cursor) INTO emp_id, emp_name, years_worked
12        IF (employee_score(emp_id) > 5 AND years_worked > 30)
13            INSERT INTO ELIGIBLE_EMPLOYEES VALUES (emp_id, emp_name,
                  years_worked)
14            count = count + 1
15            commit
16    PRINT ('# Eligible Employees: ' + count_employees)
```

Pseudocódigo 2.1: Processamento do procedimento de banco de dados ELIGIBLE_EMPLOYEES.

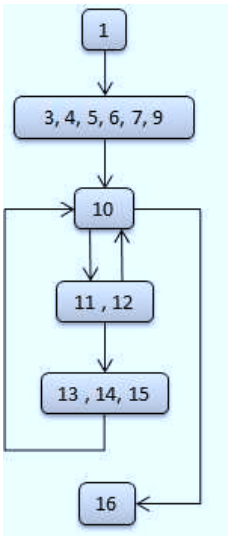

Figura 2.1: Representação do Pseudocódigo 2.1 na forma de um GFC.

Analisando a estrutura do GFC mostrado na Figura 2.1 é possível inferir as seguintes propriedades sobre a relação de dominância entre as entidades do Pseudocódigo 2.1:

- $Dom(1) = \{1\}$ $(\mathbf{a.i})$
- $Dom(3) = \{3, 1\}$ $(\mathbf{a.ii})$

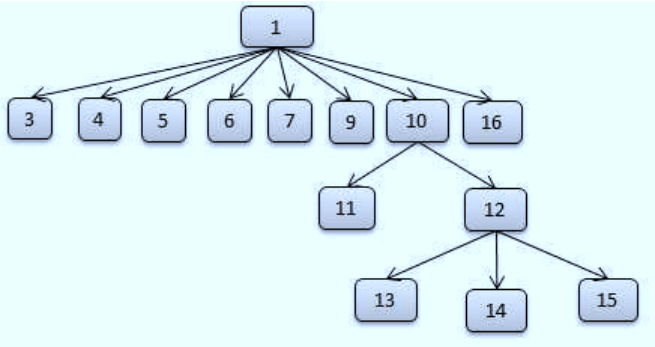

Figura 2.2: Representação do Pseudocódigo 2.1 na forma de uma ACD.

- *Dom*(4) $= \{4, 1\}$ $(\mathbf{a.iii})$
- *Dom*(5) $= \{5, 1\}$ $(\mathbf{a.iv})$
- *Dom*(6) $= \{6, 1\}$ $(\mathbf{a.v})$
- *Dom*(7) $= \{7, 1\}$ $(\mathbf{a.vi})$
- *Dom*(9) $= \{9, 1\}$ $(\mathbf{a.vii})$
- *Dom*(10) $= \{10, 1\}$ $(\mathbf{a.viii})$
- *Dom*(11) $= \{11, 10, 1\}$ $(\mathbf{a.ix})$
- *Dom*(12) $= \{12, 10, 1\}$ $(\mathbf{a.x})$
- *Dom*(13) $= \{13, 12, 10, 1\}$ $(\mathbf{a.xi})$
- *Dom*(14) $= \{14, 12, 10, 1\}$ $(\mathbf{a.xii})$
- *Dom*(15) $= \{15, 12, 10, 1\}$ $(\mathbf{a.xiii})$
- *Dom*(16) $= \{16, 1\}$ $(\mathbf{a.xiv})$

Utilizando as propriedades a.z / z $\in$ {i, ii, ... , xiv}, é possível inferir as seguintes propriedades sobre a relação de dominância estrita entre as entidades do Pseudocódigo 2.1:

- $(\mathbf{a.i}) \Rightarrow$ *eDom*$(1) = \emptyset$ $(\mathbf{b.i})$
- $(\mathbf{a.ii}) \Rightarrow$ *eDom*$(3) = \{1\}$ $(\mathbf{b.ii})$

- $(\mathbf{a.iii}) \Rightarrow eDom(4) = \{1\}$ $(\mathbf{b.iii})$
- $(\mathbf{a.vi}) \Rightarrow eDom(5) = \{1\}$ $(\mathbf{b.iv})$
- $(\mathbf{a.v}) \Rightarrow eDom(6) = \{1\}$ $(\mathbf{b.v})$
- $(\mathbf{a.vi}) \Rightarrow eDom(7) = \{1\}$ $(\mathbf{b.vi})$
- $(\mathbf{a.vii}) \Rightarrow eDom(9) = \{1\}$ $(\mathbf{b.vii})$
- $(\mathbf{a.viii}) \Rightarrow eDom(10) = \{1\}$ $(\mathbf{b.viii})$
- $(\mathbf{a.ix}) \Rightarrow eDom(11) = \{10, 1\}$ $(\mathbf{b.ix})$
- $(\mathbf{a.x}) \Rightarrow eDom(12) = \{10, 1\}$ $(\mathbf{b.x})$
- $(\mathbf{a.xi}) \Rightarrow eDom(13) = \{12, 10, 1\}$ $(\mathbf{b.xi})$
- $(\mathbf{a.xii}) \Rightarrow eDom(14) = \{12, 10, 1\}$ $(\mathbf{b.xii})$
- $(\mathbf{a.xiii}) \Rightarrow eDom(15) = \{12, 10, 1\}$ $(\mathbf{b.xiii})$
- $(\mathbf{a.xiv}) \Rightarrow eDom(16) = \{1\}$ $(\mathbf{b.xiv})$

Utilizando as propriedades a.z / z ∈ {i, ii, ... , xiv} e b.y / y ∈ {i, ii, ... , xiv}, é possível inferir as seguintes propriedades sobre a relação de dominância imediata (*iDom*) entre as entidades do Pseudocódigo 2.1:

- $(\mathbf{b.i}) \Rightarrow iDom(1) = \emptyset$ $(\mathbf{c.i})$
- $(\mathbf{b.i})$, $(\mathbf{b.ii}) \Rightarrow iDom(3) = \{1\}$ $(\mathbf{c.ii})$
- $(\mathbf{b.i})$, $(\mathbf{b.iii}) \Rightarrow iDom(4) = \{1\}$ $(\mathbf{c.iii})$
- $(\mathbf{b.i})$, $(\mathbf{b.iv}) \Rightarrow iDom(5) = \{1\}$ $(\mathbf{c.iv})$
- $(\mathbf{b.i})$, $(\mathbf{b.v}) \Rightarrow iDom(6) = \{1\}$ $(\mathbf{c.v})$
- $(\mathbf{b.i})$, $(\mathbf{b.vi}) \Rightarrow iDom(7) = \{1\}$ $(\mathbf{c.vi})$
- $(\mathbf{b.i})$, $(\mathbf{b.vii}) \Rightarrow iDom(9) = \{1\}$ $(\mathbf{c.vii})$

- $(\mathbf{b.i}), (\mathbf{b.viii}) \Rightarrow \textit{iDom}(10) = \{1\}\ (\mathbf{c.viii})$
- $(\mathbf{b.i}), (\mathbf{b.viii}), (\mathbf{b.ix}) \Rightarrow \textit{iDom}(11) = \{10\}\ (\mathbf{c.ix})$
- $(\mathbf{b.i}), (\mathbf{b.viii}), (\mathbf{b.x}) \Rightarrow \textit{iDom}(12) = \{10\}\ (\mathbf{c.x})$
- $(\mathbf{b.i}), (\mathbf{b.viii}), (\mathbf{b.x}), (\mathbf{b.xi}) \Rightarrow \textit{iDom}(13) = \{12\}\ (\mathbf{c.xi})$
- $(\mathbf{b.i}), (\mathbf{b.viii}), (\mathbf{b.x}), (\mathbf{b.xii}) \Rightarrow \textit{iDom}(14) = \{12\}\ (\mathbf{c.xii})$
- $(\mathbf{b.i}), (\mathbf{b.viii}), (\mathbf{b.x}), (\mathbf{b.xiii}) \Rightarrow \textit{iDom}(15) = \{12\}\ (\mathbf{c.xiii})$
- $(\mathbf{b.i}), (\mathbf{b.xiv}) \Rightarrow \textit{iDom}(16) = \{1\}\ (\mathbf{c.xiv})$

Utilizando as propriedades c.w / w $\in$ {i, ii, ... , xiv}, é possível inferir as seguintes propriedades sobre a estrutura hierárquica entre as entidades do Pseudocódigo 2.1 em sua representação na forma de ACD:

- $(\mathbf{c.i}) \Rightarrow \textit{iDom}(1) = \emptyset \Rightarrow parent(1) = \emptyset\ (\mathbf{d.i})$
- $(\mathbf{c.ii}) \Rightarrow \textit{iDom}(3) = 1 \Rightarrow parent(3) = 1\ (\mathbf{d.ii})$
- $(\mathbf{c.iii}) \Rightarrow \textit{iDom}(4) = 1 \Rightarrow parent(4) = 1\ (\mathbf{d.iii})$
- $(\mathbf{c.iv}) \Rightarrow \textit{iDom}(5) = 1 \Rightarrow parent(5) = 1\ (\mathbf{d.iv})$
- $(\mathbf{c.v}) \Rightarrow \textit{iDom}(6) = 1 \Rightarrow parent(6) = 1\ (\mathbf{d.v})$
- $(\mathbf{c.vi}) \Rightarrow \textit{iDom}(7) = 1 \Rightarrow parent(7) = 1\ (\mathbf{d.vi})$
- $(\mathbf{c.vii}) \Rightarrow \textit{iDom}(9) = 1 \Rightarrow parent(9) = 1\ (\mathbf{d.vii})$
- $(\mathbf{c.viii}) \Rightarrow \textit{iDom}(10) = 1 \Rightarrow parent(10) = 1\ (\mathbf{d.viii})$
- $(\mathbf{c.ix}) \Rightarrow \textit{iDom}(11) = 10 \Rightarrow parent(11) = 10\ (\mathbf{d.ix})$
- $(\mathbf{c.ix}) \Rightarrow \textit{iDom}(12) = 10 \Rightarrow parent(12) = 10\ (\mathbf{d.x})$
- $(\mathbf{c.xi}) \Rightarrow \textit{iDom}(13) = 12 \Rightarrow parent(13) = 12\ (\mathbf{d.xi})$
- $(\mathbf{c.xii}) \Rightarrow \textit{iDom}(14) = 12 \Rightarrow parent(14) = 12\ (\mathbf{d.xii})$

- $(\mathbf{c}.\mathbf{xiii}) \Rightarrow$ *iDom*(15) $= 12 \Rightarrow parent(15) = 12$ $(\mathbf{d}.\mathbf{xiii})$
- $(\mathbf{c}.\mathbf{xiiv}) \Rightarrow$ *iDom*(16) $= 1 \Rightarrow parent(16) = 1$ $(\mathbf{d}.\mathbf{xiv})$

Utilizando as propriedades d.x / x $\in$ {i, ii, ... , xiv}, é possível criar a representação do Pseudocódigo 2.1 na forma de uma ACD, a qual é mostrada na Figura 2.2.

Capítulo 3

Uma Abordagem para Análise Estática Automática de Procedimentos Armazenados em Bancos de Dados

Como discutido no Capítulo 1, existe uma série de desvantagens relacionadas à utilização de inspeções manuais de código-fonte de procedimentos armazenados em banco de dados. Neste capítulo, descrevemos uma abordagem para análise estática automática de código-fonte de procedimentos armazenados em banco de dados e sugestão melhorias com base em diretrizes pré-determinadas. A abordagem proposta foca na análise da parte procedimental de procedimentos de banco de dados e utiliza uma representação do código-fonte de procedimentos na forma de Árvore de Controle de Dependência (Capítulo 2). Com base nesta abstração, são realizadas análises para checar a conformidade do código-fonte de procedimentos de banco de dados com diretrizes pré-definidas de eficiência e qualidade. O resultado destas análises é então utilizado para criar sugestões de alterações que potencialmente melhorem o código-fonte analisado.

Inicialmente, justificamos a estratégia utilizada para representação do código-fonte de procedimentos armazenados em banco de dados (Seção 3.1). Na Seção 3.2 é apresentada a visão geral da abordagem para realização de análise estática em procedimentos de banco de dados. São também detalhadas cada etapa realizada na abordagem. Na Seção 3.3 é apresentado um exemplo de utilização da abordagem para a checagem de uma diretriz de eficiência. Finalmente, as limitações da abordagem são discutidas na Seção 3.5.

3.1 Representação do Código-fonte de Procedimentos de Banco de Dados

A estratégia para representação do código-fonte de procedimentos de banco de dados adotada pela abordagem proposta foi a estrutura de dados *Árvore de Controle de Dependência* (ACD), apresentada no Capítulo 2. A estrutura de dados ACD foi escolhida pelos seguintes motivos:

- **Simplicidade**. A estrutura de dados Árvore de Controle de Dependência é uma representação de código-fonte de baixa complexidade quando comparada à outras estratégias existentes [6] [24]. Além disso, a forma de representação adotada simplificou o desenvolvimento de um protótipo como prova de conceito da abordagem apresentada.

- **Representação Hierárquica**. A representação do código-fonte na estrutura de uma árvore simplifica a checagem de diretrizes pré-definidas. Este fato ocorre porque muito frequentemente, apenas uma parte do código-fonte que é dominada por algum comando precisa ser analisada. Neste cenário, a análise pode restringir-se aos nodos descendentes do nodo dominador representado na ACD.

- **Definição formal**. A definição da Árvore de Controle de Dependência baseia-se no modelo matemático e bem definido da teoria dos grafos. A definição formal também diminui as chances de interpretações ambíguas sobre a forma como a abordagem apresentada propõe representar o código-fonte sob análise.

3.2 Visão Geral da Abordagem

Nesta seção, descrevemos a visão geral da abordagem proposta para realizar análise estática no código-fonte de procedimentos armazenados em banco de dados. O objetivo da abordagem é realizar análise automática no código-fonte de procedimentos no intuito de verificar a conformidade dos mesmos com diretrizes de eficiência e qualidade de código. Além disso, a abordagem visa apresentar possíveis soluções para as situações em que o código-fonte não segue as diretrizes pré-definidas.

A abordagem é dividida em 6 etapas sequenciais, ilustradas na Figura 3.1. Na Etapa 1, a abordagem recebe como entrada o código-fonte de um procedimento de banco de dados *C* escrito na linguagem de programação *L*. Na Etapa 2, o código-fonte *C* é processado por um analisador sintático (*parser*) desenvolvido para a linguagem de programação *L*. Nesta etapa, o analisador sintático utiliza um analisador léxico (também desenvolvido para a linguagem *L*) para processar as palavras do código-fonte *C*. Uma vez que o objetivo da abordagem não é compilar o código-fonte, é requerido que o código-fonte *C* esteja sintaticamente correto. Caso contrário, o processo realizado pela abordagem pode reportar saídas equivocadas. Ao final da Etapa 2, é criada uma representação do código-fonte *C* na forma de uma ACD.

Na Etapa 3, uma vez que o código-fonte *C* está inteiramente representado como uma *ACD*, esta representação é analisada por dois componentes: *Analisador de Diretrizes de Eficiência* (ADE) e *Analisador de Diretrizes de Qualidade de Código* (ADQC). Na implementação destes dois componentes, algoritmos clássicos de buscas em árvores, tais como *Busca em Largura* e *Busca em Profundidade* podem ser utilizados para procurar informações específicas ou padrões pré-determinados nos nodos da ACD que representam o código-fonte C. Na Etapa 4, os componentes *ADE* e *ADQC* retornam (cada um deles) um resumo do resultado de suas respectivas análises, os quais podem conter possíveis inconformidades do código-fonte C com as diretrizes pré-definidas e suas respectivas localizações na ACD. Uma vez que existem diretrizes que são específicas para algumas linguagens de programação de banco de dados, a implementação dos componentes *ADE* e *ADQC* é dependente da linguagem de programação em que os códigos-fonte recebidos são escritos. Exemplos destas especificidades em linguagens procedimentais de banco de dados são apresentados no Capítulo 4 e Apêndice A.

Na Etapa 5, os resumos das análises dos componentes *ADE* e *ADQC* são unidos e analisados por um componente denominado *Consultor de Melhorias*. Este componente é responsável por processar o resumo das análises estáticas e decidir quais inconformidades com diretrizes serão reportados na forma de advertências para os usuários da abordagem. Nesta etapa, são também realizados filtros nas advertências criadas. Este passo é importante pois há situações em que mais de uma advertência pode estar relacionada com o mesmo comando do código-fonte. Nestes casos, o *Consultor de Melhorias* deve avaliar qual advertência é mais adequada para cada comando (ou conjunto de comandos). Na Seção 3.4 é detalhada a

estratégia para o processamento do resumo da análise utilizada pela abordagem. Na Etapa 6, o *Consultor de Melhorias* ordena as advertências de acordo com algum critério de ordenação e as redireciona para a saída do processo. Na Seção 3.4 são discutidas possíveis estratégias para ordenação das advertências reportadas pela abordagem. Por fim, as advertências são analisadas pelo usuário interessado no processamento realizado pela abordagem.

A abordagem apresentada é generalizável para a realização de análise estática em código-fonte de procedimentos armazenados em banco de dados escritos em quaisquer linguagens de programação de banco de dados. No entanto, a implementação de determinados componentes (ADE, ADQC e Consultor de Melhorias) descritos na abordagem são dependentes da linguagem de programação do código-fonte a ser analisado. A seguir, detalhamos cada etapa da abordagem apresentada.

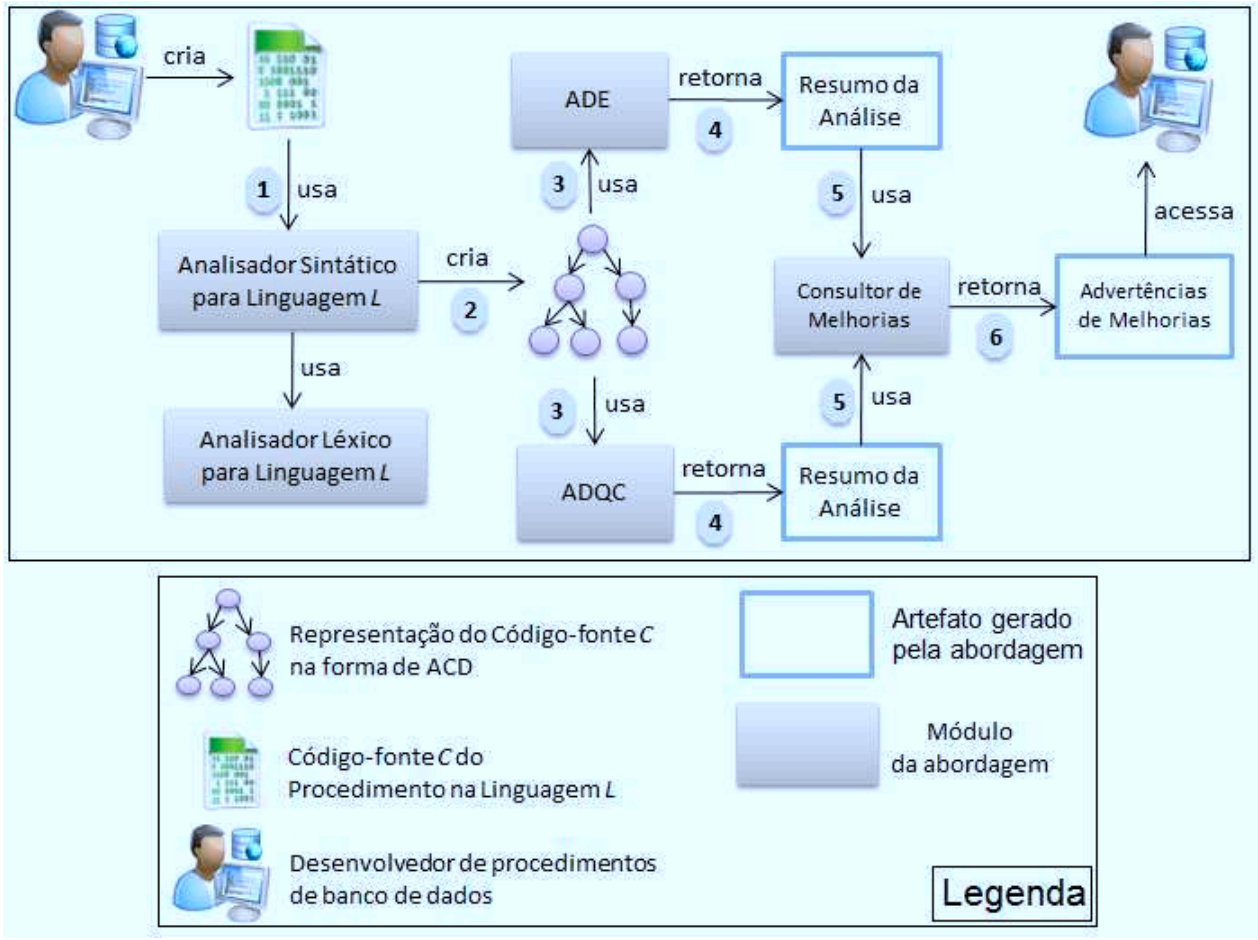

Figura 3.1: Visão geral da Abordagem para realizar análise estática em procedimentos armazenados em banco de dados.

3.2.1 Etapa 1: Análises Léxica e Sintática do Código-fonte

Esta etapa tem como objetivo realizar as análises léxica e sintática do código-fonte recebido como entrada pela abordagem. Uma vez que o resultados destas análises deve facilitar a realização da Etapa 2, que consiste em criar uma representação na forma de uma ACD do código-fonte, existem duas estratégias possíveis para realização da Etapa 1:

- **Estratégia 1**: Criação de um árvore sintática que armazene as entidades do código-fonte de forma estruturada. As informações da árvore sintática são então utilizadas para criação de uma estrutura que represente as entidades do código-fonte e os possíveis fluxos de execução entre as entidades. Como discutido no Capítulo 2, esta representação é denominada *Grafo de Fluxo de Controle* (GFC).

- **Estratégia 2**: Durante o processamento da análise sintática, devem ser executadas rotinas que criem a representação do código-fonte na forma de um GFC à medida em que a árvore sintática é gerada.

É importante destacar que o GFC criado nesta etapa deve representar um único procedimento. Dessa forma, caso o código-fonte analisado possua declarações de subprocedimentos, cada subprocedimento deve ser representado por um GFC distinto e cada GFC é consumido pelo processamento realizado na Etapa 2.

3.2.2 Etapa 2: Criação da ACD

O objetivo desta etapa é criar uma representação na forma de ACD para o código-fonte recebido como entrada da abordagem na Etapa 1. No Capítulo 2, é apresentado um exemplo prático de como o código-fonte de um procedimento de banco de dados pode ser representado na forma de uma ACD com base em uma representação do procedimento na forma de um GFC.

3.2.3 Etapa 3: Análise Estática da ACD

O objetivo desta etapa é realizar um conjunto de análises estáticas sobre a representação na forma de ACD do código-fonte do procedimento de banco de dados e retornar um Resumo

de Análise, o qual deve conter os padrões no código-fonte que, de acordo com as diretrizes, podem gerar alguma advertência. O Resumo de Análise é composto por uma lista de 5-tuplas do tipo R_A<Tipo_Diretriz, Código_Diretriz, Pontuação_Diretriz, Identificador_Comando, Linha_do_Código_Fonte>, ou de forma simplificada R_A<T_D, P_D, C_D, I_C, L_C>. A descrição e a função de cada componente da 5-tupla R_A são apresentadas a seguir:

- **Tipo_Diretriz**: valor utilizado para identificar se a diretriz é de eficiência ou qualidade do código-fonte do procedimento. Este valor pode ser utilizado para realizar classificação e/ou priorização das advertências reportadas;
- **Código_Diretriz**: representa o identificador único da diretriz. Esta identificação pode ser utilizada para: 1) resolver conflitos de advertências relacionadas a mesma parte do código-fonte; e 2) contagem do número de advertências do mesmo tipo reportadas após a análise do código-fonte;
- **Pontuação_Diretriz**: pontuação da diretriz utilizada para resolver conflitos entre advertências relacionadas a mesma parte do código-fonte; na prática, diretrizes com maior pontuação devem possuir prioridade nos casos de conflitos.
- **Identificador_Comando**: identificação única dos comandos do código-fonte que permite a extração de informações para a criação das advertências, tais como: tipo do comando e a representação textual do comando;
- **Linha_do_Código_Fonte**: identificação da linha do código-fonte em que a diretriz foi detectada.

A lista de diretrizes de eficiência e qualidade de código que são investigadas neste trabalho é discutida em detalhes no Capítulo 4 e Apêndice A. A seguir, é apresentado um exemplo de como a representação do código-fonte de um procedimento armazenado em banco de dados na forma de uma ACD pode ser utilizada para a checagem automática de uma diretriz pré-definida de eficiência do código-fonte.

3.3 Exemplo de Aplicação da Abordagem

Nesta seção é apresentado um exemplo de análise estática sobre a estrutura de uma ACD. Realizando uma inspeção manual no Pseudocódigo 2.1 é possível detectar um série de problemas de eficiência. Por exemplo, um dos principais problemas de eficiência do Pseudocódigo 2.1 deve-se ao fato do laço contido na linha 10 executar um comando *commit* em cada uma de suas iterações. Esta característica pode tornar a execução do procedimento bastante ineficiente, uma vez que um comando *commit* precisa estabelecer uma conexão com o SGBD e encadear um processo para a persistência de dados no disco rígido ou arquivo de *log*. Uma única execução do comando *commit* no Pseudocódigo 2.1 (para isso, seria necessário remover o comando *commit* do escopo do laço *while*) tornaria a execução do procedimento mais eficiente. Sendo assim, uma possível diretriz de eficiência para o desenvolvimento de procedimentos de banco de dados é a remoção de comandos *commit* do escopo de comandos de iteração [13] [9]. Esta diretriz será abreviada utilizando o código de diretriz RED_COM.

O Pseudocódigo 3.1 ilustra a estratégia utilizada neste trabalho para a checagem automática da diretriz RED_COM sobre a representação do código-fonte de um procedimento de banco de dados na forma de uma ACD. Inicialmente, é realizada uma chamada à função *search_commits_within_loops* passando o nó raiz da ACD como o parâmetro *stmt*. Esta função realiza uma busca na estrutura hierárquica da ACD por algum comando de repetição. Caso um comando de repetição seja encontrado, a função *search_commit* é chamada passando como parâmetro cada um dos nodos filhos do comando de repetição detectado. Por sua vez, a função *search_commit* realiza uma busca hierárquica entre os filhos do nodo de repetição por um comando do tipo *commit*. Caso um comando deste tipo seja encontrado, este fato é reportado no Resumo da Análise.

```
1  search_commits_within_loops(stmt Statement)
2      IF TYPE(stmt) = LOOP
3          FOR EACH child in children(stmt)
4              search_commit(child)
5      FOR EACH child IN children(stmt)
6          search_commit_within_loops(child)
7  END
8
9  search_commit(stmt Statement)
```

```
10      IF TYPE(stmt) = COMMIT
11          analysis_summary.add(Efficiency, RED_COM, score(RED_COM),
                identifier(stmt), line(stmt))
12      FOR EACH child IN children(stmt)
13          search_commit(child)
14  END
```

Pseudocódigo 3.1: Pseudocódigo que ilustra a checagem automática da diretriz RED_COM.

3.4 Etapa 5: Processamento do Resumo da Análise Estática

A função desta etapa é processar o resumo da análise criado na Etapa 4 visando três objetivos específicos: 1) resolver conflitos entre as diretrizes pré-definidas detectadas; 2) criar advertências para melhorias no código-fonte; e 3) ordenar as advertências criadas de acordo com algum critério de ordenação.

A resolução dos conflitos entre as diretrizes detectadas é realizada utilizando o atributo P_D (Pontuação da diretriz) da 5-tupla R_A. Após a verificação das diretrizes presentes no *Resumo da Análise*, se existir mais de uma diretriz relacionada com o mesmo comando no código-fonte, a diretriz com maior pontuação é mantida para a criação da advertência, enquanto as demais diretrizes relacionadas ao mesmo comando são descartadas. Em caso em empate entre a pontuação de diretrizes, a abordagem utiliza uma estratégia conservativa, isto é, todas as diretrizes (com pontuação igual) relacionadas ao mesmo comando são mantidas no Resumo da Análise. A utilização da estratégia conservativa acarreta tanto pontos positivos quanto pontos negativos para a utilização prática da abordagem proposta. Esta estratégia elimina as chances da abordagem deixar de reportar uma advertência que possa ser útil para a melhoria do código, uma vez que todas diretrizes com mesmo valor de pontuação são consideradas na etapa de criação das advertências. Por outro lado, a estratégia conservativa aumenta as chances da abordagem reportar uma advertência falso-positiva, uma vez que nas situações em que diretrizes com a o mesmo valor de pontuação podem ser aplicadas na mesma parte do código, é possível que nem todas as advertências reportadas com base nestas

diretrizes sejam úteis para a melhoria do código.

O segundo objetivo desta etapa é criar as advertências que serão analisadas pelos usuários da abordagem. Para a implementação desta etapa deve ser definido um mapeamento entre cada diretriz pré-definida e uma mensagem de advertência. Exemplos deste tipo de mapeamento são apresentados no Capítulo 4.

Denominando o mapeamento entre diretriz-advertência como M_D_A, ao identificar uma diretriz C_D_n referente ao comando *stmt* no resumo da análise, o Consultor de Advertências cria uma mensagem de advertência do tipo: *Warning*<M_D_A(C_D_n), stmt, line(stmt)>. Ao final do processamento do resumo da análise, um conjunto de tuplas do tipo *Warning* são redirecionadas para o usuário.

O terceiro objetivo desta etapa é ordenar as advertências de acordo com algum critério de ordenação. As advertências devem ser apresentadas preferencialmente ao usuário da abordagem de maneira que minimize o esforço para a visualização e interpretação das mesmas. É preferível que as advertências mais importantes apareçam antes das demais na saída da abordagem. Naturalmente, a escolha do usuário deve ser levada em consideração para este objetivo, uma vez que diferentes usuários podem apresentar opiniões diferentes em relação à importância de cada um dos tipos de advertência reportadas pela abordagem. Alguns possíveis critérios de ordenação propostos que levam em consideração a preferência do usuário são:

- **Ordenação pelo tipo de advertência**: esta estratégia utiliza o valor do *Tipo da Diretriz* da 5-tupla R_A para ordenar as advertências. Em outras palavras, o usuário pode optar por priorizar as advertências de eficiência ou qualidade de código;

- **Ordenação por impacto individual da advertência**: esta estratégia exige que o usuário crie uma pontuação específica para cada diretriz analisada pela abordagem. A ideia consiste em adicionar prioridades na ordenação das diretrizes de acordo com o impacto das respectivas advertências geradas pelas mesmas no código-fonte. Por exemplo, o usuário pode optar por priorizar as advertências que causem maior impacto na eficiência do código-fonte ou que resultem em melhorias de legibilidade mais significativas.

- **Ordenação pela frequência da advertência**: esta estratégia prioriza as advertências mais frequentemente reportadas na saída da abordagem.

- **Ordenação pela complexidade da análise**: esta estratégia prioriza as advertências cujas diretrizes são mais difíceis de serem detectadas utilizando inspeções manuais. Uma vez que a dificuldade de detecção é um conceito relativo entre os usuários, esta estratégia exige que o usuário defina uma pontuação relacionada à complexidade de cada análise realizada pela abordagem.

Idealmente, deve ser disponibilizada ao usuário final da abordagem a possibilidade de escolha entre estas opções de ordenação das advertências reportadas. Na próxima seção são discutidas algumas limitações da análise automática realizada pela abordagem proposta.

3.5 Limitações da Abordagem Proposta

A abordagem apresentada neste capítulo apresenta algumas limitações em relação a sua aplicabilidade prática e teórica. A principal limitação teórica diz respeito à capacidade da análise automática realizada pela abordagem. Como visto, o código-fonte dos procedimentos de banco de dados são representados na forma de uma ACD. Ainda que possua algumas características desejáveis, esta pode ser considerada uma representação simplificada de código-fonte que apresenta várias limitações. Por exemplo, uma ACD não preserva informações a respeito do fluxo de dados que é manipulado no código-fonte. Esta característica em especial limita a capacidade da abordagem em relação a certas classes de análise específicas como, por exemplo, a análise do uso de memória pelos procedimentos de banco de dados.

3.5.1 Advertências Falso-Positivas

Uma vez que a abordagem analisa uma representação simplificada do código-fonte, sua capacidade de análise compromete-se em alguns aspectos específicos. Em certos casos, os algoritmos para checagem de conformidade dos procedimentos com as diretrizes precisam fazer suposições sobre o fluxo de execução ou de dados do código-fonte. Uma vez que estas suposições podem nem sempre retratar a realidade durante a execução do código-fonte, algumas advertências reportadas pela abordagem são consideradas falso-positivas. Este fato se deve a três possibilidades: *p1*) no contexto da aplicação, a suposição feita pelo algoritmo de checagem da diretriz não ocorre na prática; *p2*) a implementação da advertência repor-

tada pela abordagem pode apresentar um impacto considerável no pior caso, mas no caso médio ou no contexto da aplicação o impacto da mudança não justifica a sua implementação; ou *p3*) a diretriz identificada pela abordagem não faz sentido para o contexto específico da aplicação ou o usuário final não concorda com a definição da diretriz. Ainda que a terceira possibilidade ocorra por um motivo fora do controle da abordagem, os dois primeiros casos ocorrem devido a uma limitação da mesma e devem ser minimizados para evitar a ocorrência de esforços desnecessários pelos usuários finais na interpretação de advertências falso-positivas.

As três possibilidades para a criação de advertências falso-positivas podem ser exemplificadas utilizando o exemplo do algoritmo ilustrado no Pseudocódigo 3.1. Neste exemplo, a possibilidade *p1* poderia ocorrer caso o comando de repetição detectado pelo algoritmo (linha 2) não executasse iterações. Nesta situação, apesar da lógica do algoritmo estar correta, o mesmo realiza uma suposição que não ocorre na prática (que o comando de repetição irá iterar repetidas vezes na execução do código). A possibilidade *p2* poderia ocorrer caso o comando de repetição detectado pelo algoritmo (linha 2) executasse apenas uma quantidade reduzida de iterações. Neste caso, ainda que a suposição do algoritmo esteja correta (isto é, que o comando de repetição irá de fato iterar quando o código for executado), o impacto prático proveniente da aplicação da diretriz pode ser bastante reduzido e não justificar uma alteração no código. Por fim, a possibilidade *p3* poderia ocorrer caso, no contexto específico da aplicação, não seja possível diminuir a frequência de execução dos comandos *commit*.

No exemplo da diretriz RED_COM, as possibilidades *p1*, *p2* e *p3* dependem do contexto da aplicação. Isto ocorre porque, na prática, a quantidade de iterações de um comando de repetição nem sempre pode ser verificada estaticamente (pois pode depender, por exemplo, do valor de um parâmetro de entrada ou valor acessado em um banco de dados) e a criticidade do código-fonte é algo mensurado pelos programadores (e, portanto, também não permite uma verificação estática automática). Nestes casos específicos, a abordagem proposta não é capaz de evitar que sejam criadas advertências falso-positivas em relação à diretriz RED_COM.

3.5.2 Estratégias para Diminuição de Advertências Falso-Positivas

Ainda que a representação do código-fonte usada pela abordagem apresentada não permita garantir ou provar a inexistência de advertências falso-positivas na saída reportada pela

mesma, é possível utilizar heurísticas para diminuir as chances da criação de advertências falso-positivas. Na prática, são realizadas checagens estáticas (que complementam o próprio algoritmo de checagem da diretriz) que visam diminuir a chance da advertência realizar uma suposição errada sobre a execução do código-fonte ou que a mesma acarrete em pouco impacto prático no procedimento de banco de dados. Exemplos destas heurísticas são discutidos no Capítulo 4 e Apêndice A.

Capítulo 4

Análise Automática de Procedimentos de Banco de Dados para Sugestão de Melhorias de Eficiência e Qualidade

Este capítulo apresenta estratégias de análise estática no intuito de detectar partes do código-fonte de procedimentos de banco de dados que não seguem diretrizes pré-definidas de eficiência e qualidade. As análises são apresentadas como fórmulas na notação da lógica de predicados e, em seguida, cada termo da fórmula é descrito separadamente. Neste capítulo, são também discutidas as heurísticas utilizadas no intuito de diminuir as chances de serem criadas advertências falso-positivas após a execução das análises estáticas definidas. Parte das diretrizes de eficiência e qualidade investigadas neste trabalho são também apresentadas no Apêndice A. No total, são apresentadas 23 diretrizes no Apêndice A.

Nas Seções 4.1 e 4.2 são discutidas as análises automáticas para a detecção de partes do código-fonte de procedimentos de banco de dados nas quais podem ser aplicadas diretrizes de eficiência e qualidade, respectivamente. Na Seção 4.3 são discutidas com mais detalhes as etapas necessárias para realizar o processamento do Resumo da Análise, o qual é gerado após a execução das análises descritas neste capítulo. No Capítulo 6, é apresentada uma discussão sobre os trabalhos disponíveis na literatura relacionada dos quais foram extraídas as diretrizes investigadas neste trabalho.

4.1 Análise de Eficiência de Procedimentos de Banco de Dados

Nesta seção são discutidas estratégias para a criação de análises automáticas para a detecção de partes de procedimentos de banco de dados em que podem ser aplicadas diretrizes pré-definidas de eficiência. Para cada análise, são apresentados: i) o acrônimo, nome e descrição da diretriz de eficiência; ii) a fórmula lógica que representa as condições analisadas estaticamente; e iii) descrição e discussão da estratégia para implementação das análises estáticas e das heurísticas utilizadas para diminuir as chances de que advertências falso-positivas sejam criadas após a execução da análise. Neste capítulo, são utilizadas as seguintes notações para apresentação dos termos das formulas lógicas apresentadas:

- *[term_1]*: os termos que aparecem entre colchetes representam condições necessárias para que a detecção da diretriz seja reportada no Resumo da Análise e cuja implementação é decidível de forma estática, ou seja, os dados necessários para a análise não dependem de configurações do usuário e podem ser acessados estaticamente no código-fonte.

- <*term_2*>: os termos que aparecem entre os símbolos $<$ e $>$ representam checagens adicionais que são realizadas no intuito de diminuir as chances de serem criadas advertências falso-positivas relacionadas à análise do procedimento de banco de dados. Além disso, em geral, a implementação da verificação necessária para estes termos é indecidível de forma estática e é realizada utilizando heurísticas e/ou parâmetros configuráveis pelo usuário da abordagem.

4.1.1 Tipos de Dados Nativos

Algumas linguagens procedimentais de banco de dados possuem tipos de dados específicos que são usados para realizar operações aritméticas de maneira mais eficiente [9] [12] [13] [27]. Por exemplo, a linguagem PL/SQL disponibiliza os tipos de dados nativos PLS_INTEGER, BINARY_INTEGER e BINARY_DOUBLE.

A avaliação de operações aritméticas utilizando tipos de dados nativos são otimizadas pois são executadas usando operações de baixo nível. É também comum que tipos de dados

nativos dispensem a realização de chamadas à bibliotecas para realizar operações aritméticas específicas ou para realizar checagens em tempo de execução do código, tornando-os ainda mais eficientes [12]. Outra característica destes tipos de dados é que os mesmos usualmente necessitam de um número menor de bits para sua representação interna, proporcionando assim, uma economia na memória utilizada pelos procedimentos de banco de dados. Em contrapartida, tipos de dados nativos permitem o armazenamento de valores dentro de uma faixa inferior às disponíveis pelos tipos de dados de alto nível equivalentes. Sendo assim, a utilização segura de tipos de dados nativos depende dos valores manipulados pelo procedimento de banco de dados.

Tipos de Dados Inteiros Nativos

Uma possível diretriz de eficiência é a utilização de tipos de dados inteiros nativos em variáveis e parâmetros utilizados para a realização de operações aritméticas. Esta diretriz é referenciada utilizando o Código de Diretriz (C_D) TI_NAT. As condições necessárias para a checagem estática da diretriz TI_NAT são mostradas na Fórmula Lógica 1.

- **Fórmula Lógica 1**: [*usage_of_integer_data_type(identifier)*] $\wedge$ [*used_for_arithmetic_operation(identifier)*] $\wedge$ <*executed_at_least_n_times(arithmetic_operation)*> $\wedge \neg$ <*overflow(identifier)*>

As condições dispostas na Fórmula Lógica 1 são detalhadas a seguir:

- *usage_of_integer_data_type(identifier)*: esta condição visa verificar se a o identificador (de variável ou parâmetro) analisado utiliza o tipos de dados INTEGER (ou algum tipo de dados inteiro equivalente que não possua uma representação interna em baixo nível);

- *used_for_arithmetic_operation(identifier)*: esta condição visa checar se identificador (de variável ou parâmetro) analisado é utilizado como fator no cálculo de alguma operação aritmética;

- *executed_at_least_n_times(arithmetic_operation)*: esta condição tem como objetivo realizar checagens estáticas para verificar se a operação aritmética detectada na condição *used_for_arithmetic_operation(identifier)* é executada pelo menos *n* (parâmetro

configurado pelo usuário) vezes pelo procedimento de banco de dados. Esta checagem é importante pois é necessário que a operação aritmética em questão seja executada repetidas vezes pelo procedimento para que a utilização de um tipo de dados nativo acarrete em uma otimização significativa no tempo de execução do procedimento.

Na prática, nem sempre é possível realizar a checagem desta condição de forma estática. A execução da operação aritmética em questão pode ocorrer dentro do escopo de um comando de repetição cuja quantidade de execuções é definida pelo (ou é dependente do) valor de um parâmetro de entrada do procedimento ou de um valor acessado em um banco de dados. Dessa forma, uma vez que a condição *executed_at_least_n_times(arithmetic_operation)* é indecidível de forma estática, é possível realizar esta checagem utilizando as seguintes heurísticas: TI_NAT.hi) verificar estaticamente se o comando de repetição que domina estritamente a operação aritmética define um número fixo de repetições (exs.: *(while i < 10)* ou *(while j < constant_value)*). Em caso afirmativo, deve ser verificado se a quantidade de repetições excede o parâmetro *n* definido pelo usuário; TI_NAT.hii) uma versão simplificada da heurística *TI_NAT.hi)* consiste em simplesmente verificar se a operação aritmética é dominada estritamente por pelo menos um comando de repetição; e TI_NAT.hiii) realizar uma checagem interprocedural entre as chamadas dos procedimentos a outros procedimentos no intuito de verificar se o procedimento que executa a operação aritmética detectada na condição *used_for_arithmetic_operation(identifier)* é chamado dentro do escopo de pelo menos um comando de repetição de outro procedimento. Em caso afirmativo, é ainda possível utilizar a heurística TI_NAT.hi para minimizar a chance da criação de uma advertência falso-positiva;

- *overflow(identifier)*: Esta condição visa verificar se, caso o resultado de alguma operação aritmética seja atribuída a uma variável ou parâmetro (do tipo valor/resultado ou resultado) do tipo de dados INTEGER, a mudança do tipo de dados desta variável ou parâmetro para um tipo de dados inteiro nativo pode causar um erro do tipo *overflow* em tempo de execução. Esta condição é indecidível de ser analisada de forma estática, uma vez que a operação aritmética pode, por exemplo, envolver como fator algum parâmetro de entrada do procedimento ou um valor acessado em um banco de

dados. Sendo assim, é possível realizar esta checagem utilizando a seguinte heurística: TI_NAT.hiv) nos casos em que são feitas atribuições de um valor fixo em variáveis ou parâmetros do tipo de dados INTEGER (exemplos: *k := 100.000*, *k := 50.000 + 50.000* e *k := constant_value*), verificar estaticamente se o valor atribuído irá causar um erro do tipo *overflow* na faixa de valores disponíveis para um tipo de dados inteiro nativo.

4.1.2 Redução de Trocas de Contexto

Em algumas linguagens de programação de banco de dados (T-SQL, PL/SQL e PL/pgSQL), as partes procedimentais dos procedimentos e os comandos SQL neles contidos são executados e interpretados por mecanismos distintos no SGBD. Desse modo, cada vez que, seguindo o fluxo do código, é preciso mudar de mecanismos, seja do mecanismo SQL para o Procedimental ou do Procedimental para o SQL, uma espera é gerada na execução no procedimento. Desse modo, quando excessivas esperas desta natureza são necessárias para a execução do procedimento, pode acontecer uma degradação no desempenho do procedimento. Este problema é denominado na literatura como trocas de contexto [5] [9] [12] [13] [27].

Uma possível diretriz de eficiência é a minimização do número de trocas de contexto necessárias para a execução do código. Esta diretriz é referenciada utilizando o código de diretriz TRC_CTX. As condições necessárias para a checagem estática da diretriz TRC_CTX são mostradas na Fórmula Lógica 2.

- **Fórmula Lógica 2**: <*excessive_context_switches(statement)*]>

As condições dispostas na Fórmula Lógica 2 são detalhadas a seguir:

- *excessive_context_switches(statement)*: esta condição visa identificar o comando analisado pode acarretar em excessivas trocas de contexto durante a execução do procedimento. Na prática, trocas de contexto excessivas ocorrem no código de procedimentos de banco de dados quando comandos SQL são executados dentro do escopo de comandos de repetição. Para a implementação desta checagem de forma estática, é possível utilizar a estrutura hierárquica da ACD que representa o procedimento para verificar a ocorrência de comandos SQL que são dominados estritamente por algum comando

de repetição. Além disso, é possível utilizar alguma das heurísticas discutidas na explanação da Fórmula Lógica 1 para assegurar-se de que a quantidade de iterações do comando de repetição é suficientemente grande para causar esperas perceptíveis devido a execução de trocas de contexto. Quando estas situações são verificadas no código, é possível utilizar duas estratégias para implementar a redução do número de trocas de contexto.

A primeira estratégia consiste em utilizar operadores que permitem ao procedimento acessar dados em bancos de dados e carregar parte destes dados na memória em uma única troca de contexto, evitando assim, que os dados sejam acessados no banco de dados de forma sequencial e que seja necessária a execução de uma troca de contexto para cada acesso realizado. Por exemplo, a linguagem PL/SQL disponibiliza os operadores BULK COLLECT e BULK INSERT [5] [9] [12] [13] [27] que acarretam na redução de trocas de contexto. A linguagem Transact-SQL também disponibiliza o operador BULK INSERT [15] que acarreta na redução de trocas de contexto. Similarmente, a linguagem PL/pgSQL disponibiliza o operador COPY [1] para este fim.

A segunda estratégia consiste em substituir, quando possível, partes procedimentais do código por comandos SQL equivalentes; isto porque, uma vez que um comando SQL é executado uma única vez pelo mecanismo de interpretação SQL do SGBD, uma única troca de contexto é necessária para a execução do mesmo.

4.1.3 Eliminação de Comando *Commit* em Iterações

É comum que na implementação de procedimentos de banco de dados sejam executados comandos *commit* para que as alterações geradas por comandos DML no SGBD sejam de fato refletidas no mecanismo de armazenamento do banco de dados. No entanto, a execução de comandos *commit* pode gerar esperas demasiadas [9] [12] [13] [27] [5], uma vez que este comando acarreta em uma série de operações a serem executados pelo SGBD, dentre elas, a escrita de dados em um disco rígido (o que representa uma operação bastante lenta). Sendo assim, uma possível diretriz de eficiência é a redução da quantidade de execuções de comandos *commit*. Por exemplo, ao invés de executar um comando *commit* em cada iteração de um comando repetição, executar um único comando *commit* a cada n iterações do comando

de repetição ou realizar uma execução única do comando *commit* após a finalização das iterações do comando de repetição. Esta diretriz é referenciada utilizando o C_D RED_COM. As condições necessárias para a checagem estática da diretriz RED_COM são mostradas na Fórmula Lógica 3.

- **Fórmula Lógica 3**: [*is_commit(statement)*] $\wedge$ <*executed_at_least_n_times(statement)*> $\wedge \neg$ <*is_critical_commit(statement)*>

As condições dispostas na Fórmula Lógica 3 são detalhadas a seguir:

- *is_commit(statement)*: esta condição visa checar se o comando analisado é do tipo *commit*;
- *executed_at_least_n_times(statement)*: esta condição visa checar se o comando *commit* identificado na condição *is_commit(statement)* é executado pelo menos n vezes por algum comando de repetição presente no procedimento. Para tal, podem ser utilizadas as heurísticas apresentadas na explanação da Fórmula Lógica 1;
- *is_critical_commit(statement)*: na prática, nem sempre é possível reduzir a frequência de execução de comandos *commit* pelo procedimento de banco de dados. Há situações em que é importante para o contexto da aplicação que as alterações realizadas por comandos DML sejam imediatamente refletidas no banco de dados. Como esta verificação é indecidível de forma estática, é sempre aconselhável que a redução na frequência de execução de comandos *commit* leve em consideração o contexto da aplicação e sua viabilidade deve ser discutida entre a equipe de desenvolvimento.

4.1.4 Utilização de Cursores Implícitos

É bastante comum que linguagens de programação de banco de dados (ex.: Transact-SQL, PL/SQL e PL/pgSQL) utilizem cursores para permitir a iteração e manipulação de dados armazenados em bancos de dados. Existem dois tipos de cursores: explícitos e implícitos. Ainda que estes dois tipos sejam equivalentes, cursores implícitos permitem que operações equivalentes às realizadas por cursores explícitos sejam implementadas utilizando menos comandos; isto porque, cursores implícitos não precisam ser declarados, nem requerem a

realização de operações do tipo: abertura do cursor, verificação do fim dos dados apontados pelo cursor e fechamento do cursor. Sendo assim, uma vez que linguagens de programação de banco de dados são usualmente interpretadas [9] [12], cursores implícitos tendem a ser mais eficientes, pois a utilização destes tipos de cursores implica na interpretação de menos comandos.

Neste contexto, uma possível diretriz de eficiência para a implementação de procedimentos de banco de dados é a preferência na utilização de cursores implícitos a cursores explícitos. Esta diretriz é referenciada utilizando o código de diretriz CUR_IMP. As condições necessárias para a checagem estática da diretriz CUR_IMP são mostradas na Fórmula Lógica 4.

- **Fórmula Lógica 4**: [*is_explicit_cursor(statement)*]

As condições dispostas na Fórmula Lógica 4 são detalhadas a seguir:

- *is_explicit_cursor(statement)*: esta condição visa checar se o comando analisado representa um cursor explícito no procedimento de banco de dados. Uma vez que todo cursor declarado é considerado do tipo explícito, devem ser checadas a declaração de cursores nos blocos de declaração de variáveis e nos tipos de dados dos parâmetros presentes na assinatura do procedimento.

4.2 Análise de Qualidade de Código em Procedimentos de Banco de Dados

Nesta seção, são apresentadas as diretrizes de qualidade de código para procedimentos de banco de dados investigadas neste trabalho.

4.2.1 Declaração de Variável Contador

Linguagens de programação de banco de dados permitem a criação de uma modalidade de comando de repetição bastante simples, a qual consiste em iterar sequencialmente em um intervalo de valores inteiros. Em algumas linguagens (ex. PL/SQL e PL/pgSQL), a variável

utilizada para contar a quantidade de iterações do laço (variável contador) é declarada implicitamente pelo programa. Nesse contexto, a declaração de uma variável ou definição de um parâmetro com o mesmo identificador utilizado por uma variável contador é considerada uma má pratica por três motivos: i) torna-se mais difícil entender o programa, uma vez que o valor do identificador irá variar de acordo com o trecho do programa em que o mesmo é utilizado; ii) um programador pode, erroneamente, tentar modificar o valor do identificador antes da execução do comando de repetição mas, na prática, o valor modificado não terá efeito sobre a quantidade de iterações do laço; e iii) a declaração explícita de uma variável ou parâmetro com o mesmo identificador de uma variável contador com o objetivo exclusivo de (erroneamente) permitir que a variável seja utilizada pelo laço irá acarretar, na prática, em um desperdício de memória pelo procedimento.

Desse modo, uma possível diretriz de qualidade de código consiste em não declarar variáveis ou parâmetros com o mesmo identificador utilizado por variáveis contadoras em laços. Esta diretriz é referenciada utilizando o C_D DEC_CTD. As condições necessárias para a checagem estática da diretriz DEC_CTD são mostradas na Fórmula Lógica 5.

- **Fórmula Lógica 5**: [*used_for_loop_count(identifier)*]

As condições dispostas na Fórmula Lógica 5 são detalhadas a seguir:

- *used_for_loop_count(identifier)*: esta condição visa verificar se o nome do identificador (de variável ou parâmetro) analisado é também utilizado por identificadores de variáveis contadores em comandos de repetição do procedimento de banco de dados.

4.2.2 Vínculo de Tipos de Dados

É bastante comum que na implementação de procedimentos de banco de dados sejam declaradas variáveis ou utilizados parâmetros (do tipo resultado ou valor/resultado) para o armazenamento de valores resgatados de bancos de dados. Neste contexto, caso estas variáveis ou parâmetros definam seus tipos de dados de forma estática (ex.: *pi real*, *full_name varchar(200)*), os seguintes efeitos indesejados podem ocorrer caso sejam realizadas mudanças nos bancos de dados que forneçam dados para o procedimento:

- Erro de truncamento ou do tipo *overflow*: podem ocorrer caso os tipos de dados de campos de tabelas ou de atributos de objetos armazenados nos bancos de dados aumentem de tamanho;

- Desperdício de memória: pode ocorrer caso os tipos de dados de campos de tabelas ou de atributos de objetos armazenados nos banco de dados diminuam de tamanho;

- Erro do tipo *underflow*: pode ocorrer caso os tipos de dados de campos de tabelas ou de atributos de objetos armazenados nos banco de dados recebam alterações referentes à precisão.

Para evitar a ocorrência destes efeitos decorrentes de tarefas de manutenção em bancos de dados, as seguintes estratégias podem ser utilizadas:

- Utilizar os operadores *%TYPE* ou *%ROWTYPE* para realizar o vínculo automático do tipo de dados do campo de uma tabela ao tipo de dados de uma variável (ou parâmetro) do procedimento ou dos tipos de dados de todos os campos de uma tabela ao tipo de dados de uma variável (ou parâmetro) do procedimento, respectivamente. Estes operadores são disponibilizados por algumas linguagens procedimentais de banco de dados (ex.: PL/SQL e PL/pgSQL) e permitem que mudanças no tipo de dados de campos de tabelas do banco de dados sejam automaticamente refletidas no tipo de dados utilizado por variáveis ou parâmetros nos procedimentos de banco de dados;

- Alguns SGBD's (ex.: Oracle, SQL Server, PostgreSQL dentre outros) permitem que o usuário defina tipos de dados personalizados. Esta funcionalidade pode também ser utilizada para resolver os problemas discutidos nesta seção. Para tal, é necessário a criação de um tipo de dados personalizado e utilizá-lo para definir tanto tipos de dados utilizados pelo banco de dados quanto pelas variáveis ou parâmetros dos procedimentos de banco de dados. Desse modo, alterações feitas no tipo de dados personalizado terão efeito tanto no banco de dados quanto nos procedimentos de banco de dados.

Neste contexto, uma possível diretriz de qualidade de código é a realização do vínculo automático dos tipos de dados de variáveis e parâmetros utilizados para armazenar valores acessados em bancos de dados pelos procedimentos [9] [12]. Esta diretriz é referenciada

utilizando o C_D VINC_TIP_DAD. As condições necessárias para a checagem estática da diretriz VINC_TIP_DAD são mostradas na Fórmula Lógica 6.

- **Fórmula Lógica 6**: <*used_for_storage_of_db_value(identifier)*> $\wedge$ $\neg$ ([*linked_to_database(identifier)*] $\wedge$ <*linked_to_correct_field(identifier)*>)

As condições dispostas na Fórmula Lógica 6 são detalhadas a seguir:

- *used_for_storage_of_db_value(identifier)*: esta condição tem como objetivo verificar se o identificador (de variável ou parâmetro) analisado é utilizado para armazenar valores acessados em bancos de dados. Na prática, implementar esta análise representa um custo bastante elevado, uma vez que o armazenamento do valor acessado no banco de dados pode ser atribuído ao identificador de forma indireta, por exemplo: *(a:= db_value; statement_list; b := a;)*. Desse modo, para verificar atribuições indiretas de valores provenientes de banco de dados é necessário realizar uma análise no fluxo de dados do programa e, para tal, é requerida a criação de uma estrutura de dados denominada *Grafo de Fluxo de Dados* (GFD) [30]. No entanto, é possível utilizar uma heurística, que apresenta um custo mais baixo do que o associado à análise de um GFD, para dar indícios de que o identificador analisado é utilizado para armazenar valores provenientes de banco de dados. A heurística consiste na seguinte verificação: VINC_TIP_DAD.h1) checar se a variável ou parâmetro é utilizado em algum comando de atribuição de valores provenientes de bancos de dados, tais como: *SELECT INTO* ou *FETCH INTO*. Estes comandos são disponíveis por linguagens como PL/SQL, T-SQL e PL/pgSQL;

- *linked_to_database(identifier)*: esta condição visa checar se o tipo de dados do identificador analisado encontra-se automaticamente vinculado a algum campo de tabela ou conjunto de campos de uma tabela de um banco de dados;

- *linked_to_correct_field(identifier)*: esta condição visa checar se o vínculo do tipo de dados utilizado pelo identificador referencia o campo (ou lista de campos) correto no banco de dados. Para tal, deve ser verificado se o tipo de dados do campo (ou lista de campos) da tabela do banco de dados utilizado para atribuir valores ao identificador é igual ao tipo de dados vinculado ao identificador. Este vínculo pode ser realizado com

a utilização do operador *%TYPE* (ou *%ROWTYPE*) ou a partir de um tipo de dados personalizado pelo usuário. Para a implementação desta análise, é necessário que o usuário da abordagem forneça acesso aos metadados dos bancos de dados utilizados pelo procedimento de banco de dados analisado.

4.2.3 Manipulação de Cursores

É comum que linguagens de programação de banco de dados disponibilizem tipos de dados denominados cursores para permitir a iteração e manipulação de dados armazenados em banco de dados. Em geral, cursores permitem a realização de operações como: resgate do próximo elemento, verificação da existência de próximo elemento, verificação do estado do cursor (aberto ou fechado), abertura do cursor, encerramento do cursor, apontamento do cursor para a posição inicial, dentre outras. Em algumas linguagens (ex.: T-SQL, PL/SQL, PL/pgSQL), um cursor pode ser aberto e fechado diversas vezes. Nesse caso, uma boa prática é verificar se o mesmo está aberto antes de tentar abri-lo, uma vez que a tentativa de abertura de um cursor já aberto causa um erro de execução. Esta diretriz é referenciada utilizando o C_D ABT_CUR_EXP. As condições necessárias para a checagem estática da diretriz ABT_CUR_EXP são mostradas na Fórmula Lógica 7.

- **Fórmula Lógica 7**: [*is_explicit_cursor(identifier)*] $\wedge$ [*is_opened_by_procedure(identifier)*] $\wedge \neg$ [*status_verified(identifier)*]

As condições dispostas na Fórmula Lógica 7 são detalhadas a seguir:

- *is_explicit_cursor(identifier)*: esta condição visa checar se o identificador analisado representa um cursor explícito utilizado pelo procedimento de banco de dados;
- *is_opened_by_procedure(identifier)*: esta condição visa checar se o procedimento de banco de dados realiza algum comando de abertura no cursor explícito detectado na condição *is_explicit_cursor(identifier)*;
- *status_verified(identifier)*: esta condição visa checar se o procedimento de banco de dados analisado realiza algum teste para verificar o estado do cursor explícito detectado na condição *is_explicit_cursor(identifier)* antes de executar algum comando para realizar a abertura do mesmo.

4.2.4 Retorno de Valores *Boolean* em Funções

Uma boa prática no desenvolvimento de procedimentos (do tipo função) de banco de dados que retornam valores do tipo BOOLEAN é sempre retornar apenas as opções TRUE ou FALSE [9]. Isto porque, o retorno do valor NULL em funções de retorno BOOLEAN pode trazer as seguintes dificuldades para o desenvolvimento de procedimentos de banco de dados: i) ao utilizar o retorno da função, é necessário adicionar checagens adicionais para a determinação do fluxo a ser seguido pelo código nos casos em que o retorno da função apresentar o valor NULL; ii) o valor NULL não apresenta uma semântica definida (pode, por exemplo, representar um valor "desconhecido"ou "não aplicável"), sendo assim, é difícil definir o comportamento correto para a interpretação de um valor de retorno NULL; e iii) em alguns casos, não faz sentido que a função retorne o valor NULL. Por exemplo, a avaliação da função *is_eligible_employee(empolyee_id int)*, a qual deve checar se um empregado é elegível ou não para aposentar-se, deve sempre permitir ou não a aposentadoria do funcionário (uma vez que trata-se de uma checagem com regras bem definidas e sempre decidível). Neste contexto, a diretriz de qualidade de código que define o retorno de valores não nulos em procedimentos (do tipo função) de banco de dados com retorno do tipo BOOLEAN é referenciada utilizando o C_D NULL_RET. As condições necessárias para a checagem estática da diretriz NULL_RET são mostradas na Fórmula Lógica 8.

- **Fórmula Lógica 8**: [*is_function(procedure)*] $\wedge$ [*return_boolean(procedure)*] $\wedge$ <*is_returned(null)*>

As condições dispostas na Fórmula Lógica 8 são detalhadas a seguir:

- *is_function(procedure)*: esta condição visa checar se o procedimento de banco de dados analisado é do tipo função;
- *return_boolean(procedure)*: esta condição objetiva verificar se o tipo de dados do retorno do procedimento de banco de dados é do tipo BOOLEAN;
- *is_returned(null)*: esta condição tem como objetivo verificar se é executado algum retorno de valor NULL pelo procedimento de banco de dados. Esta verificação é indecidível de forma estática, uma vez que o valor retornado pelo procedimento pode,

por exemplo, ser proveniente de um parâmetro de entrada do procedimento ou de um acesso a banco de dados. Desse modo, podem ser verificados os comandos de retorno executados pelo procedimento que utilizam o literal NULL ou constantes declaradas com o valor NULL.

4.3 Processamento do Resumo da Análise

Nesta seção, são apresentados exemplos dos conceitos definidos no Capítulo 3 que são importantes para a tarefa de processamento do Resumo da Análise gerado pela abordagem proposta.

4.3.1 Classificação das Diretrizes

Nesta seção, é apresentada uma classificação das diretrizes de eficiência e qualidade apresentadas neste capítulo e no Apêndice A em subgrupos. Na prática, esta classificação é utilizada para dois objetivos: i) permitir a ordenação das advertências reportadas pela abordagem de acordo com os subgrupos de advertências; e ii) permitir que um conjunto de diretrizes sejam referenciadas de acordo com seus respectivos subgrupos, nas discussões dos resultados obtidos com a avaliação da abordagem proposta (Capítulo 5). As diretrizes de eficiência investigadas neste trabalho podem ser agrupadadas nos seguintes subgrupos:

- **Otimização de tipos de dados**.: {TI_NAT, TN_NAT, TD_RES, OTI_TDA};
- **Otimização de operações (não SQL) para manipulação de dados em banco de dados**.: {TRC_CTX, RED_COM, CUR_IMP};
- **Mudanças na ordem de avaliação de expressões**.: {PRI_COND, ORD_LOG};
- **Redução de cópia de parâmetros**.: {COP_PAR};
- **Utilização de funções nativas**.: {FUN_NAT};
- **Remoção de identificadores não utilizados**.: {REM_PAR, REM_VAR, DEC_REP}.

As diretrizes de qualidade investigadas neste trabalho podem ser agrupadas nos seguintes subgrupos:

- **Simplificação do Fluxo de Controle**: {EXEC_GOTO, EXEC_SCP, MULT_RET, RET_LOOP}
- **Não utilização de código escorregadio**: {DEC_CTD, ELSE_CASE, NOT_NMD, NULL_RET, ABT_CUR_EXP, VINC_TIP_DAD}
- **Não utilização de efeitos colaterais**: {EFT_COL_FUN}
- **Melhoria do significado dos identificadores**: {REF_LIT, ENC_LOG}
- **Utilização de estilos de programação**: {PAR_CONV, DISP_PAR, DISP_VAR, PROC_TO_FUNC}

4.3.2 Pontuação das Diretrizes

Nesta seção, é apresentado um exemplo de atribuição de pontuação às diretrizes de eficiência e qualidade investigadas neste trabalho. Como discutido no Capítulo 3, as relações entre a pontuação das diretrizes visam resolver os conflitos de advertências entre as mesmas partes do código-fonte dos procedimentos de maneira que esforço necessário pelos desenvolvedores para aplicar as advertências seja minimizado e ao mesmo tempo determinar uma advertência única e mais adequada para conjunto de problemas que podem estar relacionados a uma mesma parte do código-fonte. Considerando as diretrizes apresentadas neste capítulo e no Apêndice A, as seguintes relações entre a pontuação das diretrizes foi utilizada na avaliação (Capítulo 5) descrita neste trabalho:

- $score$(REM_VAR) = $score$(REM_PAR)
- $score$(REM_PAR) > $score$(DEC_REP) > $score$(EFT_COL_FUN) > $score$(VINC_TIP_DAD)
- $score$(VINC_TIP_DAD) > $score$(OTI_TDA) = $score$(TI_NAT) = $score$(TN_NAT) = $score$(COP_PAR) = $score$(CUR_IMP) = $score$(DEC_CTD)
- $score$(DEC_CTD) > $score$(TD_RES)
- $score$(CUR_IMP) > $score$(ABT_CUR_EXP)

- *score*(MULT_RET) = *score*(NULL_RET) > *score*(RET_LOOP)

Além destas relações entre as pontuações, deve ser a utilizada a seguinte regra para a resolução de conflitos: sejam D1 e D2 duas diretrizes distintas, caso não exista uma relação explícita entre as pontuações das mesmas, este fato implica que *score*(D1) = *score*(D2).

Utilizando as relações entre as pontuações de diretrizes definidas, é possível apresentar exemplos práticos de como conflitos entre advertências para um procedimento de banco de dados podem ser resolvidos. Por exemplo, é preferível que um identificador (de variável ou parâmetro) não utilizado seja removido do procedimento do que ser renomeado por utilizar o mesmo identificador de uma variável contador (regra definida pela relação *score*(REM_VAR) > *score*(DEC_CTD)). Outro exemplo consiste em atribuir uma prioridade à diretriz para transformação de um cursor explícito em um cursor implícito maior do que pontuação da diretriz para verificação do status de um cursor explícito (regra definida pela relação *score*(CUR_IMP) > *score*(ABT_CUR_EXP).

4.3.3 Tabela de Mapeamento de Advertências

Nesta seção, é apresentado um exemplo de tabela de mapeamento de advertências de melhorias para procedimentos de banco de dados. Na Tabela 4.1, são apresentadas mensagens de advertências nas quais parte das diretrizes de eficiência investigadas neste trabalho podem ser mapeadas.

Diretriz	**Mensagem de Advertência**
REM_VAR	Variável <*identifier*(stmt)> é declarada mas não utilizada - *line*(stmt)
DEC_CTD	Considere renomear o identificador <*identifier*(stmt)> - *line*(stmt);
RED_COM	Considere reduzir as execuções do comando *commit* - *line*(stmt);
CUR_IMP	Considere transformar <*identifier*(stmt)> em um cursor implícito - *line*(stmt);

Tabela 4.1: Exemplo de tabela de mapeamento de advertências para as diretrizes de eficiência: REM_VAR, DEC_CTD, RED_COM e CUR_IMP.

Desse modo, utilizando as entradas da Tabela 4.1 e as relações entre as pontuações das diretrizes definidas na Seção 4.3.2 é possível realizar o mapeamento entre um con-

junto de tuplas do tipo R_A<Tipo_Diretriz, Código_Diretriz, Pontuação_Diretriz, Identificador_Comando, Linha_do_Código_Fonte> retornadas pelos módulos *Efficiency Analyzer* e *Quality Analyzer* em mensagens de advertências de melhorias para o código-fonte de um procedimento de banco de dados. Por exemplo, o conjunto de tuplas:

- <Eff, DEC_CTD, *score*(DEC_CTD), C2, 2> ∪ <Eff, REM_VAR, *score*(REM_VAR), C1, 5> ∪ <Eff, CUR_IMP, *score*(CUR_IMP), C1, 5> ∪ <Eff, RED_COM, *score*(RED_COM), C3, 8>

Pode ser mapeado para as seguintes advertências de melhorias:

- Advertência 1: Considere renomear o identificador <*identifier*(C2)> - linha de código-fonte 2;
- Advertência 2: A variável <*identifier*(C1)> é declarada mas não utilizada - linha de código-fonte 5;
- Advertência 3: Considere reduzir as execuções do comando *commit* - linha de código-fonte 8;

Neste caso, nota-se que a relação (*score*(REM_PAR) $>$ *score*(CUR_IMP) eliminou a necessidade da criação de uma advertência para uma das tuplas (<Eff, CUR_IMP, *score*(CUR_IMP), C1, 5>) do conjunto processado.

Capítulo 5

Avaliação da Abordagem Proposta

Neste capítulo, são descritos o estudo de caso e os experimentos realizados para a avaliação da abordagem proposta (Capítulo 3) para análise automática do código-fonte de procedimentos de banco de dados. Primeiramente, é apresentada uma instanciação da abordagem proposta, a qual é apresentada na forma de uma ferramenta denominada PL/SQL Advisor (Seção 5.1). Em seguida, é discutido um experimento (Seção 5.2) conduzido para analisar o impacto da aplicação das diretrizes de eficiência (Capítulo 4 e Apêndice A) na execução de procedimentos de banco de dados. São comparados os resultados das execuções de procedimentos de banco de dados antes e depois da aplicação individual de diretrizes de eficiência.

Na Seção 5.3, é apresentado um estudo de caso realizado com procedimentos de banco de dados reais extraídos de um repositório de projetos online. Neste cenário, são avaliadas a eficiência (tempo de análise automática) e eficácia (quantidade de advertências reportadas) da abordagem proposta. Por último, são apresentados experimentos (Seção 5.4) realizados com procedimentos de banco de dados reais utilizados em um projeto industrial, com o objetivo de comparar a eficácia e eficiência da abordagem proposta com inspeções manuais realizadas por desenvolvedores.

5.1 Ferramenta PL/SQL Advisor

No intuito de avaliar a abordagem proposta (Capítulo 3), foi implementada uma instanciação da mesma na forma de uma ferramenta denominada PL/SQL Advisor[1], cujo objetivo é repor-

[1] Site da ferramenta PL/SQL Advisor: sites.google.com/site/plsqladvisor

tar advertências relacionadas à aplicação de diretrizes de eficiência e qualidade em procedimentos armazenados em banco de dados escritos na linguagem de programação PL/SQL, do SGBD Oracle. Esta linguagem foi escolhida por ser bastante popular entre desenvolvedores de procedimentos de banco de dados.

Na Seção 5.1.1 são discutidos detalhes da implementação da ferramenta e apresentada uma visão geral da mesma. Na Seção 5.1.2 é apresentada uma visão geral da interface gráfica da ferramenta. Finalmente, na Seção 5.1.3 é apresentado um exemplo de utilização da ferramenta.

5.1.1 Visão Geral

A arquitetura da ferramenta PL/SQL Advisor é mostrada na Figura 5.1. A Ferramenta possui quatro componentes (GUI, CORE, CONTROLLER e PERSISTENCE), cujos módulos realizam as principais funcionalidades da ferramenta.

O componente de GUI é responsável por permitir a interação do usuário com a ferramenta. Este módulo é responsável por i) exibir ao usuário (*File Display*) uma lista de arquivos com base em um diretório escolhido pelo usuário; ii) permitir a seleção dos arquivos SQL (*File Chooser*) que serão analisados pela ferramenta; e iii) mostrar ao usuário o resultado da execução da abordagem na forma de advertências de melhorias (*Result Display*).

O componente CORE, que abrange as principais funcionalidades da abordagem proposta, é responsável por i) realizar as análises léxica (*PL/SQL Lexer*) e sintática (*PL/SQL Parser*) dos arquivos de código-fonte escritos em PL/SQL e transformá-los em representações na forma de Árvores de Controle de Dependência (ACD); ii) realizar análises (*Efficiency Analyzer* e *Quality Analyzer*) sobre as representações na forma de ACD dos procedimentos no intuito de detectar partes do código-fonte nas quais podem ser aplicadas diretrizes pré-definidas de eficiência e qualidade; e iii) executar heurísticas (*Heuristic Executor*) para diminuir as chances de advertências falso positivas serem reportadas após a análise.

O módulo CONTROLLER é responsável por processar (*Summary Processor*) os Resumos da Análise retornado pelos módulos *Efficiency Analiyzer* e *Quality Analyzer* e mapeá-los para advertências de melhorias. Além disso, esse componente é responsável por formatar as advertências (*Result Formatter*) e repassar as mesmas para o módulo de exibição (*Warning Display*) e de persistência (*Output Persistence Manager*) das advertências. Por fim, o com-

ponente PERSISTENCE é responsável por formatar as advertências (*Output Formatter*) em uma representação que facilite futuros acessos pelo usuário e persistir em arquivos (*Output Persister*) as advertências reportadas pela ferramenta.

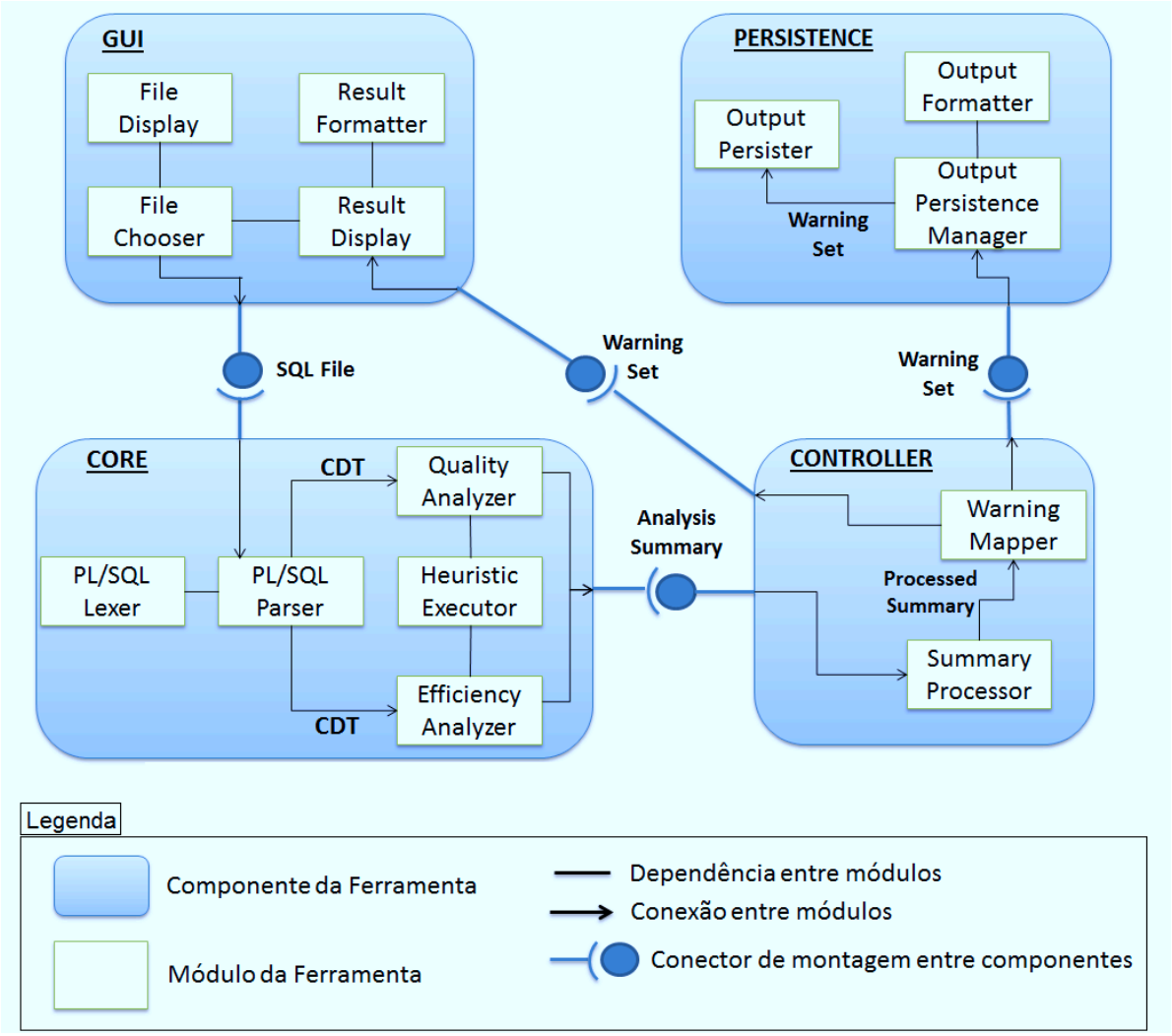

Figura 5.1: Arquitetura da ferramenta PL/SQL Advisor.

A ferramenta PL/SQL Advisor foi desenvolvida utilizando a linguagem de programação JAVA. Para a geração do código das análises léxica e sintática utilizou-se a ferramenta ANTLR [23], um gerador de analisadores sintáticos que utiliza a estratégia LL(*), ou seja, realiza a análise sintática de forma descendente e tenta deduzir as produções da gramática a partir do nó raiz. Incluindo o código gerado de forma automática, a ferramenta possui atualmente 44.976 linhas de código. A ferramenta PL/SQL Advisor é capaz de analisar todas as diretrizes de eficiência e qualidade de código discutidas no Capítulo 4 e no Apêndice A.

A ferramenta apresenta atualmente algumas limitações em relação à abordagem proposta (Capítulo 3) e à especificação dos algoritmos de análise estática propostos no Capítulo 4 e Apêndice A. As limitações são: não utilização de metadados armazenados em bancos de dados nas análises estáticas, utilização de um limiar fixo (igual a 1) para avaliar a quantidade iterações dos comandos de repetição e a impossibilidade de ordenar as advertências reportadas pela ferramenta. Um comparativo entre a ferramenta PL/SQL Advisor com outras ferramentas de propósitos semelhantes é apresentado no Capítulo 6.

5.1.2 Interface Gráfica

A interface gráfica da ferramenta PL/SQL Advisor possui três telas principais: Seleção de Arquivos, Seleção de Advertências e Exibição de Advertências. Inicialmente, o usuário da ferramenta deve selecionar um arquivo de código-fonte de um procedimento de banco de dados (com extensão .SQL) ou diretório do computador. Em seguida, a ferramenta irá exibir na tela de Exibição de Arquivos (Figura 5.2) o diretório (ou arquivo) selecionado na forma de uma árvore. Os arquivos e diretórios presentes no diretório selecionado são exibidos na forma de nodos filhos do diretório principal. Além disso, são carregados recursivamente os sub diretórios e respectivos arquivos neles contidos em qualquer nível da hierarquia.

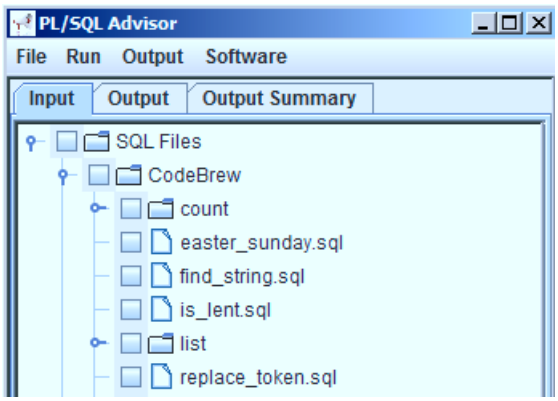

Figura 5.2: Tela de exibição de arquivos.

Após selecionar os arquivos a serem analisados pela ferramenta (exemplo apresentado na Figura 5.3), o usuário deve executar a análise automática na ferramenta (Figura 5.4). Este último passo acarreta na execução do fluxo da abordagem proposta no Capítulo 3. Neste

caso, o fluxo é instanciado para um processo de análise estática de procedimentos de banco de dados escritos na linguagem de programação PL/SQL.

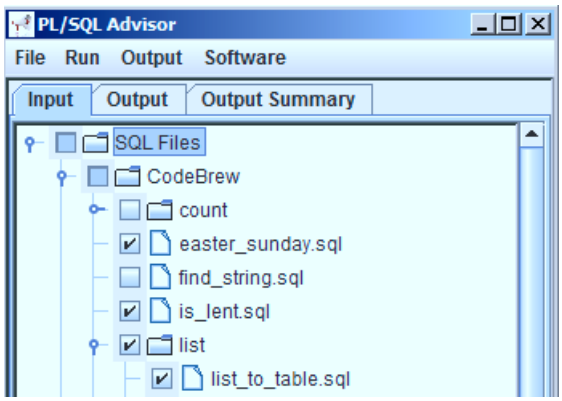

Figura 5.3: Exemplo de seleção de arquivos.

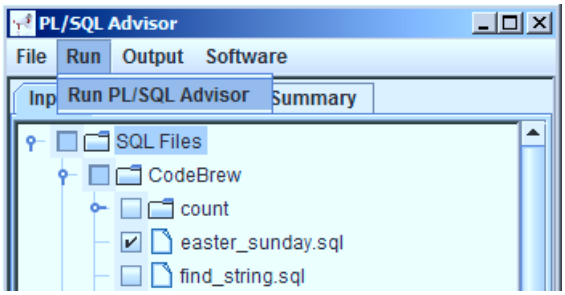

Figura 5.4: Procedimento para execução da ferramenta PL/SQL Advisor.

A tela de Seleção de Advertências (Figura 5.5) exibe uma lista contendo os arquivos que foram selecionados pelo usuário e, em seguida, analisados pela ferramenta. Quando um dos arquivos da lista é selecionado pelo usuário, a ferramenta mostra na tela de Exibição de Advertências (exemplo mostrado na Figura 5.6) as mensagens de advertências criadas para o código-fonte analisado.

5.1.3 Exemplo de Utilização da Ferramenta

A seguir, é apresentado um exemplo da utilização da ferramenta PL/SQL Advisor para analisar um código-fonte escrito na linguagem PL/SQL. O Código-fonte 5.1 tem como objetivo iterar sobre os empregados armazenados na tabela *employee_table* e verificar quais destes

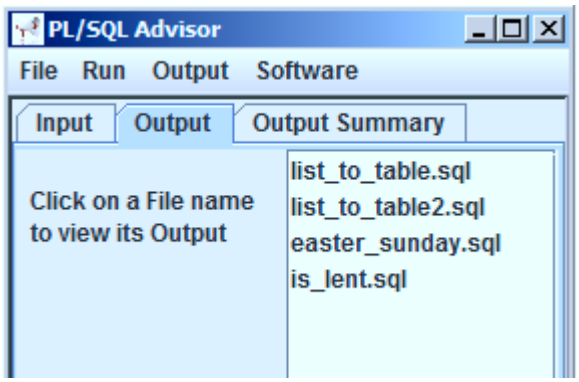

Figura 5.5: Tela de Seleção de Advertências.

empregados são elegíveis para aposentadoria. Para tal, são verificadas as condições *(employee_score(emp) > 5)* e (*emp.years_worked > 30*). As seguintes diretrizes de eficiência podem ser aplicadas no Código-fonte 5.1: TI_NAT, ORD_LOG, RED_COM, CUR_IMP, REM_PAR e REM_VAR. Além destas, podem ser aplicadas as seguintes diretrizes de qualidade: PAR_CONV e ABT_CUR_EXP.

A Figura 5.6 apresenta a saída da ferramenta PL/SQL Advisor após a análise do Código-fonte 5.1. Neste exemplo, a ferramenta necessitou de 686 milissegundos para realizar o processo de análise automática do procedimento em um computador com 4 GB de memória RAM e processador Core Intel i5 2410M.

```
1  CREATE PROCEDURE ELIGIBLE_EMPLOYEES(emp_limit IN INTEGER) AS
2      count_employees INTEGER := 0;
3      valid_employee BOOLEAN;
4      emp_rec employee_table\%ROWTYPE;
5      CURSOR emp_cur IS select id, name, years_worked from employee_table;
6   BEGIN
7      OPEN emp_cursor;
8      LOOP
9        FETCH emp_cur INTO emp_rec;
10       EXIT WHEN emp_cur\%NOTFOUND;
11     IF (employee_score(emp_rec.id) > 5 AND emp_rec.years_worked > 30) THEN
12           INSERT INTO ELIGIBLE_EMPLOYEES VALUES (emp_rec.id, emp_rec.name,
                 emp_rec.years_worked);
13           count := count + 1;
14           COMMIT;
```

```
15        END IF;
16      END LOOP;
17      DBMS_OUTPUT.PUT_LINE('# Eligible Employees:' || count_employees);
18    END ELIGIBLE_EMPLOYEES;
```

Código-fonte 1. Código do procedimento de banco de dados eligible_employees escrito na linguagem PL/SQL.

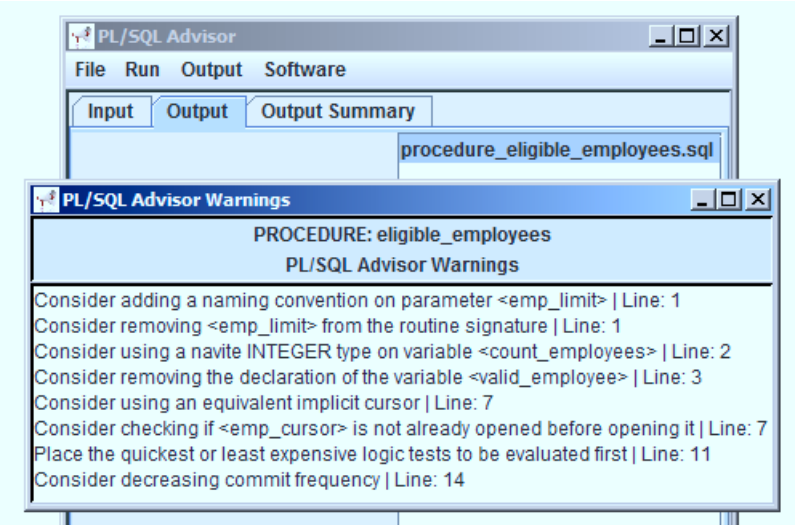

Figura 5.6: Advertências reportadas pela ferramenta PL/SQL Advisor após a análise automática do Código-fonte 5.1

5.2 Experimentos de Desempenho

Como parte da avaliação da abordagem proposta (Capítulo 3), foram realizados experimentos de desempenho com o intuito de medir os impactos relacionados com a aplicação das diretrizes de eficiência investigadas neste trabalho (Capítulo 4 e Apêndice A) na execução de procedimentos de banco de dados. Os resultados deste experimento são importantes para apresentar indícios sobre o impacto prático das referidas diretrizes de eficiência.

5.2.1 Planejamento da Investigação Experimental

O objetivo do experimento de desempenho é comparar o tempo de execução de procedimentos armazenados em banco de dados antes e depois da aplicação de melhorias individuais de eficiência. Para a criação dos procedimentos de banco de dados utilizados, foi escolhida a linguagem PL/SQL. Esta linguagem foi escolhida por ser bastante popular entre desenvolvedores de procedimentos de banco de dados e por ser a mesma linguagem escolhida para a instanciação (ferramenta PL/SQL Advisor) da abordagem proposta.

Nos experimentos, foi utilizado o contexto de um ambiente com as configurações padrões de memória global (SGA e PGA) do Sistema Gerenciador de Banco de Dados Oracle na versão 11g. Além disso, os testes foram realizados sobre o sistema operacional Windows 7 em um computador com a seguinte configuração: 4GB de memória RAM e processador Core Intel i5 2410M.

5.2.2 Seleção do Contexto

Tarefas associadas a acessos a bancos de dados usualmente representam o gargalo de desempenho das aplicações. A área de investigação de desempenho em aplicações de banco de dados é vasta e a solução de problemas relacionados a esta área é bastante importante para o mercado de software. Nesse contexto, é interessante que seja investigado o impacto de diretrizes de eficiência na execução de procedimentos de banco de dados, tal como realizado em trabalhos disponíveis na literatura [4] [5] [9] [12] [13].

O escopo deste experimento de desempenho limitou-se a um contexto específico. Foi analisado o impacto da aplicação de melhorias de eficiência em procedimentos de banco de dados escritos em uma linguagem específica (PL/SQL), executados em um um ambiente de execução fixo (Versão 11g do SGBD Oracle no sistema operacional Windows 7).

5.2.3 Variáveis

Neste experimento serão utilizadas três variáveis independentes e uma variável dependente. As variáveis independentes presentes no experimento são: a implementação do procedimento de banco de dados escrito em PL/SQL (Pr), a diretriz de eficiência analisada (Dt) e a quantidade de execuções do procedimento (Qe). A variável Pr representa a implementa-

ção do procedimento de banco de dados utilizada para medir o impacto de uma diretriz de desempenho específica. Para cada diretriz de eficiência investigada, esta variável pode assumir dois valores: P_n (implementação do procedimento antes da aplicação da diretriz) e P'_n (implementação do procedimento após a aplicação da diretriz).

A variável Qe representa a quantidade de execuções do procedimento de banco de dados. A existência desta variável no experimento é importante por dois motivos: i) é comum que o impacto de algumas diretrizes de eficiência seja perceptível apenas após a execução do procedimento múltiplas vezes (por exemplo, dentro de um comando de repetição); e ii) a mesma permite investigar a partir de quantas chamadas do procedimento o impacto da diretriz passa a ser significativo.

A variável dependente presente no experimento é o tempo de execução (Te) do procedimento de banco de dados (Pr) executado. Ainda que a aplicação de diretrizes de eficiência possa acarretar outros benefícios (por exemplo, no consumo de memória e/ou ciclos de CPU) além da diminuição do tempo de execução, o experimento limitou-se a investigar esta última variável por representar um fator mais crítico em aplicações de banco de dados.

Em resumo, cada tratamento do experimento irá comparar o tempo de execução (Te) de procedimentos de banco de dados (Pr) chamados repetidamente em um laço de n iterações (Qe), antes e depois da aplicação da diretriz de eficiência (Dt). A Tabela 5.1 apresenta detalhes das variáveis utilizadas no experimento de desempenho.

Variável	**Tipo**	**Escala**	**Níveis Utilizados**
Pr	Qualitativa	Nominal	P_n e P'_n
Qe	Quantitativa	Razão	$n/n \in N$
Dr	Qualitativa	Nominal	{TI_NAT, ORD_LOG, TRC_CTX}
Te	Quantitativa	Razão	$\{1..t\}/t \in R$

Tabela 5.1: Variáveis do experimento de desempenho.

5.2.4 Instrumentação

A medição da variável Te foi realizada utilizando a estratégia de temporizadores. Em cada bloco anônimo utilizado para executar os procedimentos do experimento em um comando

de repetição, foi utilizada uma variável denominada *current_time* para armazenar o valor da função nativa DBMS_UTILITY.get_time(). Após a execução do comando de repetição, o valor proveniente da subtração (DBMS_UTILITY.get_time() - *current_time*) representa o tempo de execução das repetidas chamadas ao procedimento analisado. O tempo de execução foi medido utilizando a precisão de milissegundos.

5.2.5 Elaboração das Unidades Experimentais

No experimento, o impacto de cada diretriz de eficiência foi medido separadamente. Para cada diretriz investigada, foram realizados os seguinte passos:

- Foi implementado um procedimento PL/SQL P_n que realiza um processamento ineficiente e permite uma otimização de acordo com a diretriz de eficiência D_j;
- A chamada do procedimento P_n foi colocada dentro de um de comando de repetição (cuja quantidade de iterações é definida pela variável Qe) de um bloco anônimo;
- Foi calculada a média de 50 execuções do bloco anônimo que realiza a chamada do procedimento P_n;
- O procedimento P_n foi reescrito em um procedimento P'_n aplicando a diretriz de eficiência D_j;
- Foi calculada a média de 50 execuções do bloco anônimo que realiza a chamada do procedimento P'_n;
- O tempo médio das execuções dos procedimentos P_n e P'_n foi plotado em gráficos de barra;

As seguintes diretrizes de eficiência foram investigadas no experimento: TI_NAT, ORD_LOG, TRC_CTX. Estas diretrizes foram escolhidas por representarem cada um dos tipos de diretrizes de eficiência investigados neste trabalho: utilização de tipos de dados nativos (TI_NAT), ordenação de expressões (ORD_LOG), otimização de operações de acesso a banco de dados (TRC_CTX).

Para a avaliação da diretriz TI_NAT, foi criado um procedimento que realiza repetidas chamadas a operações aritméticas envolvendo identificadores do tipo de dados INTEGER.

Em seguida, este procedimento foi reescrito para realizar as mesmas operações aritméticas utilizando identificadores do tipo de dados PLS_INTEGER (tipo de dados inteiro nativo).

Para a avaliação da diretriz ORD_LOG, foi criado um procedimento que realiza repetidas avaliações de uma expressão lógica composta por quatro expressões relacionais. As expressões relacionais foram dispostas em uma ordem em que, na maioria das iterações do laço do procedimento, as quatro expressões lógicas necessitassem ser avaliadas. Em seguida, o procedimento foi reescrito de maneira que apenas a primeira expressão relacional da expressão lógica precisasse ser avaliada na maioria das iterações do laço do procedimento.

Para a avaliação da diretriz TRC_CTX, foi criado um procedimento que realiza leituras sequenciais (utilizando o comando SELECT INTO) das tuplas de uma tabela de banco de dados e as insere em uma variável do tipo de dados *VARRAY*. Em seguida, o procedimento foi reescrito de maneira que a população da variável do tipo de dados *VARRAY* com as tuplas da tabela do banco de dados fosse realizada utilizando o operador *BULK COLLECT*.

Todos os procedimentos criados para o experimento de desempenho estão disponíveis no site da ferramenta PL/SQL Advisor[2]

5.2.6 Design do Experimento

Nesta seção, são apresentados os tratamentos executados para a análise de impacto das diretrizes de eficiência investigadas no experimento. Para facilitar a legibilidade do design do experimento, os tratamentos são apresentados individualmente para cada diretriz de eficiência investigada. Na Tabelas 5.2 e 5.3, são apresentados os tratamentos referentes às diretrizes TI_NAT e ORD_LOG, respectivamente. Os tratamentos referentes à análise de impacto da diretriz TRC_CTX são apresentados na Tabela 5.4. Nos tratamentos apresentados nesta seção, os procedimentos P_n e P'_n (n $\in \{1, 2, 3\}$) representam a implementação do procedimento antes e depois da aplicação da diretriz de eficiência investigada.

5.2.7 Resultados e Discussão

Os resultados dos tratamentos para análise de desempenho relacionados à utilização de tipos de dados nativos (TI_NAT) são apresentados na Figura 5.7. Para todos os valores (10^4, 10^5,

[2]http://sites.google.com/site/plsqladvisor

Dr	**Pr**	**Qe**
TI_NAT	P_1	$\{10^4, 10^5, 10^6, 10^7\}$
TI_NAT	P_1'	$\{10^4, 10^5, 10^6, 10^7\}$

Tabela 5.2: Tratamentos do experimento para a análise de impacto da diretriz de eficiência TI_NAT.

Dr	**Pr**	**Qe**
ORD_LOG	P_2	$\{10^4, 10^5, 10^6, 10^7\}$
ORD_LOG	P_2'	$\{10^4, 10^5, 10^6, 10^7\}$

Tabela 5.3: Tratamentos do experimento para a análise de impacto da diretriz de eficiência ORD_LOG.

10^6 e 10^7) atribuídos à variável Qe nos tratamentos, a aplicação da diretriz TI_NAT impactou de forma positiva (i.e, diminuindo o tempo de execução) no procedimento de banco de dados utilizado. Para o maior valor da variável Qe (10^7) utilizado, foi percebida uma otimização de $\sim$75% no tempo de execução do procedimento. Nota-se que a aplicação da diretriz TI_NAT mostra-se bastante útil no contexto em que repetidas operações aritméticas são realizadas por procedimentos de banco de dados. Exemplos de contextos desta natureza são: algoritmos para manipulação de matrizes, fatoração de valores e processamento de dados recebidos em tempo real.

Os resultados dos tratamentos para análise de impacto relacionado à ordenação de expressões lógicas (ORD_LOG) são apresentados na Figura 5.8. Assim como no caso da diretriz TI_NAT, a diretriz ORD_LOG também apresentou impacto positivo em todos os valores atribuídos à variável Qe. Para o maior valor da variável Qe (10^7) utilizado, foi percebida uma

Dr	**Pr**	**Qe**
TRC_CTX	P_3	$\{10^3, 10^4, 10^5, 10^6\}$
TRC_CTX	P_3'	$\{10^3, 10^4, 10^5, 10^6\}$

Tabela 5.4: Tratamentos do experimento para a análise de impacto da diretriz de eficiência TRC_CTX.

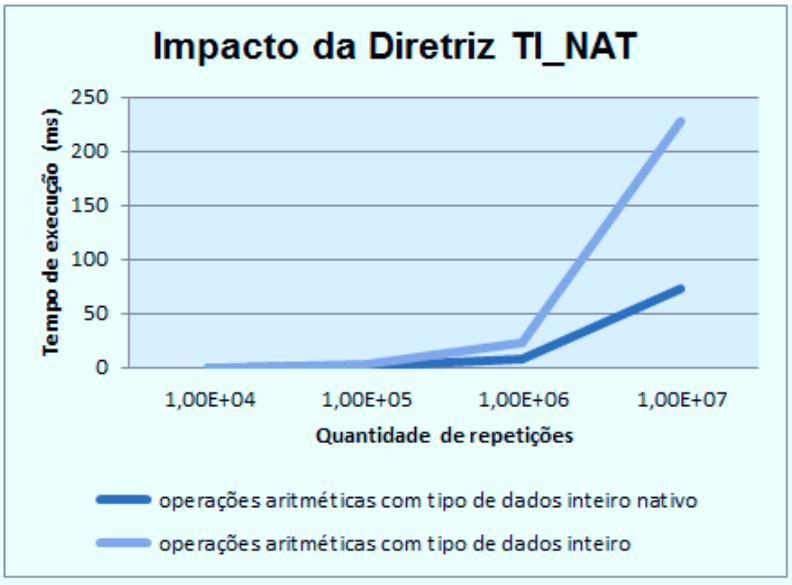

Figura 5.7: Resultados do impacto da diretriz de eficiência TI_NAT.

otimização de aproximadamente $\sim$78% no tempo de execução do procedimento. Ainda que o impacto da aplicação desta diretriz comece a ser notório apenas a partir do valor 10^6 da variável Qe, deve ser levado em consideração o fato da implementação do procedimento P_2 incluir apenas 4 termos na expressão lógica avaliada. Além disso, cada termo da expressão lógica utilizada realiza apenas a avaliação de uma expressão relacional. Em um contexto real, onde pode haver expressões lógicas mais extensas e o custo de avaliação dos termos ser significativamente maior (envolvendo chamadas a funções, execução de comandos de repetições e/ou acessos a banco de dados), a aplicação da diretriz ORD_LOG pode acarretar em otimizações mais notórias.

Os resultados dos tratamentos para análise de impacto relacionado à diminuição da quantidade de trocas de contexto executadas pelo procedimento são apresentados na Figura 5.9. Dentre as diretrizes de eficiência investigadas, a diretriz TRC_CTX apresentou o maior impacto no tempo de execução do procedimento. Ao atribuir o valor (10^6) à variável Qe, a execução do procedimento P_4 foi otimizada em mais de $\sim$92%. O impacto superior desta diretriz justifica-se pelo fato de operadores que envolvem acesso a banco de dados usualmente acarretarem em esperas significativas na execução de procedimentos de banco de dados. Os resultados da análise de impacto da diretriz de eficiência TRC_CTX são particularmente importantes, pois: i) apresentam indícios do impacto real (na escala de segundos)

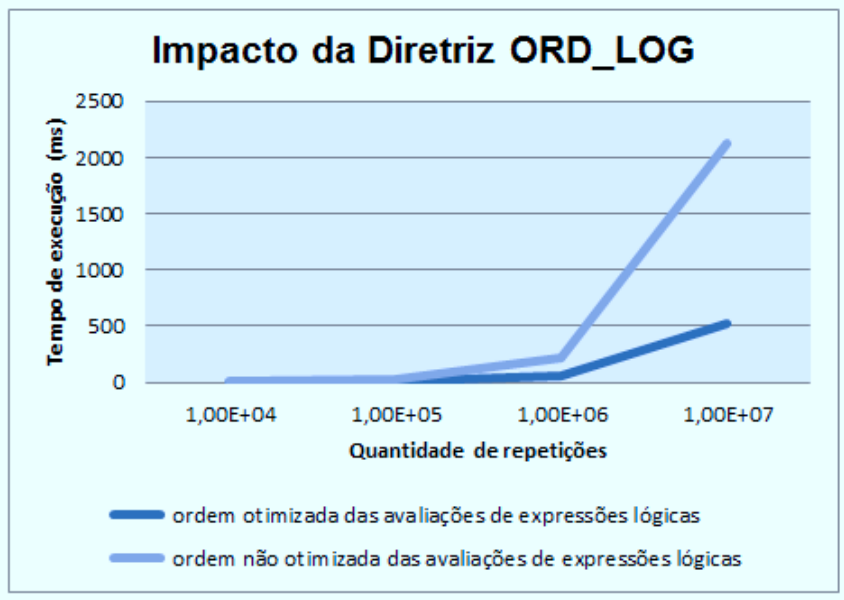

Figura 5.8: Resultados do impacto da diretriz de eficiência ORD_LOG.

da aplicação desta diretriz em procedimentos de banco de dados; e ii) demonstram como acessos sequenciais a tuplas armazenadas em banco de dados, operações bastante rotineiras na implementação de procedimentos, podem atrasar significativamente a execução dos mesmos. Nesse contexto, nota-se que a aplicação da diretriz TRC_CTX pode ser bastante efetiva em aplicações cujas tabelas armazenam uma quantidade razoável de tuplas (mais de 10^6) e cujos procedimentos de banco de dados necessitem manipular ou acessar a maioria das tuplas armazenadas em tabelas desta natureza.

No geral, todas diretrizes investigadas apresentaram impacto positivo no tempo de execução dos procedimentos de banco de dados utilizados no experimento de desempenho. Além disso, os resultados deixam claro que, quanto mais vezes os procedimentos são chamados, maior é o impacto da aplicação das diretrizes no tempo de execução dos mesmos. Ainda que algumas diretrizes investigadas tenham apresentado impactos significativos apenas a partir de 10^6 chamadas repetidas, a aplicação destas diretrizes pode tornar-se bastante relevante em cenários nos quais uma grande quantidade de dados é manipulada em banco de dados por procedimentos, tais como no contexto de um banco de dados de uma rede social, governamental ou de uma empresa de grande porte. Além disso, pequenas otimizações nos procedimentos de banco de dados podem acarretar em ganhos significativos se tais proce-

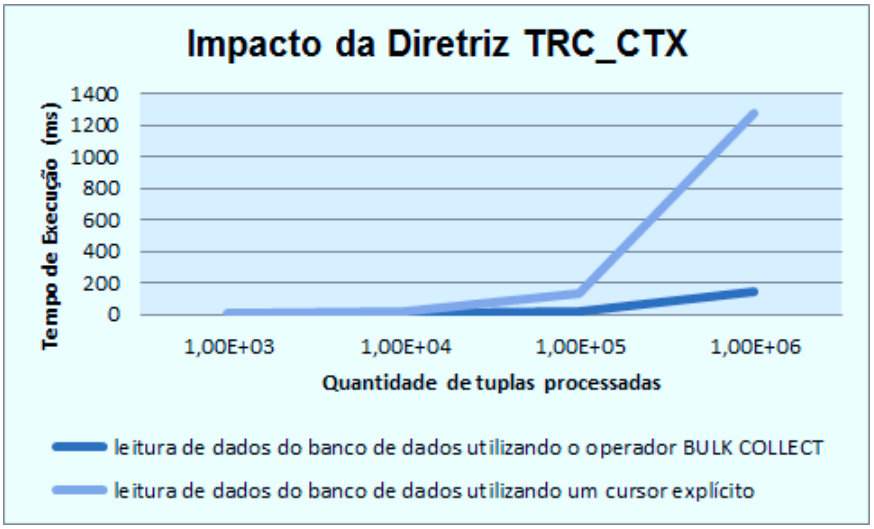

Figura 5.9: Resultados do impacto da diretriz de eficiência TRC_CTX

dimentos são chamados em laços pela camada de aplicação e/ou chamados por várias aplicações distintas. Por exemplo, se três aplicações distintas realizam, cada uma, chamadas ao procedimento P em um comando de repetição com 10 iterações, uma otimização de 200 milésimos diminui o tempo de espera em 6 segundos a cada vez que P for chamado pelas três aplicações.

5.2.8 Ameaças à Validade

Nesta seção, são discutidas as principais ameaças à validade dos resultados do experimento de análise de desempenho realizado.

Validade Externa. A principal ameaça para a validade dos resultados obtidos com os experimentos de desempenho decorre ao fato de que os procedimentos utilizados foram criados artificialmente. A aplicação das diretrizes de eficiência investigadas pode acarretar impactos diferentes (superiores ou inferiores) em contextos reais. Desse modo, testar o impacto das diretrizes no contexto de procedimentos e cargas de trabalho reais resultaria em resultados mais importantes para a avaliação por desenvolvedores de procedimentos de banco de dados.

Validade de Construção. Os experimentos de análise de desempenho não consideraram fatores que também podem influenciar o tempo de execução dos procedimentos, tais como configurações globais de memória do SGBD, o tipo de sistema operacional, a configuração de hardware do computador e a linguagem de programação utilizada. A variação do nível

destes fatores nos experimentos aumentaria a generabilidade dos resultados obtidos.

5.3 Estudo de Caso com Procedimentos Reais de Código-fonte Aberto

Nesta seção, é apresentado um estudo de caso realizado com o objetivo de avaliar a eficácia e a eficiência da abordagem proposta no contexto de procedimentos de banco de dados reais. A eficácia da abordagem foi medida analisando a quantidade de advertências reportadas pela ferramenta implementada após analisar o código-fonte de procedimentos de banco de dados reais. Esta métrica medida no estudo de caso permite a análise de dois aspectos da abordagem: i) a investigação sobre a ocorrência das diretrizes pré-definidas investigadas neste trabalho no contexto de procedimentos de banco de dados de código-fonte livre reais; e ii) dado a existência de procedimentos de banco de dados que permitem melhorias de eficiência e qualidade, analisar a capacidade da abordagem proposta em detectar estes cenários.

A eficiência da abordagem foi avaliada medindo o tempo de análise automática realizado pela ferramenta ao analisar os procedimentos de banco de dados do estudo de caso. A avaliação desta métrica no estudo de caso permite a análise dos seguintes aspectos da abordagem: i) investigar o tempo requerido pela abordagem para a realização de análise estática automática do código-fonte de procedimentos de banco de dados reais; ii) avaliar a viabilidade da utilização da abordagem proposta em um processo de desenvolvimento de software real, para tal, a mesma deve representar pouco atraso no tempo de análise; e iii) permitir uma avaliação inicial do tempo de análise automática da abordagem proposta comparado ao tempo de uma análise equivalente realizada de forma manual (i.e., realizada por desenvolvedores).

Inicialmente (Seção 5.3.1), são apresentados detalhes sobre os projetos analisados no estudo de caso. Detalhes da instrumentação do estudo de caso são mostrados na Seção 5.3.2. Finalmente (Seção 5.3.3), os resultados do estudo de caso são apresentados juntamente com a discussão sobre os mesmos.

5.3.1 Projetos Analisados

Foram utilizados quatro projetos distintos para a condução do estudo de caso. Os projetos são de código-fonte aberto e foram extraídos de um repositório[3] online de projetos desenvolvidos na linguagem PL/SQL. Dentre as opções disponíveis no repositório utilizado, foram escolhidos os projetos que possuíam mais linhas código-fonte. Desta forma, os seguintes projetos foram selecionados:

- **JSON Library**: consiste em um conjunto de ferramentas para gerar objetos JSON (JavaScript Object Notation) a partir de código-fonte PL/SQL;
- **CodeBrew**: consistem em um *framework* para desenvolvedores do PL/SQL *Gateway* (também conhecido como mod_plsql);
- **DBLens**: É um conjunto de ferramentas baseadas em banco de dados Oracle para realizar filtragem colaborativa de dados;
- **STR**: Consiste em um conjunto de ferramentas para manipulação de palavras;

De cada um destes projetos, foram selecionados 10 procedimentos de banco de dados para o estudo de caso. Foram selecionados os procedimentos que apresentaram mais linhas de código-fonte procedimental, uma vez que esta característica é preferível para a análise automática realizada pela abordagem e permite que discussões mais interessantes sejam realizadas acerca dos resultados do estudo de caso.

Na Tabela 5.5 são apresentadas algumas características dos procedimentos de banco de dados extraídos dos projetos de código-fonte aberto, tais como quantidade de procedimentos selecionados e quantidade de linhas de código analisadas.

5.3.2 Instrumentação

A instrumentação do tempo de análise automática da abordagem foi realizada utilizando a estratégia de temporizadores. A função nativa *System.currentTimeMillis()* do JAVA foi utilizada para medir o tempo de execução de cada procedimento analisado pela ferramenta

[3] http://plnet.org

Projeto	#Procedimentos	#Linhas de Código Analisadas
JSON Library	10	268
CodeBrew	10	473
DBLens	10	829
STR	10	314

Tabela 5.5: Detalhes dos projetos de código-fonte aberto selecionados para o estudo de caso.

PL/SQL Advisor. Na apresentação dos resultados (Seção 5.3.3) do estudo de caso, o tempo de análise da ferramenta foi reportado em milissegundos.

A instrumentação da eficácia da abordagem, ou seja, a quantidade de advertências reportadas pela análise automática, foi realizada utilizando o conteúdo do arquivo de saída das advertências salvo pelo componente PERSISTENCE da ferramenta PL/SQL Advisor.

5.3.3 Resultados e Discussão

Na Tabela 5.6 são apresentados os resultados gerais do estudo de caso. Para cada projeto utilizado no estudo de caso, são reportados: i) o tempo (TA) gasto para a realização da análise automática nos 10 procedimentos analisados pela ferramenta; ii) a quantidade de linhas de código-fonte analisadas automaticamente (LdC); iii) a quantidade de advertências reportadas (AR) pela ferramenta PL/SQL Advisor após a análise dos 10 procedimentos selecionados de cada projeto; e iv) a quantidade de advertências relacionadas a diretrizes de eficiência (AER) e qualidade (AQR). Na Figura 5.10 são mostrados gráficos de barra contendo a distribuição das advertências reportadas após a análise automática de todos os projetos do estudo de caso, por diretriz de eficiência e qualidade.

Eficácia da Abordagem. No total, a ferramenta reportou 299 advertências após a análise de 1884 linhas de código-fonte dos 40 procedimentos de banco de dados analisados. Dentre as advertências reportadas, foram destacadas tanto aquelas relacionadas a melhorias gerais, quanto aquelas relacionadas ao contexto de banco de dados. Ainda que no estudo de caso as advertências falso-positivas não tenham sido filtradas, a quantidade de advertências reportadas pela ferramenta no estudo de caso representa um bom indício de que as diretrizes de eficiência e qualidade investigadas neste trabalho podem ser aplicadas no código-fonte de

procedimentos em ambientes reais. Além disso, estes resultados também apontam que a execução de análises estáticas sobre uma representação simplificada (ACD) de procedimentos de banco de dados pode mostra-se bastante efetiva para a detecção de trechos em que podem ser aplicadas diretrizes pré-definidas de qualidade e eficiência.

Eficiência da Abordagem. A ferramenta PL/SQL Adivsor necessitou de menos de 7 segundos para analisar e criar advertências para 1884 linhas de código-fonte de procedimentos de banco de dados. O tempo total requerido pela análise automática é claramente inferior ao tempo necessário para realizar uma análise manual dos mesmos 40 procedimentos. Além disso, a inspeção manual é um processo propenso a erros que pode ocasionalmente falhar em detectar possibilidades de melhorias em procedimentos de banco de dados. Por outro lado, uma vez que a abordagem automática seja implementada de forma correta, é garantido que todas as possibilidades de melhorias sejam devidamente reportadas na saída da análise. Esta afirmação é também reforçada pelos resultados dos Experimentos I e II (Seções 5.4.3 e 5.4.4).

A abordagem automática demonstrou não ser sensitiva à quantidade de linhas de código-fonte analisadas ou à complexidade dos procedimentos de banco de dados. Por este motivo, a abordagem pode ser utilizada para analisar uma grande quantidade de procedimentos de banco de dados a um custo bastante reduzido. Na prática, a abordagem automática pode ser executada repetidas vezes (após mudanças subsequentes no código-fonte) durante a fase de desenvolvimento de procedimentos de banco de dados sem que o tempo de análise automática represente um gargalo para o processo de desenvolvimento. Desse modo, a utilização da abordagem proposta pode reduzir os custos iniciais (referentes ao escopo da análise) dos processos de ajuste de desempenho e refatoramento do código-fonte de procedimentos de banco de dados.

Projeto	**TA**	**LdC**	**AR**	**AER**	**AQR**
JSON Library	2,33 s	268	61	16	45
CodeBrew	1,35 s	473	56	22	34
DBLens	1,87 s	829	120	80	40
STR	1,26 s	314	62	27	35

Tabela 5.6: Resultados do estudo de caso por projeto analisado.

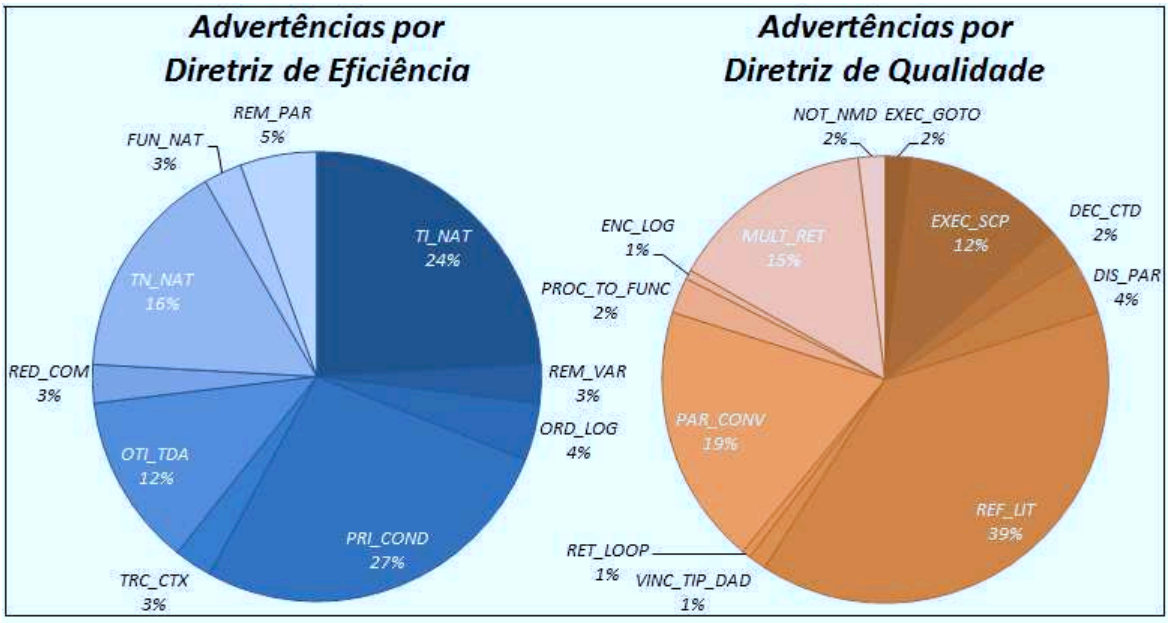

Figura 5.10: Distribuição das advertências reportadas no estudo de caso por diretriz de eficiência e qualidade.

Distribuição das Advertências por Diretriz de Eficiência. Das 299 advertências reportadas pela ferramenta, 145 foram relacionadas à eficiência dos procedimentos de código-fonte aberto. As advertências relacionadas a mudanças em tipos de dados (diretrizes TI_NAT, TN_NAT, TD_RES e OTI_TDA) foram reportadas 58 vezes. A aplicação destas advertências pode acarretar em um impacto significativo nos procedimentos que realizam muitas operações aritméticas. O fato da classe de advertências relativa a mudanças em tipos de dados ter sido muito frequentemente reportada pode ser explicado por dois motivos: i) parte destas advertências podem ser na prática, falso-positivas, uma vez que nem sempre é viável utilizar tipos de dados nativos para a execução de operações aritméticas ou, no contexto dos projetos, o impacto de melhoria na eficiência nos procedimentos não justifica as mudanças sugeridas pelas advertências; e ii) os desenvolvedores do projeto *DBLens* muito provavelmente não estavam cientes da existência das diretrizes desta classe, uma vez que a análise dos procedimentos deste projeto resultou em 46 advertências (relacionadas as diretrizes TI_NAT, TN_NAT, TD_RES e OTI_TDA) reportadas pela ferramenta.

As advertências relacionadas a mudanças na ordem de avaliação de expressões (diretrizes PRI_COND e ORD_LOG) foram reportadas 45 vezes no estudo de caso. Ainda que a

possibilidade prática da aplicação destas diretrizes dependa da avaliação dos desenvolvedores do projeto, acredita-se que uma parte considerável destas advertências pode ser aplicada de forma efetiva nos procedimentos analisados. As advertências relacionadas à otimização de operações de acesso a banco de dados (diretrizes TRC_CTX, RED_COM e CUR_IMP) foram reportadas 12 vezes no estudo de caso. Ainda que esta classe de diretrizes não tenha sido a mais frequentemente reportada, considerando os resultados do experimento de desempenho (Seção 5.2) e os resultados reportados na literatura relacionada [13] [9] [12] [5] [4], é bastante provável que a aplicação destas diretrizes cause impactos bastante significativos no tempo de execução dos procedimentos analisados.

A ferramenta reportou 12 advertências relacionadas à remoção de identificadores não utilizados (REM_PAR, REM_VAR e DEC_REP) pelos procedimentos de banco de dados. Uma vez que a detecção destas diretrizes é decidível de forma estática, é certo que a aplicação destas diretrizes irá diminuir a memória utilizada pelos procedimentos e pode facilitar a compreensão dos procedimentos pelos desenvolvedores. Em especial, a análise dos procedimentos do projeto *DBLens* desencadeou a maioria (75%) das advertências desta natureza. Os procedimentos do projeto DBLens são, em geral, bastante extensos. Desse modo, é provável que os desenvolvedores deste projeto eventualmente tenham esquecido de remover identificadores não utilizados devido ao tamanho e complexidade dos procedimentos criados.

Distribuição das Advertências por Diretriz de Qualidade. Das 299 advertências reportadas pela ferramenta, 145 foram relacionadas à qualidade dos procedimentos de código-fonte aberto analisados. As advertências relacionadas à simplificação do fluxo de controle do código (diretrizes EXEC_GOTO, EXEC_SCP, RET_LOOP, MULT_RET) foram reportadas 45 vezes pela ferramenta. Esta natureza de advertências contemplou uma parte significativa das advertências de qualidade (aproximadamente 30%). A aplicação destas advertências pode simplificar consideravelmente o fluxo de controle dos procedimentos analisados e melhorar a legibilidade do código-fonte para os desenvolvedores.

Considerando as diretrizes DEC_CTD, NULL_RET, ELSE_CASE, ABT_CUR_EXP, VINC_TIP_DAD, foram reportadas um total de 9 advertências. Ainda que não representem uma quantidade significativa do total analisado, estas advertências são bastante úteis para evitar a ocorrência de erros de execução nos procedimentos e melhorar partes do código-fonte que se tornaram muito complexas de serem entendidas pelos desenvolvedores.

As advertências relacionadas às diretrizes PAR_CONV, DISP_PAR, DISP_VAR e PROC_TO_FUNC foram reportadas 49 vezes no estudo de caso. Estas diretrizes contemplaram uma porção representativa (aproximadamente 32%) das advertências de qualidade. Na prática, estas advertências são úteis para estabelecer padrões e convenções ao longo do código-fonte dos procedimentos analisados. As advertências de qualidade mais frequentemente reportadas (61 vezes) foram as relacionadas à utilização de identificadores semanticamente significativos (ENC_LOG e REF_LIT). Em especial, as advertências relacionadas à diretriz REF_LIT foram as mais frequentemente reportadas (60 vezes). Aparentemente, os desenvolvedores dos projetos analisados não investiram um esforço significativo para a criação de constantes para literais (muitos dos quais bastante difíceis de serem interpretados) referenciados pelos procedimentos de banco de dados.

Em geral, a grande maioria das advertências reportadas no estudo de caso está relacionada a um subconjunto específico de diretrizes: TI_NAT, TN_NAT, PRI_COND, OTI_TDA, RET_LOOP, MULT_RET, PAR_CONV e REF_LIT. As advertências relacionadas às demais diretrizes contemplaram uma porção inferior ou não foram reportadas no estudo de caso. Estes resultados apontam que algumas das diretrizes investigadas neste trabalho são menos lembradas e/ou corrigidas pelos desenvolvedores durante a fase de desenvolvimento de procedimentos de banco de dados.

5.3.4 Ameaças à Validade

Nesta seção, são discutidas as principais ameaças à validade da metodologia e resultados do estudo de caso conduzido.

Validade de Conclusão. A eficácia da abordagem proposta foi medida no estudo de caso utilizando a quantidade de advertências reportadas pela análise automática. Uma vez que, pela impossibildade de contato com os desenvolvedores dos projetos utilizados, não foi possível calcular a porcentagem das advertências reportadas que são de fato falso-positivas, não é possível afirmar com certeza a aplicabilidade real das advertências reportadas no estudo de caso.

A utilização da abordagem proposta em processos reais de análise de código-fonte de procedimentos de banco de dados permitiria a coleta de métricas mais significativas a respeito do impacto prático da utilização da abordagem. Neste cenário, seria possível avaliar

a porcentagem real das advertências reportadas automaticamente que seriam aplicadas pelos desenvolvedores. Além disso, seria possível medir o tempo gasto pelos desenvolvedores para interpretar as advertências, e assim, calcular qual o tempo total economizado devido à utilização da análise automática do código-fonte dos procedimentos.

Validade Externa. Ainda que tenham sido utilizados projetos reais no estudo de caso, existem alguns problemas relacionados a escolha das unidades experimentais. Apenas 4 projetos foram utilizados para testar a abordagem no estudo de caso. Esta quantidade de projetos não é suficiente para generalizar os resultados obtidos. Além disso, é provável que os procedimentos analisados já tenham passado por processos de refatoramento e ajuste de desempenho. Por esta razão, é também importante que a abordagem proposta seja testada no contexto de procedimentos de banco de dados em processo de desenvolvimento ou desenvolvidos por programadores inexperientes. Isto porque, nestes cenários, é provável que a análise automática reporte advertências com maior possibilidade de aplicabilidade prática.

5.4 Experimentos com Procedimentos Industriais

Nesta seção, são apresentados dois experimentos envolvendo procedimentos de banco de dados industriais reais e desenvolvedores da linguagem PL/SQL. Com os resultados destes experimentos, objetiva-se discutir os seguintes aspectos da abordagem proposta: i) analisar a ocorrência das diretrizes pré-definidas investigadas neste trabalho no contexto de procedimentos de banco de dados industriais reais; ii) dado a existência de procedimentos de banco de dados industriais que permitem melhorias de eficiência e qualidade, analisar a capacidade da abordagem proposta em detectar estes cenários; iii) comparar a eficiência da abordagem proposta para análise de código-fonte de procedimentos de banco de dados com a eficiência da abordagem manual (i.e., realizada por desenvolvedores) para o mesmo fim; iv) comparar a eficácia da abordagem proposta para análise de código-fonte de procedimentos de banco de dados com a eficácia da abordagem manual para o mesmo objetivo; v) avaliar a eficácia das heurísticas executadas para diminuição da advertências falso-positivas reportadas pela abordagem em um contexto de procedimentos de banco de dados industriais reais; e vi) avaliar qualitativamente a abordagem proposta de acordo com a opinião de desenvolvedores de procedimentos de banco de dados reais.

Os experimentos apresentados nesta seção foram denominados Experimento I e Experimento II. O Experimento I visa investigar a eficiência e a eficácia da abordagem manual (i.e., realizada por desenvolvedores) para a análise de código-fonte de procedimentos de banco de dados industriais. O Experimento II investiga a eficiência e a eficácia da abordagem proposta para o mesmo fim. Em ambos os experimentos, os mesmos procedimentos de banco de dados foram utilizados como unidades experimentais. Este cenário permite a comparação direta entre as métricas coletadas pelos dois experimentos.

Os planejamentos dos Experimentos I e II são descritos nas seções 5.4.3 e 5.4.4, respectivamente. Após a condução dos Experimentos I e II, foi realizado um questionário com os desenvolvedores participantes do Experimento I. Os resultados do questionário são apresentados na Seção 5.4.5. Finalmente, a discussão dos resultados dos Experimentos I e II é apresentada na Seção 5.4.6.

5.4.1 Seleção do Contexto

Para a realização dos Experimento I e II, foram utilizados procedimentos de banco de dados industriais reais utilizados no projeto Malha Brasil[4] (MB). Este projeto foi financiado pela PETROBRAS[5] e consiste em uma solução para o escalonamento de modais (por exemplo, dutos e navios) para o escoamento da cadeia de suprimentos dos produtos e matérias-primas confeccionados e utilizados pela empresa. O projeto foi desenvolvido no Laboratório de Sistemas Distribuídos[6] (LSD) da Universidade Federal de Campina Grande[7].

O projeto MB utiliza um banco de dados Oracle para o armazenamento dos dados da cadeia de suprimentos da PETROBRAS. Além disso, uma considerável parte da lógica de negócios do projeto é implementada utilizando procedimentos de banco de dados escritos na linguagem PL/SQL. No contexto do projeto Malha Brasil, todos os desenvolvedores do projeto foram utilizados na condução do Experimento I e um subconjunto dos procedimentos de banco de dados utilizados no projeto foi selecionado como unidade experimental dos Experimentos I e II.

[4] http://www.mas-scm.lsd.ufcg.edu.br
[5] http://www.petrobras.com
[6] http://www.lsd.ufcg.edu.br
[7] http://www.ufcg.edu.br

5.4.2 Unidades Experimentais

Seleção dos Desenvolvedores. Todos os desenvolvedores do projeto Malha Brasil foram incluídos na realização do Experimento I. Os mesmos participaram do experimento de forma voluntária. A equipe do projeto MB possui aproximadamente 2,5 anos de experiência com a Linguagem PL/SQL. Neste capítulo, os desenvolvedores são referenciados utilizando os acrônimos: D1, D2 e D3.

Seleção dos Procedimentos. O Projeto Malha Brasil possui 49 pacotes Oracle, os quais englobam um total de 263 procedimentos PL/SQL. Deste total, foram selecionados quatro subconjuntos de procedimentos para serem utilizados nos Experimentos I e II: A (173 linhas de código), B (244 linhas de código), C (257 linhas de código) e D (156 linhas de código). Cada subconjunto contém 7 procedimentos de banco de dados[8].

Criação de Procedimentos Mutantes. Uma vez que a maior parte dos procedimentos PL/SQL utilizados pelo projeto Malha Brasil já se encontra em produção, é provável que tais procedimentos já tenham passado por diversos processos de inspeção manual, contemplando refatoramentos e ajustes de desempenho. Desse modo, é esperado que estes procedimentos apresentem uma baixa quantidade de problemas de eficiência e qualidade. Com o intuito de investigar a eficácia e eficiência das abordagens investigadas no contexto de procedimentos de banco de dados que possuam uma quantidade elevada de problemas de eficiência e qualidade, dois dos subconjuntos de procedimentos selecionados (C e D) foram selecionados para a criação de procedimentos de banco de dados mutantes. A inserção de problemas de eficiência e qualidade nos procedimentos dos subconjuntos C e D foi realizada de maneira manual. Neste capítulo, os subconjuntos C e D são referenciados utilizando a expressão "subconjuntos mutantes".

5.4.3 Experimento I

Objetivos

O objetivo do Experimento I é avaliar a eficácia e eficiência da atividade de inspeção manual para a verificação da conformidade do código-fonte dos procedimentos de banco de dados do

[8]Por motivos de confidencialidade do projeto Malha Brasil, não é possível publicar informações sobre os procedimentos utilizados.

projeto *Malha Brasil* com diretrizes pré-definidas de eficiência e qualidade. Para tal, foram medidas a quantidade de advertências reportadas pelos desenvolvedores (eficácia) após a análise dos procedimentos selecionados do projeto *Malha Brasil* e o tempo de análise manual (eficiência) requerido para esta atividade.

Variáveis

O Experimento I utilizou um total de 5 variáveis. Dentre estas, 3 variáveis são independentes: Desenvolvedor do Projeto (DP), Subconjunto de Procedimentos PL/SQL (SP) e o Tempo de Restrição (TR) para análise manual no qual os desenvolvedores foram submetidos. Além destas, foram utilizadas duas variáveis dependentes: Tempo de Análise Manual (TAM) e quantidade de Advertências Detectadas Manualmente (ADM). Na Tabela 5.7, são apresentados detalhes sobre as variáveis do Experimento I.

Em resumo, TAM representa o tempo requerido por um desenvolvedor (DP) para realizar análise manual do código-fonte de todos os procedimentos de um subconjunto de procedimentos (SP) PL/SQL do projeto MB, sujeito a uma restrição de tempo (TR). Analogamente, ADM representa a quantidade de advertências reportadas por um desenvolvedor (DP) após a análise manual do código-fonte de todos os procedimentos de um subconjunto de procedimentos (SP) PL/SQL do projeto MB, sujeito a uma restrição de tempo (TR). No contexto do Experimento I, as restrições de tempo foram utilizadas com o objetivo de simular limitações no tempo de inspeções manuais de código-fonte impostas pelos processos de desenvolvimento em ambientes reais.

Variável	Abreviação	Escala	Níveis Utilizados
Desenvolvedor do Projeto	DP	Nominal	E1, E2, E3
Subconjunto de Procedimentos	SP	Nominal	{A, B, C, D}
Restrição de Tempo	RT	Razão	$\{1..t\}/t \in R$
Tempo de Análise Manual	TAM	Razão	$\{1..t\}/t \in R$
Advertências Detectadas Manualmente	ADM	Razão	$n/n \in N$

Tabela 5.7: Variáveis utilizadas no Experimento I.

Design

O design do Experimento I é mostrado na Tabela 5.8. A notação *VD(*VI_1, ... , VI_n*)* representa o valor resultante da variável dependente VD após a execução do tratamento $\{VI_1, ... , VI_n\}$ de variáveis independentes.

#Identificador	DP	SP	RT	TAM
Tr1	E1	A	Sem Restrição	ET(E1, A)
Tr2	E2	A	Sem Restrição	ET(E2, A)
Tr3	E3	A	Sem Restrição	ET(E3, A)
Tr4	E1	B	MAX(ET(E1, A), ET(E2, A), ET(E3, A)) / 2	ET(E1, B)
Tr5	E2	B	MAX(ET(E1, A), ET(E2, A), ET(E3, A)) / 2	ET(E2, B)
Tr6	E3	B	MAX(ET(E1, A), ET(E2, A), ET(E3, A)) / 2	ET(E3, B)
Tr7	E1	C	Sem Restrição	ET(E1, C)
Tr8	E2	C	Sem Restrição	ET(E2, C)
Tr9	E3	C	Sem Restrição	ET(E3 C)
Tr10	E1	D	MAX(ET(E1, C), ET(E2, C), ET(E3, C)) / 2	ET(E1, D)
Tr11	E2	D	MAX(ET(E1, C), ET(E2, C), ET(E3, C)) / 2	ET(E2, D)
Tr12	E3	D	MAX(ET(E1, C), ET(E2, C), ET(E3, C)) / 2	ET(E3, D)

Tabela 5.8: Design do Experimento I: Desenvolvedor do Projeto (DP), Subconjunto de Procedimentos (SP), Restrição de Tempo (RT) aplicada e Tempo de Análise Manual (TAM)

Execução e Instrumentação

Todas as diretrizes de eficiência e qualidade apresentadas no Capítulo 4 e Apêndice A foram discutidas com os desenvolvedores do projeto antes da execução do Experimento I. Além disso, os desenvolvedores utilizaram uma lista impressa contendo um resumo das diretrizes durante a análise manual dos procedimentos do projeto MB. Durante a execução do experimento, o código-fonte dos procedimentos contidos nos subconjuntos selecionados foi impresso e entregue aos desenvolvedores para que os mesmos apontassem os trechos nos quais as diretrizes discutidas poderiam ser aplicadas. As inspeções manuais foram realizadas de acordo com os tratamentos do Experimento I (Tabela 5.8).

Resultados

No intuito de comparar os resultados dos Experimentos I e II, foi criada uma variável dependente denotada Limiar Superior (LS), a qual representa o limiar superior da quantidade de advertências reportadas após a análise de cada subconjunto de procedimentos do projeto MB. Para tal, foram unidas as advertências reportadas automaticamente (função ADA) pela ferramenta PL/SQL Advisor (Seção 5.4.4) com as advertências anotadas manualmente pelos desenvolvedores. No entanto, uma vez que a ferramenta pode reportar advertências falso-positivas, foi pedido aos desenvolvedores que excluíssem da lista de advertências reportadas pela ferramenta aquelas consideradas falso-positivas pelos mesmos. Esta exclusão das advertências falso-positivas é representada pela função *Filtro*.

Desse modo, o limiar superior de advertências reportadas após a análise de um subconjunto (S) de procedimentos de banco de dados do projeto MB foi calculado como:

- $LS(S) = ADM(E1, S) \cup ADM(E2, S) \cup ADM(E3, S) \cup Filtro(ADA(PL/SQLAdvisor, S)); S = \{A, B, C, D\};$

Os resultados do Experimento I são mostrados na Tabela 5.9. São reportados o tempo de análise manual (TAM), o número de advertências relacionadas à eficiência (AE), qualidade (AQ) e a soma de ambas (ADM). Como também, o limiar superior (LS) para cada subconjunto de procedimentos analisado nos tratamentos e a proporção (ADM/LS) entre os resultados dos desenvolvedores e o limiar superior para cada tratamento.

5.4.4 Experimento II

Objetivos

O objetivo do Experimento II é avaliar a eficácia e eficiência da abordagem proposta para analisar automaticamente a conformidade do código-fonte de procedimentos de banco de dados do projeto MB com diretrizes pré-definidas de eficiência e qualidade. Para tal, foi medida a quantidade de advertências reportadas pela ferramenta PL/SQL Advisor (eficácia) após a análise dos procedimentos selecionados do projeto e o tempo de análise automática (eficiência) requerido para esta atividade.

#Tratamento	RT	TAM	ADM	AE	AQ	LS	(ADM/LS)
Tr1	Sem Restrição	34 min	15	12	3	55	27%
Tr2	Sem Restrição	25 min	19	9	10	55	34%
Tr3	Sem Restrição	33 min	7	6	1	55	12%
Tr4	17 min	17 min	8	3	5	54	14%
Tr5	17 min	16 min	39	8	31	54	72%
Tr6	17 min	17 min	13	8	5	54	24%
Tr7	Sem Restrição	26 min	36	4	32	87	42%
Tr8	Sem Restrição	23 min	47	16	31	87	52%
Tr9	Sem Restrição	26 min	30	15	15	87	34%
Tr10	13 min	13 min	16	1	15	67	23%
Tr11	13 min	11 min	22	4	18	67	32%
Tr12	13 min	13 min	31	17	14	67	46%

Tabela 5.9: Resultados do Experimento I.

Variáveis

O Experimento II utilizou um total de 4 variáveis. Dentre estas, apenas uma variável é independente: o Subconjunto de Procedimentos (SP) analisado pela ferramenta PL/SQL Advisor. Além destas, foram utilizadas 3 variáveis dependentes: Tempo de Análise Automática (TAA), a quantidade de Advertências Detectadas Automaticamente (ADA) e a quantidade de advertências detectadas automaticamente excluindo-se as consideradas falso-positivas pelos desenvolvedores (ADAEF). Na Tabela 5.10 são apresentados detalhes sobre as variáveis do Experimento II.

Em resumo, TAA representa o tempo requerido pela ferramenta PL/SQL Advisor para realizar análise automática do código-fonte de todos os procedimentos de um subconjunto de procedimentos (SP) PL/SQL do projeto MB. Analogamente, ADA representa a quantidade de advertências reportadas pela ferramenta após realizar análise automática do código-fonte de todos os procedimentos de um subconjunto de procedimentos (SP) PL/SQL do projeto MB.

Abreviação	**Natureza**	**Escala**	**Níveis Utilizados**
SP	Independente	Nominal	{A, B, C, D}
ADA	Dependente	Ratio	$n/n \in N$
ADAEF	Dependente	Razão	$n/n \in N$
TAA	Dependente	Razão	$\{1..t\}/t \in R$

Tabela 5.10: Variáveis do Experimento II.

Execução e Instrumentação

Todos os subconjuntos de procedimentos A, B, C e D foram analisados automaticamente pela ferramenta PL/SQL Advisor. O experimento foi realizado em um computador com a seguinte configuração: Intel Core i5 2410M, 4GB de memória RAM e sistema operacional Windows 7. O tempo de execução da ferramenta foi medido em milésimos de segundos, utilizando a função nativa do JAVA *System.currentTimeMillis()*.

Resultados

Os resultados do Experimento II são mostrados na Tabela 5.11. São reportados os tempos de análise automática (TAA) de cada subconjunto de procedimentos analisado, a quantidade de advertências relacionadas à eficiência (AE), qualidade (AQ) e a soma de ambas (ADAF). Estas três últimas variáveis representam a quantidade de advertências reportadas após a exclusão das advertências falso-positivas. Além disso, são também reportados a proporção entre a quantidade de advertências reportadas e o limiar superior para o subconjunto (ADAF/LS). Por último, é mostrada a proporção entre a quantidade de advertências falso-positivas (FP) reportadas pela ferramenta em relação à quantidade de advertências não consideradas falso-positivas somada à quantidade de advertências falso-positivas (FP/(FP + ADAF)).

5.4.5 Questionário

Após a execução dos Experimentos I e II, os resultados foram mostrados e discutidos com os desenvolvedores. Em seguida, foi entregue aos desenvolvedores um questionário contemplando as seguintes perguntas sobre a opinião dos mesmos em relação aos resultados dos

SP	TAA (msec)	ADAF	AE	AQ	LS	FP	(ADAF/LS)	FP/(FP + ADAF)
A	554	52	16	36	55	5	95%	9%
B	753	54	13	41	54	8	100%	13%
C	710	82	32	52	87	9	94%	10%
D	912	67	20	37	67	17	100%	20%

Tabela 5.11: Resultados do Experimento II.

Experimentos I e II:

- ☐ *I) Quantas advertências (em porcentagem) reportadas pela abordagem automática nos experimentos são úteis para melhorar a qualidade e eficiência de procedimentos de banco de dados*?

- ☐ *II) Quantas advertências (em porcentagem) reportadas pela abordagem automática nos experimentos são úteis para melhorar a qualidade e a eficiência dos procedimentos do banco de dados do projeto Malha Brasil*?

- ☐ *III) Quantas advertências (em porcentagem) reportadas pela abordagem automática são simples de serem interpretadas*?

- ☐ *IV) É mais provável que a interpretação das advertências reportadas pela abordagem automática atrase ou agilize o processo de inspeção do código-fonte de procedimentos de banco de dados*?

Os resultados do questionário são mostrados na Tabela 5.12.

5.4.6 Discussão dos Experimentos I e II

Esta seção discute a metodologia utilizada na execução dos Experimentos I e II e os resultados obtidos com os mesmos.

Eficácia da Abordagem Manual. Considerando os resultados mostrados na Tabela 5.9, na maioria dos casos a eficácia (ADM/LS) das análises manuais foi inferior a 55%. Com exceção do tratamento *Tr5*, a eficácia das inspeções manuais apresentou uma eficácia ainda

	Desenvolvedor		
	E1	**E2**	**E3**
Questão I	100%	∼75%	∼50%
Questão II	∼75%	∼75%	∼50%
Questão III	100%	100%	100%
Questão IV	Agilize o processo	Agilize o processo	Agilize o processo

Tabela 5.12: Resultados do questionário aplicado aos desenvolvedores do projeto MB.

inferior (abaixo de 35%) considerando apenas os subconjuntos de procedimentos não mutantes. Este resultado pode ser explicado devido ao fato dos procedimentos não mutantes (A e B) apresentarem inconformidades com diretrizes menos óbvias de serem detectadas, uma vez que os procedimentos destes subconjuntos provavelmente já passaram por processos de revisão manual e ajustes de desempenho durante a fase de desenvolvimento do projeto. As inspeções manuais apresentaram uma eficácia moderadamente superior (em média 38%) na análise dos subconjuntos de procedimentos mutantes (C e D). Analogamente, estes resultados aconteceram devido à presença de inconformidades com diretrizes mais óbvias de serem detectadas, uma vez que muitos dos problemas de eficiência e qualidade presentes nestes procedimentos foram inseridos manualmente.

Outro fato interessante relacionado à eficácia das inspeções manuais é que a mesma não sofreu grande influência das restrições de tempo aplicadas aos tratamentos no Experimento I. A média da eficácia das inspeções manuais nos tratamentos com restrição de tempo (∼33%) foi aproximadamente igual à média dos tratamentos nos quais foram aplicadas restrições de tempo (∼34%). Na prática, aumentar o tempo alocado aos desenvolvedores para a realização de inspeções manuais de procedimentos de banco de dados pode não resultar em melhorias significativas na eficácia dos mesmos. No geral, a análise manual realizada pelos desenvolvedores mostrou-se mais eficaz na detecção de possibilidades de aplicação de melhorias de qualidade. Este comportamento pode ser explicado pelo fato de que, em um contexto de procedimentos reais, é mais difícil detectar melhorias relacionadas à eficiência do código do que as relacionadas à qualidade do mesmo. Isto porque, quanto maior e mais complexo for o procedimento, mais complicado torna-se seu fluxo de dados e de controle, e mais difícil

torna-se a tarefa de detectar melhorias de eficiência.

Nos tratamentos do Experimento I nos quais não foram aplicadas restrições de tempo (subconjuntos A e C), a média do tempo de análise manual por subconjunto foi de 29 minutos. Comparada com a média de tempo de análise automática dos mesmos subconjuntos (Tabela 5.11), a abordagem manual apresenta-se como uma técnica bastante custosa para a análise de código-fonte de procedimentos de banco de dados. Além disso, o tempo necessário para a realização de análise manual é usualmente proporcional ao tamanho e à complexidade dos procedimentos analisados. Sendo assim, a abordagem manual tende a ser custosa nos casos em que uma parte considerável da lógica de negócios das aplicações é implementada na forma de procedimentos armazenados em banco de dados com tamanho e complexidade elevados. Por outro lado, os resultados do estudo de caso (Tabela 5.6) e do Experimento II (Tabela 5.11) apontam que a abordagem proposta é pouco influenciada pelas características (tamanho, complexidade, fluxo de controle e fluxo de dados) do código-fonte analisado.

Ainda que análises manuais sejam custosas, estas tarefas podem ser úteis para a detecção de otimizações específicas que envolvem condições de parada de comandos de repetição e melhorias no tempo assintótico de algoritmos, uma vez que a análise destes tipos de melhorias é complexa de ser automatizada. Além disso, outra vantagem da abordagem manual diz respeito a sua aplicabilidade. Diferente da abordagem automática, inspeções manuais não são limitadas por uma lista de diretrizes pré-definidas. Sendo assim, desenvolvedores podem detectar possibilidades de melhorias não consideradas por analisadores estáticos.

Eficácia da Abordagem Automática. A ferramenta PL/SQL Advisor reportou 106 advertências após a análise de 417 linhas de código-fonte dos procedimentos não mutantes (A e B) e reportou 142 advertências após a análise de 413 linhas de código-fonte dos procedimentos mutantes (C e D). A abordagem automática apresentou uma eficácia (ADAF/LS) superior à apresentada pelas inspeções manuais em todos os subconjuntos analisados (tanto na média dos desenvolvedores quanto em comparação individual).

Na análise automática dos subconjuntos mutantes, 37% das advertências reportadas estão relacionadas com diretrizes de eficiência e 63% das mesmas com diretrizes de qualidade. Acredita-se que a ocorrência de uma quantidade elevada de advertências de qualidade foi reportada devido ao fato de que os desenvolvedores não estavam cientes de algumas diretrizes

de qualidade específicas. Por exemplo, após a análise dos procedimentos não mutantes, a ferramenta reportou 49, 10, 9, 3, e 2 advertências relacionadas com as diretrizes de qualidade PAR_CONV, REF_LIT, EFT_COL_FUN, MULT_RET e EXEC_SCP, respectivamente.

A baixa quantidade de advertências de eficiência ($\sim$37%) reportadas pela ferramenta pode ser explicada pelo fato de que os procedimentos industriais analisados já foram possivelmente analisados em processos de ajuste de desempenho anteriores. No entanto, ainda que não representem a maioria das advertências reportadas, algumas advertências de eficiência específicas podem ser úteis para otimizar alguns procedimentos de banco de dados do projeto. Por exemplo, após a análise dos procedimentos não mutantes, a ferramenta reportou 17, 4, 2, 2, 2, 1, e 1 advertências relacionadas com as diretrizes de eficiência OTI_TDA, TN_NAT, TI_NAT, ORD_LOG, PRI_COND, RED_COM e TRC_CTX, respectivamente. A abordagem automática alcançou a eficácia ótima (ADAF/LS = 1) em dois dos quatro tratamentos do Experimento II. A ferramenta apenas não alcançou o limiar superior nas análises dos subconjuntos de procedimentos A e C devido a uma falha na implementação da análise relativa à diretriz EXEC_SCP. Esta falha foi devidamente corrigida após a execução dos Experimentos I e II.

Aplicabilidade da Abordagem Automática. Ao analisar os resultados relacionados com a quantidade de advertências falso-positivas reportadas pela ferramenta no contexto do projeto MB, nota-se que este problema não se apresenta como um empecilho grave para a utilização da abordagem proposta, tendo em vista os seguintes aspectos: i) apenas uma parte significantemente reduzida ($\sim$13%) das advertências reportadas foi considerada falso-positiva pelos desenvolvedores; ii) os desenvolvedores ressaltam nas respostas do questionário (Tabela 5.12) que a utilização da ferramenta pode agilizar o processo de análise de código-fonte de procedimentos de banco de dados, mesmo considerando os atrasos devido à avaliação manual das advertências falso-positivas; e iii) nos casos em que, em um contexto específico, uma quantidade excessiva de advertências falso-positivas sejam reportadas pela abordagem automática relacionadas a uma mesma diretriz, podem ser criadas e implementadas novas heurísticas no intuito de detectar os padrões que estão acarretando a criação dos resultados falso-positivos.

Os resultados do questionário aplicado aos desenvolvedores indicam, em resumo, uma boa aceitação da abordagem proposta pelos mesmos. Em todos os casos, os desenvolvedo-

res apontaram que pelo menos 50% das advertências reportada pela abordagem automática são de fato úteis para melhorar o código-fonte de procedimentos armazenados em banco de dados. Além disto, os desenvolvedores acreditam que o mesmo percentual ($\sim$50%) das advertências relacionadas aos procedimentos do projeto MB é útil para uma melhoria prática dos mesmos. Todos os desenvolvedores avaliaram as advertências reportadas pela ferramenta PL/SQL Advisor como simples de serem interpretadas. Essa facilidade resulta em um baixo custo relacionado à utilização da abordagem proposta, o que representa mais um indício de que a mesma pode ser aplicada em um cenário real; resultado este confirmado pelos desenvolvedores de acordo com os dados do questionário (Tabela 5.12).

5.4.7 Ameaças à Validade

Nesta seção, são discutidas as principais ameaças à validade da metodologia e resultados dos Experimentos I e II.

Validade Interna. A produtividade (eficácia e eficiência) das inspeções manuais realizadas pelos desenvolvedores pode ter sofrido variações durante a execução do Experimento 1. A eficácia dos tratamentos finais do Experimento I podem ter sido maiores devido a familiarização dos desenvolvedores com as diretrizes. Em outros casos, a eficácia e eficiência dos últimos tratamentos podem ter sido diminuídas devido a fatores como cansaço ou desânimo dos desenvolvedores. Por estas razões, a acurácia das métricas de produtividade dos desenvolvedores pode não ter sido muito precisa

Validade de Construção. A criação dos procedimentos mutantes utilizados nos Experimentos I e II não seguiu um processo formalmente definido, ao invés disso, os problemas foram inseridos nos procedimentos de forma manual e aleatória. Por esta razão, a criação desta classe de procedimentos pode ter sido enviesada pelo conhecimento prévio dos autores da capacidade de análise da abordagem proposta.

Validade Externa. Ainda que os procedimentos os procedimentos utilizados nos Experimentos I e II sejam reais, todos eles foram selecionados de um mesmo projeto industrial. Além disso, todos os desenvolvedores que se voluntariaram para participação no projeto possuem experiência semelhante com a linguagem PL/SQL e também estavam ligados a um único projeto industrial. Sendo assim, ainda que os resultados dos Experimentos I e II possam apresentar bons indícios sobre a efetividade das abordagens manual e automática

no contexto de análise de procedimentos, uma variação mais concreta nos níveis das unidades experimentais e sujeitos utilizados permitiria uma generalização maior dos resultados obtidos.

Capítulo 6

Conclusões

Neste trabalho, foi proposta uma abordagem, baseada em análise estática de código-fonte, para sugerir melhorias em procedimentos de banco de dados. A abordagem baseia-se em conceitos consolidados da área de análise estática, tais como gráfico de fluxo de controle, dominância, árvore de controle de dependência e busca hierárquica em árvores. Estes conceitos foram utilizados no intuito de checar automaticamente a conformidade de procedimentos de banco de dados com diretrizes pré-definidas de eficiência e qualidade, tanto relacionadas a melhorias gerais quanto as específicas ao contexto de banco de dados. Como mostrado no Capítulo 4 e Apêndice A, as diretrizes investigadas neste trabalho estão presentes em diversas linguagens de programação procedimentais de banco de dados, tais como PL/SQL, Transact-SQL e PL/pg-SQL.

A abordagem proposta é utilizada na etapa inicial dos processos de refatoramento e ajuste de desempenho de procedimentos de banco de dados, a qual consiste na análise do código-fonte dos procedimentos. Inicialmente, a abordagem transforma o código-fonte do procedimento de banco de dados em uma representação equivalente na forma de Árvore de Controle de Dependência (ACD). Em seguida, esta representação é analisada por módulos responsáveis por detectar padrões no código-fonte nos quais podem ser aplicadas diretrizes pré-definidas de eficiência e qualidade. Durante esta etapa, são aplicadas heurísticas no intuito de diminuir as chances de criação de advertências falso-positivas. Como resultado da etapa de análise, são gerados Resumos da Análise, os quais contêm os padrões identificados e as respectivas posições destes padrões no código-fonte. Estes resumos são então processados no intuito de resolver conflitos entre advertências e, em seguida, são criadas advertências

para possíveis melhorias no código-fonte. Finalmente, estas advertências são apresentadas ao usuário da ferramenta, ordenadas de acordo com critérios específicos, tais como subclasse da advertência, frequência da advertência, dificuldade de detecção da advertência e/ou impacto da advertência no desempenho do código-fonte.

A abordagem proposta é suficientemente genérica para a criação de analisadores estáticos para qualquer linguagem de programação procedimental de banco de dados. Como prova de conceito, a abordagem proposta foi instanciada em uma ferramenta denominada PL/SQL Advisor, que objetiva analisar automaticamente procedimentos escritos em PL/SQL. A abordagem proposta foi avaliada em um estudo de caso e dois experimentos. Inicialmente, foram realizados experimentos de desempenho para medir o impacto de diretrizes individuais de eficiência no tempo de execução de procedimentos de banco de dados. Nas maiores cargas de trabalho investigadas, foi possível otimizar o tempo de execução de procedimentos de banco de dados em $\sim$80% (TI_NAT), $\sim$82% (ORD_LOG) e $\sim$91% (TRX_CTX). Além disso, este impacto pode ser consideravelmente aumentado se os procedimentos forem chamados em comandos de repetição pela camada de aplicação.

Em seguida, a abordagem foi avaliada em um estudo de caso com projetos de código-fonte aberto reais. A ferramenta PL/SQL Advisor reportou 299 advertências após a análise de 40 procedimentos de banco de dados reais e necessitou de um tempo bastante reduzido ($\sim$7 segundos) para a análise de 1884 linhas de código. As diretrizes mais frequentemente reportadas no estudo de caso foram: TI_NAT, TN_NAT, PRI_COND, OTI_TDA, RET_LOOP, MULT_RET, PAR_CONV e REF_LIT.

Por último, foram comparadas a eficácia e eficiência da abordagem automática proposta com inspeções manuais realizadas no contexto de um projeto industrial real (*Malha Brasil*). A abordagem apresentou eficácia e eficiência superiores aos desenvolvedores na análise de todos os subconjuntos de procedimentos selecionados do projeto industrial. Além disso, apenas uma quantidade reduzida (13% em média) das advertências reportadas pela abordagem foram consideradas falso-positivas pelos desenvolvedores do projeto. Mesmo considerando que as inspeções manuais apresentaram eficácia e eficiência inferiores à abordagem automática, a primeira técnica mostrou-se bastante eficaz na detecção de melhorias de qualidade. Além disso, inspeções manuais são importantes para a análise de classes de melhorias de eficiência específicas que são difíceis de serem automatizadas, tais como otimização assintótica

de algoritmos e mudanças na condição de parada de comandos de repetição.

A abordagem proposta não garante a corretude das análises de conformidade dos procedimentos com as diretrizes investigadas. No entanto, ainda que apresente esta limitação, a mesma mostrou-se bastante eficaz. Como apontam os resultados do estudo de caso (Seção 5.3.4) e Experimento II (Seção 5.4.6), ainda que sejam simples de serem aplicadas, as diretrizes discutidas neste trabalho são frequentemente ignoradas por desenvolvedores na criação de procedimentos de banco de dados reais.

Ainda que, na prática, as advertências falso-positivas reportadas pela abordagem proposta possam atrasar o processo de avaliação das advertências, as heurísticas apresentadas neste trabalho (Capítulo 3 e Apêndice A) mostraram-se eficazes para a diminuição deste tipo de advertências. Além disso, a partir dos resultados da utilização prática da abordagem, a mesma pode ser estendida para considerar novas heurísticas que diminuam as chances da criação de advertências falso-positivas existentes em contextos específicos.

O tempo de análise da abordagem proposta não se mostrou sensível a características específicas do código-fonte, tais como quantidade de linhas de código, complexidade dos fluxos de dados e de controle ou complexidade ciclomática. Além disso, a abordagem apresenta vantagens que simplificam a implementação de uma instanciação da mesma, são elas: representação simplificada do código-fonte (ACD), heurísticas de baixa complexidade e fraco acoplamento entre os módulos. Algumas características observadas na avaliação da abordagem proposta também indicam uma possibilidade concreta de sua aplicação em um cenário real, tais como: i) pouca sensibilidade às características do código-fonte; ii) heurísticas de baixa complexidade com potencial de reduzir a criação de advertências falso-positivas; e iii) eficácia satisfatória e superior quando comparada à abordagem manual.

A principal limitação prática da abordagem diz respeito ao seu impacto em um cenário real de inspeção de procedimentos de banco de dados. A utilização da abordagem acarreta esforços subsequentes consideráveis, os quais envolvem etapas de: i) análise e interpretação as advertências; ii) implementação das advertências; iii) teste do impacto da aplicação das advertências; e iv) verificação da preservação do comportamento do código-fonte dos procedimentos após a implementação das advertências. É importante ressaltar que três destas etapas (ii, iii e iv) precisam ser realizadas independente da técnica (automática ou manual) aplicada para análise de procedimentos de banco de dados. Desse modo, ainda que

concentre-se apenas na etapa de análise e adicione mais uma etapa (i) ao processo em questão, a utilização da abordagem pode potencialmente acarretar em uma diminuição de custos, uma vez que a mesma apresenta um custo de análise baixo quando comparado com os custos de inspeções manuais (Tabelas 5.11 e 5.9). Estes resultados são também confirmados pelas respostas à pergunta *IV)* do questionário aplicado aos desenvolvedores do projeto MB (Tabela 5.12).

6.1 Trabalhos Relacionados

Nosso trabalho é relacionado com trabalhos que investigam diretrizes de eficiência e qualidade para procedimentos de banco de dados, que realizam análise de impacto de melhorias de eficiência na execução de procedimentos, ferramentas para análise automática de procedimentos e abordagens para análise dinâmica (perfilamento) de procedimentos de banco de dados. A seguir, relacionamos essas abordagens com a proposta neste trabalho.

Investigação de Diretrizes para Procedimentos. Em [9] [12] [13], os autores apresentam e discutem uma série de diretrizes de eficiência e qualidade para procedimentos de banco de dados. Em [12] [13], os autores discutem as consequências e o impacto da aplicação de diretrizes de eficiência na execução de procedimentos de banco de dados. Os autores de [9] discutem, com exemplos práticos de tarefas de programação de aplicações de banco de dados, como a manutenção de procedimentos de banco de dados pode ser facilitada com a aplicação de diretrizes de qualidade. Neste trabalho, são investigadas muitas das diretrizes de eficiência e qualidade investigadas em [9] [12] [13]. No entanto, além da discussão das diretrizes propriamente ditas, nós também criamos estratégias para a análise de conformidade das mesmas com procedimentos de banco de dados utilizando uma técnica de análise estática. Além disso, avaliamos de maneira prática a possibilidade de aplicação destas diretrizes no contexto de procedimentos de banco de dados de código-fonte aberto e industriais.

Análise de Desempenho de Procedimentos. Muitos trabalhos [5] [4] [12] [13] concentram-se em avaliar o impacto de melhorias de eficiência específicas na execução de procedimentos de banco de dados. Em [4], os autores discutem como a execução de comandos SQL pode ser otimizada utilizando processamentos equivalentes na forma de código procedimental. Os mesmos reportam resultados da otimização de um procedimento escrito

em T-SQL utilizado para processar dados do desempenho acadêmico de alunos. Em [5], os autores realizam uma investigação experimental do impacto da diretriz TRC_CTX em procedimentos escritos em PL/SQL. São discutidos o impacto no tempo de execução dos procedimentos analisados e quais fatores (número de campo das tabelas, tipos de dados dos campos e configurações globais do SGBD) podem afetar a execução do comando BULK COLLECT da linguagem PL/SQL.

Os autores de [12] [13] discutem resultados relacionados ao impacto de melhorias individuais de eficiência no código procedimental de procedimentos de banco de dados. Por exemplo, os autores investigam as seguintes diretrizes de eficiência também discutidas neste trabalho (Capítulo 4 e Apêndice A): TI_NAT, TN_NAT, PRI_COND, ORD_LOG, TRC_CTX, RED_COM, REM_VAR e REM_PAR.

Nosso trabalho é diferente dos realizados por [4] [5], pois, além de realizarmos análise de desempenho de melhorias procedimentais que não são cobertas por estes, nós investigamos a possibilidade prática da aplicação destas melhorias em cenários reais. Nosso trabalho estende os conduzidos por [12] [13] uma vez que, diferente da nossa análise de desempenho (Seção 5.2), os experimentos de desempenho conduzidos por estes trabalhos não realizaram repetições para medir a média do tempo de execução dos procedimentos. Acredita-se que esta característica limita bastante a generalidade destes trabalhos, uma vez que a execução de procedimentos de banco de dados pode sofrer influência de diversos fatores aleatórios, tais como processos em execução em paralelo pelo sistema operacional, dados presentes na memória *cache* do computador, dados presentes na memória *cache* do SGBD, esperas provenientes do algoritmo de escalonamento de CPU do sistema operacional, dentre outros.

Ferramentas para Análise Estática de Procedimentos. A ferramenta *SQL Enlight* [29] objetiva checar a conformidade de procedimentos escritos em T-SQL com aspectos como: convenção de nomes (*Naming*), *design*, desempenho e manutenção do código-fonte analisado. *SQL Enlight* especifica aproximadamente 70 regras pré-definidas para o desenvolvimento de procedimentos escritos em T-SQL.

A ferramenta *PL/SQL Sonar Source* [25] é capaz de analisar aproximadamente 120 regras pré-definidas para a criação de procedimentos PL/SQL, as quais envolvem eficiência, design, boas práticas, convenções de nomes e convenções de comentários do código-fonte. Além disso, as advertências reportadas pela ferramenta [25] podem ser visualizadas de forma

gráfica utilizando o módulo *Sonar Source Dashboard*.

A ferramenta *PLSQL Test Coverage* [26] utiliza técnicas de análise estática para reportar informações sobre a cobertura de testes escritos para procedimentos PL/SQL. A ferramenta reporta os resultados da análise de cobertura de forma gráfica, apontando com cores distintas no código-fonte as partes que são cobertas ou não pelos testes automáticos analisados.

As ferramentas disponibilizadas por [29] [25] [26] visam realizar a análise automática de código-fonte e testes de linguagens procedimentais de banco de dados específicas. Na prática, nosso trabalho tem um objetivo distinto dos propostos por [29] [25] [26], uma vez que o objetivo do mesmo é propor uma abordagem generalizável para a análise de procedimentos de banco de dados e avaliar esta abordagem no contexto de procedimentos reais. Além disso, as ferramentas criadas por [29] [25] [26] possuem licença comercial e os detalhes destas ferramentas relacionados à arquitetura, design e implementação das mesmas não são divulgados pelos fabricantes. Por outro lado, um dos focos do nosso trabalho é especificar de forma detalhada como os módulos da abordagem proposta podem ser modelados e implementados.

Perfilamento de Procedimentos. Outra possível maneira de identificar gargalos de desempenho em procedimentos de banco de dados é através de uma técnica denominada perfilamento. Nesta estratégia, são coletadas informações sobre a execução do código-fonte em questão e esses dados são sumarizados para permitir futuras análises. O pacote *DBMS_PROFILER* [27], por exemplo, é capaz de perfilar a execução de procedimentos escritos em PL/SQL. Analogamente, a ferramenta *SQL Server Profiler* [14] é capaz de perfilar a execução de procedimentos escritos em Transact-SQL. No entanto, o processo de perfilamento pode demorar demasiadamente nos casos onde muitos procedimentos de banco de dados precisam ser executados variando parâmetros de entrada e cargas de processamento. Além disso, após a etapa de perfilamento, os programadores ainda precisam investir um tempo substancial interpretando os dados provenientes das execuções e criando soluções para possíveis gargalos de desempenho detectados. Sendo assim, devido a limitações de tempo e recursos, nem sempre é viável utilizar técnicas de perfilamento no processo de ajuste de desempenho de procedimentos de banco de dados. Nosso trabalho difere das abordagens disponibilizadas por [14] [27], uma vez que a abordagem proposta baseia-se exclusivamente na análise estática de código-fonte. Além disso, nossa abordagem apresenta um custo in-

ferior para análise de procedimentos de banco de dados quando comparado com os custos atrelados ao perfilamento dinâmico dos mesmos.

6.2 Trabalhos Futuros

Em trabalhos futuros, é possível estender a abordagem proposta nos seguintes aspectos: utilização de outras técnicas para a diminuição de advertências falso-positivas, assistência nas tarefas subsequentes à etapa de análise, utilização efetiva dos metadados armazenados em banco de dados e facilitação da extensão das análises realizadas pela abordagem.

Diminuição de Advertências Falso-positivas. Além da aplicação de heurísticas e o recebimento de parâmetros configurados pelo usuário, outra possível técnica para diminuir as chances de criação de advertências falso-positivas é a utilização de dados da avaliação dos usuários em relação às advertências reportadas pela abordagem. Neste cenário, deve ser realizada uma análise histórica dos casos em que as advertências foram consideradas falso-positivas. Com base nesta análise, devem ser executados algoritmos que comparem a semelhança da advertência criada com alguma advertência avaliada como falso-positiva no histórico armazenado. Esta técnica pode tornar-se ainda mais efetiva caso as avaliações de diferentes usuários sejam armazenadas em uma base única por meio de dados coletados via internet. Neste contexto, pretende-se estender a abordagem para considerar os dados históricos das avaliações dos usuários na etapa de criação das advertências.

Tarefas Subsequentes à Etapa de Análise. Outra possibilidade de trabalho futuro é a extensão da abordagem para automatizar tarefas que são subsequentes (e necessárias) após a etapa de análise do código-fonte de procedimentos de banco de dados. Pretende-se investigar e criar técnicas que possam automatizar as tarefas de implementação das advertências reportadas, análise de impacto da aplicação das advertências e verificação da preservação do comportamento dos procedimentos após a aplicação das advertências. Acredita-se que a execução automática destas tarefas apresente um custo inferior ao custo de execução das mesmas por desenvolvedores. Além disso, utilizando uma abordagem automática, podem ser aplicados modelos de refatoramento formais que garantam a aplicação correta das advertências e a preservação semântica do comportamento dos procedimentos após estas aplicações; uma vez que estas tarefas são bastante propensas a erros quando executadas de forma manual.

Utilização de Metadados. Acredita-se que a abordagem proposta pode ser beneficiada em diversos aspectos caso os metadados armazenados nos bancos de dados sejam utilizados de forma mais efetiva. As seguintes informações podem ser exploradas para melhorar a completude da análise automática dos procedimentos: i) análise das configurações de memória do SGBD; ii) análise do fluxo de chamadas interprocedural dos procedimentos de banco de dados; e iii) análise dos índices presentes nos banco de dados. Neste contexto, outro trabalho futuro que pode ser realizado consiste em investigar que tipos de metadados são armazenados pelos SGBD's que permitem a criação de procedimentos de banco de dados e como estes metadados podem ser utilizados para beneficiar o funcionamento da abordagem proposta.

Extensibilidade das Análises. Analisadores estáticos automáticos limitam-se a examinar um conjunto pré-definido de regras/diretrizes nos artefatos de interesse. Da forma como foi arquitetada, uma instanciação da abordagem proposta exige que o usuário implemente novas análises utilizando uma linguagem de domínio específico. Nesse contexto, uma possibilidade para a realização de trabalhos futuros consiste em criar uma linguagem que permita ao usuário da abordagem especificar novas diretrizes para serem checadas automaticamente pela mesma. Na prática, a especificação das diretrizes criadas pelos usuários deve ser interpretada para o código-fonte de uma linguagem de programação de domínio específico e, em seguida, incorporada aos módulos *Efficiency Analyzer* ou *Quality Analyzer* da abordagem.

Bibliografia

[1] PostgreSQL online manual: http://www.postgresql.org.

[2] Samson Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[3] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, pages 86–, Washington, DC, USA, 1995. IEEE Computer Society.

[4] Tan Baohua and Zeng Ling. A performance optimization based on stored procedure in rdbs project. In *International Conference On Computer and Communication Technologies in Agriculture Engineering (CCTAE), 2010*, volume 3, pages 594 –597, june 2010.

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

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[7] Edsger W. Dijkstra. Software pioneers. chapter Go to statement considered harmful, pages 351–355. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[8] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2:17–23, May 2000.

[9] Steven Feuerstein. *Oracle pl/sql best practices, 2nd edition*. O'Reilly, second edition, 2007.

[10] Tor Guimaraes. Managing application program maintenance expenditures. *Communications of the ACM*, 26:739–746, October 1983.

[11] Ravindra Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. *Proceedings of the VLDB Endowment*, 1:1107–1123, August 2008.

[12] Timonthy Hall. *Oracle PL/SQL Tuning: Expert Secrets for High Performance Programming (Oracle in-Focus)*. Rampant TechPress, 2006.

[13] Guy Harrison. *Oracle SQL High-Performance Tuning, Second Edition*. Prentice Hall Professional Technical Reference, 2nd edition, 2000.

[14] Microsoft Inc. Sql server profiler: http://msdn.microsoft.com/en-us/library/ms181091.aspx.

[15] Microsoft Inc. Transact-sql server online reference: http://msdn.microsoft.com/pt-br/library/ms187926.aspx.

[16] J. Howard Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, pages 120–126, Washington, DC, USA, 1994. IEEE Computer Society.

[17] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1994.

[18] Meir M. Lehman and Juan F. Ramil. Rules and tools for software evolution planning and management. *Ann. Softw. Eng.*, 11:15–44, November 2001.

[19] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, January 1979.

[20] Edward S. Lowry and C. W. Medlock. Object code optimization. *Commun. ACM*, 12(1):13–22, January 1969.

[21] Dimas Nascimento, Tiago Massoni, and Carlos Pires. A static analysis-based approach to suggest improvements for stored procedures. In *16th Brazilian Symposium on Programming Languages (SBLP)*, 2012.

[22] Dimas Nascimento, Tiago Massoni, and Carlos Pires. Uma abordagem baseada em anlise estática para sugerir melhorias em procedimentos armazenados em banco de dados. In *II Workshop de Teses e Dissertasções do CBSoft (Congresso Brasileiro de Software)*, 2012.

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

[24] Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Static program analysis via 3-valued logic. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 401–404. Springer Berlin / Heidelberg, 2004.

[25] SonarSource S.A. Sonar for pl/sql: http://www.sonarsource.com/products/plugins/.

[26] Inc. Semantic Designs. Plsql test coverage: http://www.semanticdesigns.com/products/testcoverage/plsqltestcoverage.html, 2005.

[27] Eric Belden Sheila Moore. *Oracle Database PL/SQL Language Reference, 11g Release 2 (11.2)*. Oracle, 2009.

[28] Ian Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 9. edition, 2010.

[29] Ubisoft. Sql enlight for t-sql: http://www.ubitsoft.com/index.php, 2007.

[30] Michal Young and Mauro Pezze. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2005.

Apêndice A

Diretrizes de Eficiência e Qualidade

Nesta seção, são descritas as demais diretrizes de eficiência e qualidade para o desenvolvimento de procedimentos de banco de dados investigadas neste trabalho.

A.1 Diretrizes de Eficiência

Tipos de Dados Numéricos Nativos

Assim como acontece com tipos de dados inteiros nativos, algumas linguagens de programação de banco de dados também disponibilizam tipos de dados numéricos nativos que permitem a realização de operações aritméticas de maneira mais eficiente e requerem menos espaço de armazenamento [9] [12] [13] [27]. Uma possível diretriz de eficiência para ser seguida no desenvolvimento de procedimentos armazenados em banco de dados é a utilização de tipos de dados numéricos nativos para variáveis e parâmetros que são utilizados para a realização de operações aritméticas. Esta diretriz é referenciada utilizando o Código de Diretriz (C_D) TN_NAT. As condições necessárias para a checagem estática da diretriz TN_NAT são mostradas na Fórmula Lógica 9.

- **Fórmula Lógica 9**: [*usage_of_numeric_data_type(identifier)*] $\wedge$ [*used_for_arithmetic_operation(identifier)*] $\wedge$ <*executed_at_least_n_times(arithmetic_operation)*> $\wedge$ $\neg$ (<*overflow(identifier)*> $\vee$ <*underflow(identifier)*>)

As condições dispostas na Fórmula Lógica 9 são detalhadas a seguir:

- *usage_of_numeric_type(identifier)*: esta condição visa verificar se o identificador (de variável ou parâmetro) analisado utiliza o tipo de dados NUMERIC (ou algum tipo de dados numérico equivalente que não possua uma representação interna em baixo nível);

- *used_for_arithmetic_operation(identifier)*: esta condição visa checar se o identificador (de variável ou parâmetro) analisado é utilizado como fator no cálculo de alguma operação aritmética;

- *executed_at_least_n_times(arithmetic_operation)*: esta condição tem como objetivo realizar checagens estáticas para verificar se a operação aritmética detectada na condição *used_for_arithmetic_operation(identifier)* é executada pelo menos *n* (parâmetro definido pelo usuário da abordagem) vezes pelo procedimento de banco de dados. As heurísticas utilizadas para a checagem desta condição são discutidas na explanação da Fórmula Lógica 1;

- *overflow(identifier)*: esta condição visa verificar se, caso o resultado de alguma operação aritmética seja atribuído a uma variável ou parâmetro (do tipo resultado ou valor/-resultado) do tipo de dados NUMERIC (ou equivalente), a mudança do tipo de dados desta variável ou parâmetro para um tipo de dados numérico nativo não causará um erro do tipo *overflow* em tempo de execução. Esta condição é indecidível de ser analisada de forma estática. Isto ocorre porque a operação aritmética pode, por exemplo, envolver como fator algum parâmetro de entrada do procedimento ou valor acessado em um banco de dados. Sendo assim, é possível realizar esta checagem utilizando a seguinte heurística: TN_NAT.hi) nos casos em que são feitas atribuições de valor fixo em uma variável (exemplos:*n := 100.000*, *n := 50.000 + 50.000* e *n := constant_value*), verificar estaticamente se o valor atribuído pode causar um erro do tipo *overflow* na variável ou parâmetro cujo tipo de dados é numérico nativo;

- *underflow(identifier)*: esta condição visa verificar se, caso o resultado de alguma operação aritmética que é atribuído a uma variável ou parâmetro (do tipo resultado ou valor/resultado) do tipo de dados NUMERIC (ou equivalente), a mudança do tipo de dados desta variável ou parâmetro para um tipo de dados numérico nativo não causará

um erro do tipo *underflow* em tempo de execução. Esta condição também é indecidível de ser analisada de forma estática. Isto ocorre porque a operação aritmética também pode, por exemplo, envolver como fator algum parâmetro de entrada do procedimento ou um valor acessado em um banco de dados. Sendo assim, é possível realizar esta checagem utilizando a seguinte heurística: TN_NAT.hii) nos casos em que são feitas atribuições de valor fixo em uma variável (exemplos: *n := 0.0001*, *n := 0.00005 + 0.00005* e *n := constant_value*), verificar estaticamente se o valor atribuído pode causar um erro do tipo *underflow* na variável ou parâmetro cujo tipo de dados é um numérico nativo.

A.1.1 Tipos de Dados Restritos

Algumas linguagens procedimentais de banco de dados disponibilizam tipos de dados restritos que exigem checagens adicionais em tempo de execução para garantir que as restrições dos tipos de dados sejam obedecidas. Caso ocorra alguma violação nas restrições de valores permitidos a estes tipos de dados, uma exceção é lançada durante a execução do procedimento de banco de dados. Por exemplo, a linguagem PL/SQL disponibiliza os seguintes tipos de dados restritos: NATURAL, NATURALN, POSITIVE e POSITIVEN.

Uma possível diretriz de eficiência é a não utilização de tipos de dados restritos, uma vez que repetidas checagens de valores atribuídos a variáveis ou parâmetros (com tipos de dados restritos) em tempo de execução podem causar um atraso significativo na execução dos procedimentos. Esta diretriz é referenciada utilizando o Código de Diretriz (C_D) TD_RES. As condições necessárias para a checagem estática da diretriz TD_RES são mostradas na Fórmula Lógica 10.

- **Fórmula Lógica 10**: [*usage_of_constrained_data_type(identifier)*] $\wedge$ <*used_for_assignment(identifier)*> $\wedge$ <*assignment_executed_at_least_n_times(identifier)*>

As condições dispostas na Fórmula Lógica 10 são detalhadas a seguir:

- *usage_of_constrained_data_type(identifier)*: esta condição visa verificar se o identificador (de variável ou parâmetro) analisado utiliza algum tipo de dados restrito.

- *used_for_assignment(identifier)*: Esta condição visa checar se o identificador (de variável ou parâmetro) analisado é utilizado na parte esquerda de algum comando de atribuição presente no procedimento analisado. Esta verificação é importante uma vez que, caso o identificador utilize um tipo de dados restrito não seja usado para a realização de uma atribuição, não é necessário realizar checagens em tempo de execução do procedimento para garantir que as restrições do tipo de dados sejam obedecidas;

- *assignment_executed_at_least_n_times(identifier)*: esta condição tem como objetivo realizar checagens estáticas para verificar se a atribuição detectada na condição <*used_for_assignment(identifier)*> é executada pelo menos *n* (parâmetro definido pelo usuário da abordagem) vezes pelo procedimento de banco de dados. As heurísticas utilizadas para a checagem desta condição são discutidas na explanação da Fórmula Lógica 1.

A.1.2 Otimização de Expressões Lógicas

Em alguns casos, quando expressões lógicas são compostas por diversos termos separados por operadores AND e OR, é possível determinar o resultado da avaliação da expressão como um todo sem que necessariamente cada um dos termos da expressão seja avaliado. Para tal, são utilizadas as seguintes regras básicas da lógica de primeira ordem:

- FALSE AND (*any_logic_expression*) = FALSE

- TRUE OR (*any_logic_expression*) = TRUE

Algumas linguagens de programação de banco de dados (por exemplo, PL/SQL e Transact-SQL) utilizam estas regras em tempo de execução para economizar a avaliação de partes de expressões lógicas. Esta característica é chamada de avaliação de circuito rápido [9] [12] [13] [27]. Nas ocasiões em que as expressões lógicas envolvem chamadas a funções que necessitem de um tempo significativo no contexto da aplicação para serem avaliadas (por exemplo, executando comandos de repetição ou acarretando esperas devido ao acesso a bancos de dados), a ordenação eficiente dos termos das expressões lógicas pode otimizar o tempo de execução de um procedimento de banco de dados.

A determinação da ordem ótima dos termos de uma expressão lógica depende de duas variáveis: custo de avaliar o termo (CA) e a probabilidade do resultado da avaliação do termo permitir a avaliação da expressão lógica como um todo (PA). Na prática, a análise de apenas uma destas duas variáveis não é suficiente para decidir a ordenação ótima dos termos de uma expressão lógica. Por exemplo, considerando a expressão lógica:

- L1: (A $\vee$ B)

Em uma situação onde: CA(A) = 70, PA(A) = 99%, CA(B) = 30 e PA(B) = 1%, a ordem dos termos de L1 na forma (A $\vee$ B) é mais vantajosa, pois, ainda que apresente um custo de avaliação mais elevado, a avaliação do termo A apresenta uma chance muito maior de decidir a expressão. Neste caso, a variável PA foi utilizada para ordenar os termos. Em outra situação, onde: CA(A) = 99, PA(A) = 70%, CA(B) = 1 e PA(B) = 30%, a ordem dos termos de L1 na forma (B $\vee$ A) é mais vantajosa, pois, ainda que ofereça uma probabilidade mais baixa de permitir a avaliação da expressão lógica como um todo, o termo B oferece um custo muito mais baixo que o termo A e causa pouco atraso na tentativa de avaliar a expressão. Neste caso, a variável CA foi utilizada para ordenar os termos.

Neste contexto, uma possível diretriz de eficiência para ser seguida no desenvolvimento de procedimentos armazenados em banco de dados é a ordenação dos fatores de expressões lógicas no intuito de minimizar o tempo e o esforço de avaliação da expressão lógica como um todo utilizando a estratégia de avaliação de circuito rápido. Esta diretriz é referenciada utilizando o Código de Diretriz (C_D) ORD_LOG. As condições necessárias para a checagem estática da diretriz ORD_LOG são mostradas na Fórmula Lógica 11.

- **Fórmula Lógica 11**: <*is_logic(expression)*> $\wedge$ <*is_complex(expression)*> $\wedge$ <*executed_at_least_n_times(expression)*> $\wedge \neg$ <*optimal_order(expression)*>

As condições dispostas na Fórmula Lógica 11 são detalhadas a seguir:

- *is_logic(expression)*: esta condição visa checar se a expressão analisada representa uma expressão lógica;
- *is_complex(expression)*: esta condição visa verificar se a expressão lógica existente no procedimento de banco de dados é complexa o bastante para permitir uma ordenação

de fatores que minimize o esforço para avaliação da expressão como um todo. Para esta verificação, as seguintes heurísticas podem ser utilizadas: ORD_LOG.hi) verificar estaticamente se a expressão lógica contém pelo menos t (parâmetro configurado pelo usuário da abordagem) termos; e ORD_LOG.hii) checar se a expressão lógica realiza pelo menos i (parâmetro configurado pelo usuário) chamadas a funções. Isto porque, no geral, a avaliação de funções (as quais podem realizar acessos a bancos de dados, executar comandos de repetição, dentre outros) tende a representar um atraso e esforço maiores do que a avaliação de simples expressões relacionais;

- *executed_at_least_n_times(expression)*: esta condição tem como objetivo realizar checagens estáticas para verificar se a expressão lógica detectada na condição <*is_logic(expression)*> é executada pelo menos n (parâmetro a ser definido pelo usuário) vezes pelo procedimento de banco de dados. As heurísticas utilizadas para a checagem desta condição são discutidas na explanação da Fórmula Lógica 1;

- *optimal_order(lexpression)*: esta condição visa checar se a expressão lógica detectada na condição <*is_logic(expression)*> já se encontra na ordem ótima dos termos. De fato, esta condição é indecidível utilizando análise estática por dois motivos: i) a expressão lógica pode manipular valores de parâmetros de entrada do procedimento ou acessar valores em um banco de dados e, desta forma, impedir a previsão do comportamento de avaliação dos fatores em tempo de execução; e ii) como discutido no exemplo da fórmula lógica **L1** desta seção, a análise da ordem ótima dos fatores de uma expressão lógica depende de duas variáveis (custo do fator e probabilidade do valor do fator avaliar a expressão) e, na prática, pode depender de conhecimentos prévios sobre quais são os parâmetros de entrada mais frequentemente recebidos pelo procedimento ou valores mais comumente acessados em um banco de dados. Sendo assim, é mais indicado que o desenvolvedor do procedimento utilize informações sobre o contexto da aplicação para tentar ordenar os termos da expressão lógica. No entanto, é possível utilizar uma heurística simples para dar indícios de que a expressão não esteja na ordem ótima, a qual consiste na seguinte verificação: ORD_LOG.hiii) a expressão lógica avalia pelo menos j termos que realizam chamadas a funções antes de avaliar k termos que não realizam chamadas a funções, onde j e k são parâmetros configurados pelo usuário da

abordagem.

A.1.3 Otimização de Comandos Condicionais

Nos casos em que um procedimento de banco de dados executa comandos condicionais cujos ramos de fluxo de execução são aninhados e mutuamente exclusivos (exemplos: comandos *case* ou comandos *if-elsif-else* aninhados), priorizar (na ordem de execução dos ramos) a avaliação das condições que acarretam em um menor custo e/ou maior chance de serem avaliados como 'verdadeiro' pode otimizar o tempo de execução do procedimento [9] [12] [13]. Esta priorização pode ser utilizada como uma diretriz no desenvolvimento de procedimentos de banco de dados e é referenciada utilizando o Código de Diretriz PRI_COND.

Assim como acontece com a diretriz ORD_LOG, a priorização dos ramos de um comando condicional também depende de duas variáveis: custo de avaliar a condição do ramo (CR) e a probabilidade da condição do ramo ser avaliada como verdadeiro (PG). Além disso, usando um raciocínio semelhante ao utilizado no exemplo da Fórmula Lógica **L1**, a análise de apenas uma das duas variáveis (CR ou PG) não é suficiente para decidir a priorização adequada dos ramos de um comando condicional. As condições necessárias para a checagem estática da diretriz PRI_COND são mostradas na Fórmula Lógica 12.

- **Fórmula Lógica 12**: [*is_complex(nested_conditional_command*] $\wedge$ <*executed_at_least_n_times(conditional_command)*> $\wedge$ $\neg$ <*optimal_order(conditional_command)*>

As condições dispostas na Fórmula Lógica 12 são detalhadas a seguir:

- *is_complex(nested_conditional_command)*: esta condição visa verificar se o comando condicional cujos ramos são aninhados e mutuamente excludentes existente no procedimento de banco de dados é complexo o bastante para permitir uma ordenação eficaz. Para esta verificação, as seguintes heurísticas podem ser utilizadas: PDI_COND.hi) verificar estaticamente se o comando condicional contém pelo menos g (parâmetro configurado pelo usuário da abordagem) ramos aninhados e excludentes; e PDI_COND.hii) checar se as condições dos ramos do comando condicional aninhado realizam pelo menos i (parâmetro configurado pelo usuário) chamadas a funções. Isto

porque, no geral, ramos cujas condições realizam chamadas à funções (as quais podem acarretar esperas relacionadas a acessos a bancos de dados, execução de comandos de repetição, dentre outros) tendem a representar um atraso e esforço maiores em suas avaliações do que condições de ramos que não envolvem chamadas à funções;

- *executed_at_least_n_times(nested_conditional_command)*: esta condição tem como objetivo realizar checagens estáticas para verificar se o comando condicional detectado na condição *is_complex(nested_conditional_command)* é executado pelo menos *n* (parâmetro a ser definido pelo usuário) vezes na pelo procedimento de banco de dados. As heurísticas utilizadas para a checagem desta condição são discutidas na explanação da Fórmula Lógica 1;

- *optimal_order(nested_conditional_command)*: esta condição visa checar se o comando condicional detectado na condição *is_complex(nested_conditional_command)* já realiza a priorização ótima dos ramos. De fato, esta condição é indecidível utilizando análise estática por dois motivos: i) as condições lógicas dos ramos podem manipular valores de parâmetros de entrada do procedimento ou valores provenientes de bancos de dados e, desta forma, impedir a previsão do comportamento do fluxo de execução do procedimento em tempo de execução; e ii) semelhante ao exemplo discutido na Seção A.1.2 referente à fórmula lógica **L1**, a análise da priorização ideal dos ramos do comando condicional depende de duas variáveis (CR e PG) e, na prática, pode depender de conhecimentos prévios sobre quais os parâmetros de entrada mais frequentemente recebidos pelo procedimento ou valores mais frequentemente acessados em bancos de dados. Desse modo, é mais indicado que o desenvolvedor do procedimento utilize informações sobre o contexto da aplicação para tentar priorizar os ramos do comando condicional. No entanto, é possível utilizar uma heurística simples para dar indícios de que os ramos do comando condicional aninhado não estejam priorizados na ordem ótima, a qual consiste na seguinte verificação: PRI_COND.hi) a estrutura do comando condicional apresenta pelo menos j ramos, cujas condições realizam chamadas a funções, com maior prioridade no fluxo de execução do código e pelo menos k ramos, que não realizam chamadas a funções, com menor prioridade na execução do código. Neste caso, j e k são parâmetros configurados pelo usuário da abordagem.

A.1.4 Redução de Cópia de Parâmetros

Quando parâmetros do tipo cópia de valor ou cópia de valor/resultado são utilizados pelo procedimento, é necessário que seja realizada uma cópia dos valores dos mesmos, em tempo de execução, cada vez que uma chamada ao procedimento é executada [12] [13] [27]. Nos casos em que o procedimento é muito frequentemente chamado, o tempo de cópia dos parâmetros pode gerar esperas consideráveis para o funcionamento da aplicação. Sendo assim, uma possível diretriz de eficiência no desenvolvimento de procedimento de banco de dados é a redução de operações de cópias de parâmetros nas chamadas aos procedimentos. Para tal, os parâmetros devem ser passados nas assinaturas dos procedimentos por referência, ao invés de por cópia. Esta diretriz é referenciada utilizando o Código de Diretriz COP_PAR. As condições necessárias para a checagem estática da diretriz COP_PAR são mostradas na Fórmula Lógica 13.

- **Fórmula Lógica 13**: [*is_copyed(parameter)*] $\wedge$ <*is_large(parameter)*> $\wedge$ <*called_at_least_n_times(procedure)*> $\wedge \neg$ <*used_for_assignment(parameter)*>

As condições dispostas na Fórmula Lógica 13 são detalhadas a seguir:

- *is_copyed(parameter)*: esta condição visa checar se o parâmetro presente na assinatura do procedimento de banco de dados é passado como cópia de valor ou cópia de valor/resultado.

- *is_large(parameter)*: esta condição visa checar se o parâmetro analisado apresenta um tamanho grande o suficiente para acarretar em uma espera significativa quando o valor do mesmo é copiado na chamada do procedimento. Uma vez que o limiar que define um tempo de espera significativo depende do contexto da aplicação, a definição da classificação de um parâmetro como "grande"deve ser preferencialmente configurada pelo usuário da abordagem. Na prática, as seguintes heurísticas podem ser utilizadas para a checagem desta condição: COP_PAR.hi) verificar estaticamente se o tipo de dados do parâmetro é maior ou igual a um tamanho k configurado pelo usuário da abordagem; e COP_PAR.hii) verificar se o tipo de dados do parâmetro representa um tipo composto, por exemplo, uma lista, mapa ou *array*;

- *called_at_least_n_times(procedure)*: esta condição visa checar se o procedimento de banco de dados cujas condições *is_copyed(parameter)* e <*is_large(parameter)*> foram detectadas é executado pelo menos n vezes por algum comando de repetição de outro procedimento. Na prática, esta análise é realizada checando estaticamente as chamadas interprocedurais. Adicionalmente, podem ser utilizadas as heurísticas discutidas na explanação da Fórmula Lógica 1 na tentativa de checar estaticamente a quantidade de execuções dos comandos de repetição;

- *used_for_assignment(parameter)*: esta condição tem como objetivo verificar, caso o parâmetro passado por cópia for do tipo valor/resultado, se o mesmo é utilizado na parte esquerda de algum comando de atribuição do procedimento de banco de dados. Em caso afirmativo, pode ser utilizada uma das seguintes estratégias: i) Não reportar (retornando "falso"na condição *used_for_assignment(parameter)*) no Resumo da Análise a possibilidade da aplicação da diretriz COP_PAR no parâmetro analisado, uma vez que uma atribuição neste parâmetro gera um efeito colateral no valor do parâmetro fora do escopo do procedimento; ou ii) Reportar (retornando "verdadeiro"na condição *used_for_assignment(parameter)*) no Resumo da Análise a possibilidade de aplicação diretriz COP_PAR, mas alertar ao usuário que a mudança no tipo de passagem do parâmetro (de cópia para referência) irá acarretar um efeito colateral.

A.1.5 Otimização do Tamanho de Tipos de Dados

Caso variáveis ou parâmetros sejam declarados utilizando tipos de dados cujas escalas de valores sejam significativamente maiores do que a escala necessária para o armazenamento dos valores utilizados nas variáveis ou parâmetros do procedimento de banco de dados, a execução do procedimento irá consumir mais memória do que a quantidade realmente necessária para o armazenamento dos valores temporários por ele manipulado. Por exemplo, a linguagem Transact-SQL disponibiliza tipos de dados para o armazenamento de valores dentro de uma faixa reduzida, tais como: *smallint*, *smalldatetime* e *bit*. A utilização destes tipos de dados, além de prover economia de memória, pode beneficiar o desempenho da aplicação, uma vez que: i) quanto menor o tamanho do tipo de dados, menor é a quantidade de dados lidos e, de um modo geral, menos tempo para a execução de operações de leitura

é necessário; e ii) tipos de dados menores acarretarão em um menor tráfego de dados pelos canais de comunicação.

Em algumas linguagens de programação de banco de dados [12] [27], aumentar o tamanho de tipos de dados específicos pode economizar memória, uma vez que a partir de certo tamanho, apenas a quantidade de fato utilizada para o armazenamento do valor atribuído é alocada na memória. Este fenômeno acontece, por exemplo, com o tipo de dados VARCHAR2 da linguagem PL/SQL.

Uma possível diretriz de eficiência é a otimização do tamanho dos tipos de dados utilizados no procedimento, seja aumentando ou diminuindo o tamanho dos mesmos. Esta diretriz é referenciada utilizando o C_D OTI_TDA. As condições necessárias para a checagem estática da diretriz OTI_TDA são mostradas na Fórmula Lógica 14.

- **Fórmula Lógica 14**: ¬ <*optimal_size(identifier)*>

As condições dispostas na Fórmula Lógica 14 são detalhadas a seguir:

- *optimal_size(identifier)*: esta condição visa checar se o tipo de dados utilizado pelo identificador (de variável ou parâmetro) já não se encontra no tamanho ideal. De fato, esta checagem é indecidível de forma estática, uma vez que os valores atribuídos a uma variável ou parâmetro (do tipo resultado ou valor/resultado) podem depender do valor de um parâmetro de entrada ou de uma fonte externa (por exemplo, um valor proveniente de um banco de dados). Desse modo, é possível utilizar as seguintes heurísticas para realizar esta checagem: OTI_TDA.hi) verificar se o parâmetro ou variável é utilizado apenas na parte esquerda de atribuições de valores fixos, por exemplo: (pi := 3.14) ou pi := *constant_value*. Nestes casos, pode ser verificado estaticamente se a faixa de valores do tipo de dados do parâmetro ou variável excede o tamanho necessário para armazenar os valores atribuídos; e OTI_TDA.hii) Se o tipo de dados utilizado pela variável ou parâmetro possui a característica que permite que tamanhos maiores acarretem na alocação ótima de memória, deve ser verificado se o tamanho do tipo de dados é maior ou igual ao tamanho mínimo para que a alocação otimizada de memória seja realizada pelo interpretador da linguagem.

A.1.6 Declarações Repetidas

A declaração de variáveis, tipos de dados, cursores e subprocedimentos dentro do escopo de comandos de repetição representa uma operação bastante ineficiente, uma vez que é necessário realocar toda a memória das declarações em cada uma das iterações do comando de repetição. Desse modo, uma possível diretriz de eficiência é a remoção de declarações do escopo de comandos de repetição. Esta diretriz é referenciada utilizando o C_D DEC_REP. As condições necessárias para a checagem estática da diretriz DEC_REP são mostradas na Fórmula Lógica 15.

- **Fórmula Lógica 15**: [*is_declaration(statement)*] $\wedge$ [*dominated_by_loop(statement)*]

As condições dispostas na Fórmula Lógica 15 são detalhadas a seguir:

- *is_declaration(statement)*: esta condição visa checar se o comando analisado é representa um comando de declaração na parte executável do procedimento de banco de dados;
- *dominated_by_loop(statement)*: esta condição visa checar o comando executado é dominado estritamente por algum comando de repetição na estrutura hierárquica da ACD que representa o procedimento de banco de dados.

A.1.7 Utilização de Funções Nativas

Tentativas de re-implementação de funções nativas existentes em linguagens de programação de banco de dados muito frequentemente apresentam desempenho inferior às próprias funções nativas. Isto porque, é comum que funções nativas sejam implementadas utilizando operações de baixo nível que não podem ser utilizadas diretamente nas linguagens de programação [12]. Desse modo, uma possível diretriz de eficiência é dar preferência, quando possível, à utilização de funções nativas disponíveis na linguagem de programação. Esta diretriz é referenciada utilizando o C_D FUN_NAT. As condições necessárias para a checagem estática da diretriz FUN_NAT são mostradas na Fórmula Lógica 16.

- **Fórmula Lógica 16**: [*is_function(procedure)*] <*similar_to_native(procedure)*> $\wedge$ [*clone_implementation_of_native_function(procedure)*]

As condições dispostas na Fórmula Lógica 16 e a estratégia utilizada para a detecção automática das mesmas utilizando análise estática são detalhadas a seguir:

- *is_function(procedure)*: esta condição objetiva checar se o procedimento de banco de dados analisado é do tipo função;
- *similar_to_native(procedure)*: esta condição visa checar se o procedimento (do tipo função) armazenado no banco de dados analisado é semelhante à alguma função nativa disponível na linguagem de programação. Na prática, podem ser utilizadas as seguintes heurísticas para realizar esta checagem: FUN_NAT.hi) Para cada função nativa disponível na linguagem de programação, verificar se a distância entre as palavras dos nomes da função analisada e da função nativa é menor do que um limiar de tamanho k (definido pelo usuário da abordagem); ii) Para cada função nativa disponível na linguagem de programação, verificar se os tipos de dados dos parâmetros da função nativa e do procedimento (do tipo função) analisado são equivalentes;
- *clone_implementation_of_native_function(procedure)*: esta condição tem como objetivo verificar a semelhança semântica entre o procedimento (do tipo função) analisado e as funções nativas disponíveis na linguagem de programação. Na prática, deve ser utilizado algum algoritmo para detecção de clones de implementação [3] [16] para verificar se a semântica da implementação do procedimento (do tipo função) de banco de dados analisado é equivalente à semântica de alguma função nativa da linguagem de programação.

A.1.8 Variáveis não Utilizadas

A declaração de variáveis não utilizadas em procedimentos de banco de dados representa sempre um desperdício de memória. Além disso, este fato aumenta o esforço dos desenvolvedores no entendimento do código, uma vez que os mesmos necessitam analisar todo o código do procedimento para identificar a não utilização de variáveis. Nesse contexto, uma possível diretriz de eficiência é a remoção de variáveis não utilizadas. Esta diretriz é referenciada utilizando o C_D REM_VAR. As condições necessárias para a checagem estática da diretriz REM_VAR são mostradas na Fórmula Lógica 17.

- **Fórmula Lógica 17**: ¬ [*is_referenced(variable)*]

As condições dispostas na Fórmula Lógica 17 são detalhadas a seguir:

- *is_referenced(variable)*: esta condição tem como objetivo verificar se a variável declarada no procedimento é referenciada pelo menos uma vez pelo mesmo. Esta verificação é decidível estaticamente e pode ser implementada checando as referências aos identificadores realizadas por todos os comandos e expressões do procedimento.

A.1.9 Parâmetros não Utilizados

Parâmetros presentes na assinatura de procedimentos mas não utilizados são indesejáveis, pois acarretam em i) tempo extra de cópia para parâmetros deste tipo e ii) um elemento adicional para entendimento do procedimento. Nesse contexto, uma possível diretriz de eficiência é a remoção de parâmetros não utilizados. Esta diretriz é referenciada utilizando o C_D REM_PAR. As condições necessárias para a checagem estática da diretriz REM_PAR são mostradas na Fórmula Lógica 18.

- **Fórmula Lógica 18**: ¬ [*is_referenced(parameter)*]

As condições dispostas na Fórmula Lógica 18 são detalhadas a seguir:

- *is_referenced(parameter)*: esta condição tem como objetivo verificar se o parâmetro presente na assinatura do procedimento é referenciado pelo menos uma vez pelo procedimento. Esta verificação é decidível estaticamente e pode ser implementada checando as referências aos identificadores realizadas por todos os comandos e expressões do procedimento.

A.2 Diretrizes de Qualidade

A.2.1 Parâmetros do Tipo Resultado em Funções

A utilização de parâmetros do tipo resultado ou valor/resultado em procedimentos (do tipo função) de banco de dados representa uma má prática de programação. Isto porque, por

definição, uma função deve calcular e retornar um único valor. Se valores de parâmetros do tipo resultado ou valor/resultado são modificados por uma função, esta função passa a causar um efeito colateral na lógica do software como um todo. Em geral, entender um programa que realiza efeitos colaterais requer conhecimento sobre seus estados e contextos e, por isso, torna-se mais difícil compreendê-lo, testá-lo e depurá-lo [2].

Neste contexto, uma possível diretriz de qualidade de código é a não utilização de parâmetros do tipo resultado ou valor/resultado em procedimentos (do tipo função). Esta diretriz é referenciada utilizando o C_D EFT_COL_FUN. As condições necessárias para a checagem estática da diretriz EFT_COL_FUN são mostradas na Fórmula Lógica 19.

- **Fórmula Lógica 19**: [*is_function(procedure)*] $\wedge$ ([*is_result(parameter)*] $\vee$ [*is_value_result(parameter)*])

As condições dispostas na Fórmula Lógica 19 são detalhadas a seguir:

- *is_function(procedure)*: esta condição visa verificar se o procedimento de banco de dados analisado é do tipo função;
- *is_result(parameter)*: esta condição tem como objetivo verificar se o parâmetro presente na assinatura do procedimento é do tipo resultado;
- *is_value_result(parameter)*: esta condição tem como objetivo verificar se o parâmetro presente na assinatura do procedimento é do tipo valor/resultado;

A.2.2 Notação para Chamadas a Procedimento

Em geral, linguagens procedimentais de banco de dados disponibilizam duas notações para realizar chamadas para passagem de parâmetros a procedimentos de banco de dados: notação posicional e notação nomeada. Na primeira notação, os parâmetros são passados para o procedimento de acordo com a ordem que aparecem na chamada. Na segunda notação, os nomes dos parâmetros são explicitamente associados aos valores na chamada ao procedimento. Por exemplo, supondo a existência do procedimento *fire_employee(employee_id int)* armazenado no banco de dados, o qual recebe como parâmetro o identificador (*employee_id*) do tipo de dados inteiro. A chamada deste procedimento utilizando a sintaxe da linguagem Transact-SQL pode ser realizada das seguintes maneiras:

- Notação posicional: EXEC proc_schema.fire_employee 115;
- Notação nomeada: EXEC proc_schema.fire_employee @employee_id = 115;

Utilizando a sintaxe da linguagem PL/SQL, a chamada do procedimento *fire_employee* pode ser realizada das seguintes maneiras:

- Notação posicional: fire_employee(115);
- Notação nomeada: fire_employee(employee_id => 115);

Neste contexto, a utilização da notação nomeada facilita a legibilidade e manutenção de procedimentos de banco de dados pelos seguintes motivos: i) é facilitada a interpretação dos valores que são passados para o procedimento chamado sem que, necessariamente, o desenvolvedor precise checar manualmente a assinatura do procedimento; ii) caso ocorra alguma mudança na assinatura do procedimento chamado na qual a ordem dos parâmetros seja alterada, a parte do código que realiza chamadas a este procedimento utilizando a notação nomeada permanecerá funcionado sem a necessidade de atualizações; e iii) caso sejam adicionados parâmetros de entrada opcionais (i.e., que possuem valor padrão definido) ao procedimento chamado, as chamadas a este procedimento utilizando a notação posicional também permanecerão funcionado sem a necessidade de atualizações.

Neste contexto, uma possível diretriz de qualidade de código para o desenvolvimento de procedimentos de banco de dados é a utilização da notação nomeada ao invés da notação posicional para realizar chamadas a procedimentos de banco de dados [9] [12]. Esta diretriz é referenciada utilizando o código de diretriz NOT_NMD. As condições necessárias para a checagem estática da diretriz NOT_NMD são mostradas na Fórmula Lógica 20.

- **Fórmula Lógica 20**: $(([\textit{is\_procedure\_call(statement)}] \wedge [\textit{uses\_positional\_notation(statement)}])) \vee (([\textit{is\_function\_call(expression)}] \wedge [\textit{uses\_positional\_notation(expression)}]))$

As condições dispostas na Fórmula Lógica 20 são detalhadas a seguir:

- *is_procedure_call(statement)*: esta condição visa checar a se o comando analisado representa uma chamada a um procedimento;

- *uses_positional_notation(statement)*: esta condição visa checar se a chamada ao procedimento detectado na condição *is_procedure_call(statement)* realiza a passagem de pelo menos um parâmetro utilizando a notação posicional.

- *is_function_call(expression)*: esta condição visa checar a se a expressão analisada representa uma chamada a uma função;

- *uses_positional_notation(expression)*: esta condição visa checar se a chamada a função detectada na condição *is_function_call(expression)* realiza a passagem de pelo menos um parâmetro utilizando a notação posicional.

A.2.3 Desvios Incondicionais

A utilização de comandos que causam desvios incondicionais no fluxo de controle do programa (comandos *GO TO*) em um procedimento de banco de dados pode dificultar consideravelmente o entendimento do procedimento pelos desenvolvedores. Além disso, a utilização de desvios incondicionais deve ser evitada em linguagens de programação de alto nível uma vez que estes comandos dificultam a tarefa de verificação de corretude dos programas, especialmente aqueles que envolvem a execução de comandos de repetição [7]. Nesse contexto, uma possível diretriz de qualidade de código é a não utilização de desvios incondicionais. Esta diretriz é referenciada utilizando o C_D EXEC_GOTO. As condições necessárias para a checagem estática da diretriz EXEC_GOTO são mostradas na Fórmula Lógica 21.

- **Fórmula Lógica 21**: [*is_goto(statement)*]

As condições dispostas na Fórmula Lógica 21 são detalhadas a seguir:

- *is_goto(statement)*: esta condição tem como objetivo verificar se o comando analisado representa um desvio incondicional no fluxo de execução do procedimento de banco de dados.

A.2.4 Comandos de Escape

Algumas linguagens de programação de banco de dados (ex.: PL/SQL, T-SQL e PL/pgSQL) permitem que a lógica de continuação das iterações de comandos de repetições seja definida

por comandos de escape, tais como CONTINUE e EXIT. Em alguns casos, quando a lógica de repetição do laço é consideravelmente complexa ou envolve vários comandos de escape, a utilização de comandos de escape podem aumentar consideravelmente a complexidade do fluxo de controle do procedimento de banco de dados e, consequentemente, dificultar o entendimento do mesmo pelos desenvolvedores [9]. Uma vez que, por definição, é sempre possível substituir um laço que envolve comandos de escape por outro laço que não utiliza estes operadores e preserva a mesma semântica, uma possível diretriz de qualidade de código é a não utilização de comandos de escape para controlar a lógica de continuação das iterações de comandos de repetição. Esta diretriz é referenciada utilizando o C_D EXEC_SCP. As condições necessárias para a checagem estática da diretriz EXEC_SCP são mostradas na Fórmula Lógica 22.

- **Fórmula Lógica 22**: [*is_escape(statement)*]

As condições dispostas na Fórmula Lógica 22 são detalhadas a seguir:

- *is_scape(statement)*: esta condição tem como objetivo verificar se o comando analisado representa um comando de escape no fluxo de execução do procedimento de banco de dados.

A.2.5 Comandos de Retorno em Laços

Uma boa prática na implementação de comandos de repetição cuja quantidade de iterações é definida de forma explítica (ex.: *while (count < 100)*, *for i in 1 .. constant_value* e *for i in 1 .. value_from_db*) é permitir que todas as iterações do comando de repetição sejam de fato executadas pelo programa. Para tal, não devem ser utilizados comandos que interrompam a execução das iterações de comandos de repetição (cuja quantidade de iterações é definida de forma fixa) [9]. Isto porque, se a execução do comando de repetição necessita ser interrompida por alguma checagem realizada pelo programa, o comando deve ser substituído por um laço que não defina uma quantidade fixa de iterações.

Uma vez que as diretrizes EXEC_GOTO e EXEC_SCP já definem a não utilização de desvios (condicionais e incondicionais) em laços de quaisquer naturezas, uma possível diretriz de qualidade de código é a não utilização de comandos de retorno (RETURN) no escopo

de laços que definam uma quantidade fixa de iterações. Esta diretriz é referenciada utilizando o C_D RET_LOOP. As condições necessárias para a checagem estática da diretriz TI_NAT são mostradas na Fórmula Lógica 23.

- **Fórmula Lógica 23**: [*is_return(statement)*] $\wedge$ [*dominated_by_fixed_iteration_loop(statement)*]

As condições dispostas na Fórmula Lógica 23 são detalhadas a seguir:

- *is_return(statement)*: esta condição visa checar se o comando analisado representa um comando de retorno no procedimento de banco de dados;
- *dominated_by_fixed_iteration_loop(statement)*: esta condição tem como objetivo verificar se o comando de retorno identificado pela condição *is_return(statement)* é dominado estritamente por pelo menos um comando de repetição cujo número de iterações é definido de forma fixa no procedimento.

A.2.6 Encapsulamento de Expressões Lógicas

É comum que o código de procedimentos de banco de dados necessite avaliar expressões lógicas complexas para determinar o fluxo de execução do programa. Utilizando como exemplo a avaliação da expressão lógica **Emp1**, a qual é utilizada para verificar se um empregado é elegível para a aposentadoria:

- **Emp1**: ((emp.years_worked + accumulated_vacation(emp)) > 35) AND (employee.age > 60) AND (score(emp) > 5)

Uma possível estratégia para facilitar o entendimento da expressão **Emp1** é encapsular a mesma em uma função que utilize um identificador significativo para o contexto da aplicação, como mostrado no Pseudo Código A.1

```
1 FUNCTION eligible_employee(Employee emp)
2     RETURN ((emp.years_worked + accumulated_vacation(emp)) > 35) AND (
          employee.age > 60) AND (score(emp) > 5)
```

Pseudo Código A.1. Função utilizada para encapsular a expressão Emp1.

Neste contexto, uma possível diretriz de qualidade de código é o encapsulamento de expressões lógicas complexas em funções que possuam identificadores significativos [9]. Esta diretriz é referenciada utilizando o C_D ENC_LOG. As condições necessárias para a checagem estática da diretriz ENC_LOG são mostradas na Fórmula Lógica 24.

- **Fórmula Lógica 24**: [*is_logic_expression(expression)*] $\wedge$ <*is_complex(expression)*>

As condições dispostas na Fórmula Lógica 24 são detalhadas a seguir:

- *is_logic_expression(expression)*: esta condição visa verificar se a expressão analisada representa uma expressão lógica.

- *is_complex(expression)*: esta condição tem como objetivo verificar se a expressão lógica detectada é suficientemente complexa para justificar o encapsulamento da mesma em uma função. Neste caso, a complexidade da expressão é um conceito relativo (i.e., depende da avaliação do desenvolvedor do procedimento) e não pode ser verificada estaticamente. No entanto, é possível utilizar duas heurísticas para se obter indícios sobre a complexidade da expressão lógica analisada: ENC_LOG.hi) verificar se a expressão lógica possui pelo menos j (parâmetro configurado pelo usuário) termos; e ENC_LOG.hii) verificar se a expressão lógica realiza pelo menos k (parâmetro configurado pelo usuário) chamadas a funções.

A.2.7 Disposição de Identificadores

A forma como variáveis e parâmetros são dispostos em procedimentos de banco de dados pode facilitar a legibilidade dos mesmos pelos desenvolvedores [9]. Neste contexto, uma boa prática consiste em dispor apenas um identificador de variável ou parâmetro por linha do código-fonte do procedimento. No caso da disposição de variáveis, a diretriz de qualidade de código é referenciada utilizando o C_D DISP_VAR. Analogamente, no caso da disposição de parâmetros, a diretriz é referenciada utilizando o C_D DISP_PAR. As condições necessárias para a checagem estática das diretrizes DISP_VAR e DISP_PAR são mostradas nas Fórmulas Lógicas 25 e 26, respectivamente.

- **Fórmula Lógica 25**: [*sole_line(variable)*]

- **Fórmula Lógica 26**: [*sole_line(parameter)*]

As condições dispostas na Fórmula Lógica 25 são detalhadas a seguir:

- *sole_line(variable)*: esta condição visa checar se a declaração da variável pelo procedimento de banco de dados é disposta de forma individual nas linhas do código-fonte do procedimento.

As condições dispostas na Fórmula Lógica 26 são detalhadas a seguir:

- *sole_line(parameter)*: esta condição visa verificar se o parâmetro presente na assinatura do procedimento de banco de dados é disposto de forma individual nas linhas do código-fonte do procedimento.

A.2.8 Declaração de Constantes

É bastante comum que valores literais sejam utilizados ao longo do código procedimentos de banco de dados para os mais variados objetivos, tais como testar condições e realizar atribuições. Podem ocorrer casos onde os valores literais utilizados em programas representem informações muito específicas para o contexto da aplicação e, consequentemente, sejam difíceis de serem interpretados por desenvolvedores não habituados ao contexto em questão. Nestes casos, substituir os literais por referências à constantes que referenciadas por identificadores semanticamente representativos pode facilitar o processo de entendimento dos procedimentos de banco de dados [9]. Por exemplo, ao invés de acessar literais como:

- *if (temperature > 120) then*
- *flag := (1 +* $\sqrt{5}$*) / 2*

Substituir tais referências por:

- *if (temperature > MAX_TURBINE_TEMPERATURE) then*
- *flag := PHI*

Nesse contexto, uma possível diretriz de qualidade de código é a substituição do uso de literais não óbvios ao longo do código por referências a constantes que possuam identificadores significativos. Esta diretriz é referenciada utilizando o C_D REF_LIT. As condições necessárias para a checagem estática da diretriz REF_LIT são mostradas na Fórmula Lógica 27.

- **Fórmula Lógica 27**: [*is_literal(expression)*] $\wedge \neg$ <*self_explaining(expression)*>

As condições dispostas na Fórmula Lógica 27 são detalhadas a seguir:

- *is_literal(expression)*: esta condição visa checar se a expressão analisada representa um literal no procedimento de banco de dados analisado;
- *self_explaining(expression)*:esta condição visa checar se o valor da expressão literal detectada na condição *is_literal(expression)* é auto explicativo para os desenvolvedores. Uma vez que o conceito de autoexplicativo é dependente do contexto da aplicação e deve ser avaliado por um desenvolvedor do procedimento, esta condição é indecidível estaticamente. No entanto, ainda é possível utilizar a seguinte heurística para a checagem estática desta condição: REF_LIT.h1) verificar se a expressão literal condiz com pelo menos uma das expressões regulares definidas pelo usuário da abordagem para a análise automática de literais. Neste caso, os literais que condizem com pelo menos uma expressão regular definida pelo usuário da abordagem devem ser considerados auto explicativos. Por exemplo, caso o usuário julgue que os literais inteiros 1 (ou equivalentes como 01 e 0001) e 0 (ou equivalentes como 00 e 000) são suficientemente auto explicáveis, o mesmo deve configurar a expressão regular *[0]*([2-9]+[0-9]*)|(1+[0-9]+)* para os casos em que os literais analisados representem números inteiros.

A.2.9 Múltiplos Comandos de Retorno

Uma boa prática no desenvolvimento de procedimentos de banco de dados é a utilização de um único comando de retorno (RETURN) em cada procedimento (do tipo função) armazenado no banco de dados. Isto porque, a utilização de múltiplos comandos de retorno aumenta

a complexidade do fluxo de controle do programe e, consequentemente, também dificulta o entendimento do programa pelos desenvolvedores [9].

Nos casos em que vários comando de retorno são dispostos na corpo do procedimento (do tipo função), deve ser declarada uma variável (referenciada nesta seção como *proc_ret*) com o mesmo tipo de dados do retorno do procedimento e os valores retornados devem, ao invés disso, ser atribuídos à variável *proc_dec*. Desse modo, pode ser adicionado ao final do procedimento um único comando de retorno utilizando a variável *proc_ret*. Esta diretriz de qualidade de código é referenciada utilizando o C_D MULT_RET. As condições necessárias para a checagem estática da diretriz MULT_RET são mostradas na Fórmula Lógica 28.

- **Fórmula Lógica 28**: ([*is_function(procedure)*] $\wedge$ [*multiple_return_statements(procedure)*])

As condições dispostas na Fórmula Lógica 28 são detalhadas a seguir:

- *is_function(proceure)*: esta condição visa verificar se o procedimento de banco de dados analisado é do tipo função;
- *multiple_return_statements(procedure))*: esta condição visa verificar se o procedimento de banco de dados executa mais de um comando de retorno.

A.2.10 Convenções de Identificadores de Parâmetros

Uma boa prática para o desenvolvimento de procedimentos de banco de dados é a utilização de convenções (usando prefixos ou sufixos) para indicar o tipo de passagem dos parâmetros presentes na assinatura do procedimento [9] [12]. Estas convenções facilitam a identificação de quais operações podem ser realizadas sobre quais parâmetros no corpo do procedimento. Um exemplo de convenção para a nomeação dos tipos de passagem de parâmetros é apresentada a seguir:

- Passagem de parâmetro do tipo valor: ***in**_parameter_name* ou *parameter_name_**in***;
- Passagem de parâmetro do tipo resultado: ***out**_parameter_name* ou *parameter_name_**out***;

- Passagem de parâmetro do tipo valor/resultado: ***in_out**_parameter_name* ou *parameter_name_**in_out***;

A diretriz de qualidade de código referente à utilização de convenções para identificar o tipo de passagem dos parâmetros presentes na assinatura de procedimentos de banco de dados é referenciada utilizando o C_D PAR_CONV. As condições necessárias para a checagem estática da diretriz PAR_CONV são mostradas na Fórmula Lógica 29.

- **Fórmula Lógica 29**: $\neg$ <*follow_parameter_convention(parameter)*>

As condições dispostas na Fórmula Lógica 29 são detalhadas a seguir:

- *follow_parameter_convention(parameter)*: esta condição visa checar se o identificador do parâmetro analisado segue as convenções definidas para o tipo de passagem do mesmo. Neste caso, as convenções (prefixos ou sufixos) devem ser configuradas pelo usuário da abordagem.

A.2.11 Procedimentos com Parâmetro de Retorno Único

Um procedimento de banco de dados cuja assinatura possui um único parâmetro do tipo resultado ou valor/resultado e uma quantidade qualquer de parâmetros do tipo valor representa, de fato, o funcionamento de uma função. Desse modo, uma possível diretriz de qualidade de código é a transformação de procedimentos de banco de dados com estas características em funções. Esta diretriz é referenciada utilizando o C_D PROC_TO_FUNC. As condições necessárias para a checagem estática da diretriz PROC_TO_FUNC são mostradas na Fórmula Lógica 30.

- **Fórmula Lógica 30**: [*is_procedure(procedure)*] $\wedge$ ([*has_sole_result_parameter(procedure)*] $\vee$ [*has_sole_value_result_parameter(procedure)*])

As condições dispostas na Fórmula Lógica 30 são detalhadas a seguir:

- *is_procedure(procedure)*: esta condição visa verificar se o procedimento de banco de dados analisado é do tipo função;

- *has_sole_result_parameter(procedure)*: esta condição tem como objetivo verificar se o procedimento de banco de dados analisado possui um único parâmetro do tipo resultado;

- *has_sole_value_result_parameter(procedure)*: esta condição tem como objetivo verificar se o procedimento de banco de dados analisado possui um único parâmetro do tipo valor/resultado.

A.2.12 Cláusula *ELSE* em Comandos *CASE*

Na execução de procedimentos de banco de dados de linguagens de programação (PL/SQL, Transact-SQL, PL/pgSQL), pelo menos uma das condições dos ramos de comandos *CASE* deve ser avaliada como verdadeiro; caso contrário, a impossibilidade de seguir o fluxo do programa causa um erro de execução. Por este motivo, uma possível diretriz de qualidade consiste em sempre inserir uma cláusula *ELSE* na estrutura de comandos *CASE*, e assim, impedir a ocorrência de erros de execução desta natureza. Esta diretriz é referenciada utilizando o código de diretriz ELSE_CASE. As condições necessárias para a checagem estática da diretriz ESLE_CASE são mostradas na Fórmula Lógica 31.

- **Fórmula Lógica 31**: [*is_case(statement)*] $\wedge$ (*has_else_clause(statement)*]

As condições dispostas na Fórmula Lógica 31 são detalhadas a seguir:

- *is_case(statement)*: esta condição visa verificar se comando analisado é do tipo *CASE*;

- *has_else_clause(statement)*: esta condição visa checar se o comando *CASE* detectado na condição *is_case(statement)* possui uma cláusula *ELSE*.