Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e Computação
Departamento de Engenharia de Computação e Automação Industrial

Análise de Mutantes em Aplicações SQL de Banco de Dados

Autor: Andrea Gonçalves Cabeça
Orientador: Mario Jino
Co-Orientador: Plínio de Sá Leitão Júnior

Trabalho apresentado à Faculdade de Engenharia Elétrica e de Computação da UNICAMP como parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica na área de Engenharia de Computação.

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA E ARQUITETURA - BAE - UNICAMP

Cabeça, Andrea Gonçalves
C111a Análise de mutantes em aplicações SQL de banco de dados / Andrea Gonçalves Cabeça. --Campinas, SP: [s.n.], 2009.

Orientadores: Mario Jino, Plínio de Sá Leitão Júnior.
Dissertação de Mestrado - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Software - Desenvolvimento - Banco de Dados. 2. SQL (Linguagfem de programação de computador). 3. Programas de computador - Testes. 4. Engenharia de software. I. Jino, Mario. II. Leitão Júnior, Plínio de Sá. III. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. IV. Título.

Título em Inglês: Mutation analysis for SQL database applications
Palavras-chave em Inglês: Software development database, SQL (Computer software language), Computer software testing, Software enginering
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora: Auri Marcelo Rizzo Vincenzi, Ivan Luiz Marques Ricarte
Data da defesa: 16/02/2009
Programa de Pós Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - TESE DE MESTRADO

Candidata: Andrea Gonçalves Cabeça

Data da Defesa: 16 de fevereiro de 2009

Título da Tese: "Análise de Mutantes em Aplicações SQL de Banco de Dados"

Prof. Dr. Mario Jino (Presidente): _____

Prof. Dr. Auri Marcelo Rizzo Vincenzi: _____

Prof. Dr. Ivan Luiz Marques Ricarte: _____

Resumo

O teste de aplicações de banco de dados é crucial para assegurar a alta qualidade do software, pois defeitos não detectados podem resultar em corrupção irrecuperável dos dados. SQL é a mais amplamente utilizada interface para sistemas de banco de dados. Nossa abordagem visa a alcançar testes efetivos pela seleção de bases de dados reveladoras de defeitos. Usamos a análise de mutantes em comandos SQL e discutimos dois cenários para aplicar as técnicas de mutação forte e fraca. Uma ferramenta para auxiliar na automatização da técnica foi desenvolvida e implementada. Experimentos usando aplicações reais, defeitos reais e dados reais foram conduzidos para: (i) avaliar a aplicabilidade da abordagem; e (ii) comparar bases de dados de entrada quanto à habilidade para detectar defeitos.

Palavras-Chaves: Teste de Mutação, Qualidade de Software, Análise de Mutantes, Banco de Dados, Aplicações de Banco de Dados, SQL, Automação do teste.

Abstract

Testing database applications is crucial for ensuring high quality software as undetected faults can result in unrecoverable data corruption. SQL is the most widely used interface language for relational database systems. Our approach aims to achieve better tests by selecting fault-revealing databases. We use mutation analysis on SQL statements and discuss two scenarios for applying strong and weak mutation techniques. A tool to support the automatization of the technique has been developed and implemented. Experiments using real applications, real faults and real data were performed to: (i) evaluate the applicability of the approach, and (ii) compare fault-revealing abilities of input databases.

Keywords: Mutation Testing, Software Quality, Mutation Analysis, Databases, Database Aplications, SQL, Automatizion.

Ao meu pai, Antonio Gonçalves Cabeça

que dedicou a sua vida inteira para que um dia eu pudesse fazer esta dedicatória.

(in memorian)

Agradecimentos

Ao professor Maurício Magalhães pela oportunidade de pisar na Unicamp como aluna. Não tenho como agradecer ao Mario Jino pela enorme paciência enquanto eu colocava milhares de coisas na frente dos estudos. Ao Plínio, meu co-orientador, que já me orientava muito antes. Sem seus conselhos e críticas este trabalho nunca seria terminado.

À minha mãe, Maria do Carmo, por sua doçura e compreensão e por nunca ter deixado de apoiar minhas decisões, erradas ou não. Ao meu pai por ter sido duro o suficiente para me mostrar que quando você quer algo nada pode te impedir. Aos meus outros pais, Beta e Adalgiso por me amarem e me apoiarem. A minha tia Dada pelo carinho.

Aos meus irmãos Marcos, Marcio e Fábio por mostrarem o caminho que eu deveria seguir e por serem as constantes da minha vida. Ao Paulinho, pelo semestre mais cantante desta jornada. Às minhas irmãs Débora, Flávia, Laura e Regina por serem minha família. Aos meus sobrinhos que estão chegando e com isso trazendo uma alegria enorme a mim e a todos aqueles que amo. À minha prima Andrea por ser um exemplo a seguir.

Ao Ed, Fe, Le e Bruno gostaria de me desculpar pela ausência na vida deles enquanto eles nunca se ausentaram na minha. À Debys, à Kiki e ao Fe por serem exemplos vivos de que nada é impossível. À Van por me mostrar que um trabalho bem feito é aquele que tem fim. Ao Koba, por me ensinar que paciência é uma virtude. Ao Adriano por me ajudar a terminar este trabalho e por sempre incentivar minha leitura. Ao Eros, por ter me ensinado mais do que inglês em uma época que eu era tola e pretensiosa. Ao Ed, à Kiki e à Laura por estarem ao meu lado na caminhada mais difícil da minha vida.

Aristóteles definiu um amigo como uma única alma habitando dois corpos, portanto à Kiki por acreditar tanto em mim que mesmo quando eu sinto que não tenho mais forças para continuar eu continuo apenas para não decepcioná-la.

Parafraseando Derfel Cadarn existe um momento em que é possível olhar nos olhos de alguém e saber que a vida será impossível sem eles. Saber que a sua voz pode fazer seu coração falhar, e que a companhia dela é tudo que sua felicidade pode desejar e que a sua ausência deixará sua alma solitária, desolada e perdida. Ao Lu por ser este alguém.

Sumário

Lista de Tabelas

Lista de Figuras

Trabalhos afins Publicados pela Autora

1. A. G. Cabeça, P. S. Leitão-Junior e M. Jino. Análise de Mutantes em Aplicações SQL de Banco de Dados. VII Simpósio Brasileiro de Qualidade de Software (SBQS 2008), Brasil, Florianópolis, Junho 2008.

1. Introdução

1.1. Contexto

A incapacidade humana para executar tarefas e comunicar-se com perfeição é um dos fatores motivadores para que o desenvolvimento de software seja acompanhado por esforços direcionados à garantia de qualidade [Deutsch, 2002]. Idealmente, tais esforços devem ser aplicados durante todo o processo de desenvolvimento. Nesse cenário, a Engenharia de Software é uma disciplina que propõe métodos, ferramentas e procedimentos visando ao aumento da produção, comprometida com sua boa qualidade.

Da perspectiva de software, podemos dizer que o esforço para a melhoria da qualidade é expressivo tanto no meio industrial quanto no acadêmico. Para cada fase do processo de desenvolvimento encontramos dezenas de metodologias e guias para auxiliar o profissional na busca da qualidade, na direção da confiabilidade e da corretude do software. Nesse contexto, o teste de software aparece como ferramenta para a busca da qualidade, sendo um elemento crítico para este fim, assumindo um papel importante, pois representa, segundo Pressman (2005), a última revisão de especificação, projeto e codificação.

O teste de software consome em média 50% do tempo e do custo do desenvolvimento de um produto de software [Pressman, 2005; Myers, 2004]. Este custo é, em parte, resultante da carência de profissionais qualificados e de esforços empregados na construção de ferramentas que auxiliem na aplicação do teste.

O teste é basicamente formado por atividades de planejamento, desenvolvimento de casos de teste, execução e avaliação de resultados. Casos de teste – especificação da entrada (dado de teste) e de sua saída esperada – devem ser projetados para que o menor número deles detecte o maior número de defeitos possíveis, diminuindo, portanto, o custo associado.

Diversas técnicas de teste têm sido propostas e podem ser classificadas em *estruturais*, *funcionais* e *baseadas em defeitos*. A primeira requer que aspectos da estrutura do software implementado sejam utilizados na definição dos requisitos de teste; a segunda

estabelece requisitos de teste a partir da especificação funcional do software; e a terceira utiliza informação de defeitos típicos para a geração de dados de teste.

A análise de mutantes ou teste de mutação [DeMillo, 1978] é um dos critérios de teste que utiliza informação de defeitos para a geração de dados de teste. O fundamento desse critério é baseado na premissa de que programadores experientes escrevem programas próximos do correto; esta premissa é conhecida também como "hipótese do programador competente". Desta forma, pela inserção de pequenas "perturbações", uma por vez, no programa original, é esperado que alguns erros sejam descobertos [Woodward, 1993]. Para a definição de quais modificações serão feitas no programa original um conjunto de *operadores de mutação* é definido. A inserção destas perturbações gera programas alterados denominados *Mutantes.* Um conjunto de casos de teste é então utilizado para executar os mutantes, com o intuito de revelar as alterações feitas pelos operadores de mutação. Se o mutante apresentar uma saída diferente da obtida pelo programa original ele é dito *morto*, senão ele é dito *vivo* e um estudo mais aprofundado deve ser feito para apontar se os casos de teste utilizados podem ser aperfeiçoados ou não para matá-lo.

Apesar do estado da arte para o desenvolvimento de software ter progredido drasticamente nas últimas décadas, sendo criados novos paradigmas, linguagens de programação e arquitetura de computadores, observa-se que o estado da prática em muitas organizações não progrediu na mesma proporção. Um exemplo disso é o fato dos modelos de banco de dados ainda serem fortemente baseados no modelo de Codd (1970). Os bancos de dados desempenham um papel de destaque nas operações da maioria das organizações modernas [Chays et al., 2000].

De acordo com Kapfhammer e Soffa (2003), uma aplicação de banco de dados é um programa cujo ambiente sempre contém uma ou mais bases de dados. Em geral, a aplicação de *Banco de Dados Relacional* pode ser composta por vários programas codificados em diversas linguagens como C e Java, sendo sua principal característica a manipulação de dados por comandos que acessam as bases de dados controladas por um *Sistema Gerenciador de Banco de Dados* (SGBD) [Spoto, 2000]. No âmbito das aplicações de banco de dados, as bases de dados integram um conjunto de dados de entrada dos testes, fazendo com que um caso de teste seja formado por: (i) um dado de

entrada para um programa; (ii) um conjunto de instâncias de bases de dados e; (iii) a saída esperada para este dado de entrada e instâncias das bases.

A disciplina de teste de software para aplicações de banco de dados seguiu essa tendência e, apenas recentemente, tem progredido na direção da melhoria de qualidade dessas aplicações [Chan e Cheung, 1999; Chays et al., 2000; Kapfhammer e Soffa, 2003; Zhang et al., 2001; Leitão et al., 2005; Carey, 1993].

A SQL (*Structured Query Language*) é uma linguagem padrão utilizada para a comunicação com um banco de dados relacional, representando a forma de acesso das aplicações às bases de dados. Ela foi inicialmente projetada e implementada pela IBM como uma interface para um sistema de banco de dados relacional denominado *Sistema R*. Desde sua criação ela tem sofrido várias modificações, o que resultou em diferentes tipos de comandos, muitas vezes incompatíveis entre os fornecedores [Spoto, 2000], tornando pertinente a definição de padrões para a linguagem. A versão contemporânea do padrão é a ANSI SQL-3[ANSI/ISSO, 1999], que possui cinco documentos inter-relacionados visando à cobertura de todos os aspectos relacionados à linguagem. Existem esforços para testar a SQL embutida em aplicações de bases de dados, tais como em Tuya et al. (2007) e Chan et al. (2005), mas em escala inferior aos esforços direcionados ao teste de programas convencionais (programas sem acesso a bases de dados).

1.2. Motivação

A SQL é a linguagem mais usada pelos SGBDRs comerciais [Spoto, 2000], estando presente significativamente em aplicações de banco de dados. Técnicas de teste para programas convencionais e até mesmo para a geração de bases de dados têm sido propostas, implementadas e avaliadas [Chan e Cheung, 1999 e Chays e Deng, 2003]; contudo, relativamente pouco esforço tem sido dedicado ao estudo da linguagem SQL [Suárez-Cabal e Tuya, 2004; Tuya et al., 2007; Chan et al., 2005 e Leitão-Junior et al., 2005]. Apesar das iniciativas correntes de pesquisa na área de testes estarem considerando a linguagem SQL e a qualidade dos dados das bases alvos, ainda existe uma carência de abordagens sistemáticas de testes na busca da melhoria das aplicações de banco de dados. Alguns exemplos de trabalhos existentes são os de Tuya et al. (2007) e

Chan et al. (2005), nos quais o estudo da aplicação de operadores de mutação considera o comando de manipulação SELECT para a definição de operadores de mutação.

1.3. Objetivos

Este trabalho visa a realizar experimentos com o teste de mutação aplicado a SQL em cenários distintos: (i) teste isolado de comandos SQL e (ii) teste de scripts SQL gerados a partir de aplicações de banco de dados. São empregados operadores de mutação que constituem uma evolução dos propostos na literatura e que representam defeitos comuns inseridos por programadores SQL. É utilizado suporte automatizado na geração de mutantes e na avaliação do teste, visando a analisar a eficácia dos conjuntos de teste (bases de dados de entrada).

Especificamente, os seguintes objetivos são visados:

- estudo dos defeitos relacionados aos comandos SQL que podem ser modelados com operadores de mutação;
- análise e extensão de um conjunto de operadores de mutação para aplicações que utilizam comandos de manipulação SQL;
- investigação e definição dos cenários de aplicação do teste de mutação para a SQL;
- implementação de uma ferramenta para o suporte à geração, à execução e à análise de mutantes pela aplicação de operadores de mutação para a SQL;
- validação dos operadores de mutação e da automatização do processo para a técnica proposta por meio da ferramenta implementada;
- condução de experimentos que investiguem a eficácia dos operadores propostos nos cenários analisados, utilizando dados e sistemas reais.

1.4. Organização

Esta introdução discute o contexto no qual o trabalho se insere, a motivação e os objetivos da pesquisa.

No Capítulo 2 é apresentada uma revisão de teste de banco de dados e da análise de mutantes. São abordados conceitos básicos de teste de software e termos ligados ao modelo relacional, à SQL, ao critério de análise de mutantes e à aplicação deste critério em banco de dados.

O Capítulo 3 apresenta contribuições para o emprego da análise de mutantes nas aplicações que utilizam a linguagem de manipulação SQL, por meio de definição de novos conceitos e pela investigação dos cenários possíveis para a aplicação da análise de mutantes.

A definição dos operadores de mutação para os comandos de manipulação da SQL encontra-se no Capítulo 4.

No Capítulo 5 são apresentados os experimentos que objetivam a investigação da aplicabilidade e da habilidade de detecção de defeitos dos operadores de mutação (definidos no Capítulo 4) e seu comportamento nos cenários apresentados no Capítulo 3.

No Capítulo 6 é apresentada uma síntese do trabalho e suas contribuições.

São ainda incluídos apêndices que complementam o conteúdo da tese: descrição da ferramenta que suporta a automatização da geração, execução e análise dos mutantes; exemplos da aplicação dos operadores de mutação e dados obtidos pela aplicação dos operadores de mutação nos experimentos descritos no Capítulo 5.

2. Revisão Bibliográfica

Neste capítulo as técnicas de teste de software são revistas, sobretudo a análise de mutantes, e as pesquisas relacionadas ao teste de banco de dados são discutidas de acordo com seu foco: geração de dados de teste, teste de aplicações de bases de dados e teste de projeto de bases de dados. Inicialmente, as técnicas de teste de software são revistas, visando a elucidar os principais conceitos relacionados ao teste. Em seguida, é apresentada a terminologia relativa a banco de dados, mais especificamente os termos relativos ao modelo relacional. São discutidas as principais iniciativas de pesquisa em testes de banco de dados e de aplicações que usam a linguagem SQL. A análise de mutantes é ressaltada e os principais conceitos relacionados a esta técnica são apresentados. Ao final, as pesquisas voltadas ao estudo da extensão de operadores de mutação para banco de dados são discutidas.

2.1. Teste de Software

Segundo Myers (2004), teste é o processo de executar um programa com a intenção de descobrir defeitos, sendo este o seu principal objetivo. Deve ser visto como um processo destrutivo, ao contrário dos demais passos da Engenharia de Software, nos quais os processos são direcionados para a construção do sistema.

Em geral, é necessário que um conjunto de teste seja gerado com o objetivo de satisfazer a um critério associado a alguma técnica do teste de software. Basicamente casos de testes são formados por (i) dados de entrada, que fazem parte dos domínios de entrada válidos e inválidos do programa, e (ii) resposta esperada na execução do programa com os dados de entrada de acordo com a sua especificação [Maldonado, 1991].

Idealmente, um caso de teste deveria revelar um defeito que ainda não tenha sido descoberto. Na execução do programa com um particular dado de entrada, se a saída produzida for diferente da esperada, significa que a aplicação do caso de teste revelou a presença de um defeito no programa em teste. Para que a análise dos resultados da execução do teste seja feita, é necessário o uso de um *oráculo*. Qualquer mecanismo (automatizado ou manual) que seja capaz de avaliar um resultado pode ser chamado de oráculo [Weyuker, 1982].

Segundo Pressman (2005), à medida que os resultados de teste são reunidos e avaliados, uma indicação da qualidade e da confiabilidade do software começa a surgir. O teste de software é um procedimento crítico para a qualidade do software, representando a última atividade, na qual é possível revelar defeitos na especificação, no projeto e na codificação do software.

Para que bons dados de testes sejam selecionados, aumentando a probabilidade da descoberta de defeitos, é necessária a aplicação de critérios de testes. Esses critérios estabelecem requisitos a serem satisfeitos e que podem orientar: (i) a seleção de dados de teste; (ii) a avaliação da qualidade do teste; e (iii) a definição de requisitos de suficiência a serem atingidos para o encerramento da atividade de teste [Bueno, 1999].

Segundo o padrão IEEE (1991), um defeito (*fault*) é a manifestação física de um engano (ação humana) em uma representação do software. Uma falha (*failure*) é a ocorrência observável de um ou mais defeitos quando o software é testado ou utilizado em campo. O estado intermediário ou final caracterizado por um comportamento incorreto ou por um desvio da especificação é chamado de erro (*error*). O conhecimento dos defeitos é crucial para a programação SQL e para os testes em banco de dados [Leitão-Junior et al., 2005].

As subseções seguintes apresentam as três principais técnicas de teste de software: funcional, estrutural e baseada em defeitos.

2.1.1. Teste Funcional (*black-box*)

O teste funcional, ou caixa preta, é uma abordagem na qual o teste é derivado da especificação do sistema [Sommerville, 2003]. O comportamento do software é estimado levando em consideração as entradas e saídas do sistema, independentemente de sua estrutura interna, focando-se, assim, em aspectos funcionais do software.

Os principais defeitos que os testes funcionais tentam revelar são funções incorretas ou omitidas, erros de interface, erros de estrutura de dados ou de acesso à base de dados externa, erros de comportamento ou desempenho e erros de iniciação e término [Pressman, 2005].

Os critérios de teste funcional mais populares são: *particionamento de equivalência* e *análise de valores limites*. No particionamento de equivalência o domínio de entrada do

programa é dividido em subconjuntos; o testador seleciona um ou mais elementos de cada subconjunto. A meta deste particionamento é fazer as divisões (classes de equivalência) de forma que quando o testador selecionar casos de testes baseados nos subconjuntos, o resultado do conjunto de teste seja uma boa representação do domínio inteiro [Jeng e Weyuker, 1989]. O particionamento de equivalência divide os dados em partições por similaridade, mas não determina quais dos dados existentes em uma partição têm a maior probabilidade de encontrar um defeito. A análise de valores limites visa a considerar essa possibilidade ao levar o testador a escolher valores de dados que exercitem as condições limites das classes de equivalência do domínio; esse critério baseia-se na suposição, não comprovada cientificamente, de que é mais provável encontrar defeitos nos limites dos valores dos dados de entrada, pois os programadores e projetistas tendem a utilizar valores típicos de entrada não se preocupando com os valores limites dos dados [Myers, 2004].

2.1.2. Teste Estrutural (*white-box*)

O teste estrutural, ou caixa branca, é uma abordagem baseada no conhecimento da estrutura interna da implementação do software, fazendo uso da estrutura de controle do projeto procedimental para derivar os casos de teste [Kapfhammer, 2003]. Existem diversos critérios de teste estrutural, cada um visando a exercitar aspectos diferentes de um programa. Os mais conhecidos são: *complexidade ciclomática, fluxo de controle e fluxo de dados*.

A complexidade ciclomática [McCabe, 1976] é uma métrica de software que mede a complexidade de um programa. O valor calculado define o número de caminhos independentes no conjunto base de um programa e fornece um limite superior para a quantidade de testes que deve ser conduzida para garantir que todos os comandos e todos os desvios condicionais tenham sido executados pelo menos uma vez. Há diversas técnicas de teste estrutural baseadas na complexidade ciclomática como, por exemplo, o teste de caminhos básicos, que visa a exercitar o maior número de caminhos independentes dentro de um programa [Beizer, 1990].

Os critérios baseados no fluxo de controle utilizam as características de controle da execução do programa para determinar quais estruturas são requeridas. Esses critérios

utilizam uma representação gráfica do programa, chamada grafo de fluxo de controle ou grafo de programa [Rapps, 1985] para determinar as estruturas que devem ser cobertas pelos casos de teste. Exemplos de critérios baseados nesta técnica são: (i*) todos nós*: requer que nós, isto é, comandos de um programa P sejam exercitados pelo conjunto de casos de teste (critério todos os nós); (ii) *todos ramos*: requer que desvios condicionais de um programa P sejam exercitados pelo conjunto de casos de teste (critério todos os arcos) e; (iii*) todos caminhos*: requer que caminhos de um programa P sejam exercitados pelo conjunto de casos de teste.

Os critérios baseados em fluxo de dados visam a utilizar a análise de fluxo de dados como fonte de informação para derivar os requisitos de testes, isto é, baseiam-se nas associações entre as definições das variáveis e seus possíveis usos subseqüentes. A principal carência desses critérios é que na presença de caminhos não executáveis eles não garantem a inclusão do critério *todos ramos*. Os critérios potenciais usos aparecem neste contexto como uma alternativa para esta carência, pois estabelecem uma hierarquia de inclusão de critérios contendo os critérios *todos ramos* e *todos caminhos,* mesmo na presença de caminhos não executáveis [Maldonado, 1991].

2.1.3. Teste Baseado em Defeitos

A técnica de teste baseada em defeitos utiliza informações sobre os enganos mais freqüentes cometidos no processo de desenvolvimento de software e sobre os tipos específicos de defeitos que se desejam revelar [DeMillo, 1987], e é bem sucedida devido ao fato de que programadores tendem a criar defeitos que podem ser bem definidos [Offutt, 1992]. Uma vez que o número de defeitos em um dado programa pode ser enorme, o teste baseado em defeitos assume que pelo teste de uma classe de defeitos pode-se descobrir diversas outras classes deles [Offutt, 1992]. A técnica está baseada em dois princípios: *Hipótese do Programador Competente* [DeMillo et al., 1979] e *Efeito de Acoplamento* [Demillo, 1978]. A hipótese do programador competente pressupõe que o programador tende a escrever programas "próximos" do correto. O efeito de acoplamento

pressupõe que um conjunto de dados de testes que detecta todos os defeitos simples de um programa também consegue revelar defeitos mais complexos[1] [Demillo, 1978].

O critério baseado em defeitos mais popular é conhecido como análise de mutantes ou teste de mutação, originalmente proposto por DeMillo (1978), e utiliza a inserção de pequenas alterações no programa em teste, resultando em programas ligeiramente diferentes do original, chamados de mutantes. O objetivo é encontrar um conjunto de teste capaz de revelar as diferenças de comportamento existentes entre um programa e seus mutantes. A Seção 2.5 contém maiores detalhes sobre este critério.

2.2. Terminologia de Banco de Dados

Um *banco de dados* é uma coleção de dados relacionados [Elmasri, 2006]. *Sistema Gerenciador de Banco de Dados (SGBD)* é um conjunto de programas que permite ao usuário executar operações como criar e manter bancos de dados. A junção de um sistema gerenciador de banco de dados com um ou mais bancos de dados constitui os chamados *sistemas de banco de dados*. Uma *aplicação* ou *aplicação de banco de dados* consiste em um ou mais programas que interagem com sistemas de banco de dados. Os *bancos de dados relacionais* representam os dados como um conjunto de relações. Uma *relação* é uma tabela na qual cada linha representa dados sobre uma entidade particular e cada coluna representa um aspecto particular dos dados. Um *esquema de relação* é a descrição das estruturas dos dados pertencentes ao banco de dados e um *esquema de um banco de dados relacional* é um conjunto de esquemas de relações que possuem restrições de integridade, isto é, um conjunto de esquemas de relações acrescidos de restrições com relação às persistências que serão acrescentadas no banco de dados. Um atributo pode sofrer diversas restrições como uma *restrição explícita de domínio* (por exemplo, atributo dependente de constantes ou dependente do próprio atributo) ou uma *restrição explícita de integridade referencial* (por exemplo, atributos declarados como chave primária ou chave estrangeira). Uma *chave primária* garante que cada linha de dados de uma tabela seja única dentro de um banco de dados relacional e uma *chave estrangeira* garante a integridade referencial do banco de dados, na qual os valores de

[1] Um defeito simples é um defeito que pode ser consertado com uma única mudança no código fonte; e um defeito complexo não pode ser consertado com uma única mudança no código fonte [Offutt, 1992].

uma coluna em uma tabela dependem dos valores de uma coluna em outra (ou na mesma) tabela.

2.3. SQL

A SQL (*Structured Query Language*) é uma linguagem padrão utilizada para a comunicação com um banco de dados relacional. A SQL foi inicialmente projetada e implementada pela IBM como uma interface para um sistema de banco de dados relacional denominado *Sistema R*. Atualmente, a SQL é a linguagem mais usada pelos SGBDRs comerciais e tem sido bastante modificada pelos diversos fornecedores, o que resultou em diferentes tipos de comandos, muitas vezes incompatíveis entre os fornecedores [Spoto, 2000], o que motivou a definição de padrões para a linguagem. A versão contemporânea do padrão é a ANSI SQL-3, que é descrita em cinco documentos inter-relacionados visando à cobertura de todos os aspectos da linguagem [ANSI-SQL3]. Este padrão contempla: (i) linguagem para a definição de dados (DDL - *Data Definition Language*), para a descrição das relações do banco de dados; (ii) linguagem de manipulação de banco de dados (DML - *Data Manipulation Language*),para especificar a manipulação do estado da base de dados; (iii) linguagem de consulta de dados (DQL - *Data Query Language*), para especificar a consulta ao banco de dados; (iv) linguagem de controle dos dados (DCL - *Data Control Language*), para especificar o controle de acesso aos dados contidos no banco de dados; (v) comandos de administração de dados; e (vi) controle de transações [ANSI – SQL3].

Uma *Aplicação de Banco de Dados Relacional (ABDR)* é composta por um ou mais módulos (programas), em que cada módulo é composto por unidades de programas que são os procedimentos ou funções de um programa; um módulo pode ser composto por uma ou mais unidades de programa e pode ser codificado em diferentes linguagens de programação, denominadas *linguagens hospedeiras*[2] com SQL embutida. Cada unidade de programa é uma seqüência finita de comandos da linguagem hospedeira e comandos da linguagem SQL [Spoto et. al., 2005].

[2] Linguagens de programação como C ou Java que permitem a utilização da SQL.

2.4. Testes de Banco de Dados

Muitos esforços têm sido empregados para garantir a qualidade dos softwares desenvolvidos, porém poucos foram direcionados ao aumento da qualidade de aplicações para banco de dados [Chan e Cheung, 1999; Chays et al., 2000; Kapfhammer e Soffa, 2003; Zhang et al., 2001; Leitão-Junior et al., 2005]. A qualidade dos bancos de dados e das aplicações que os acessam são de grande interesse e importância para os mundos acadêmico e industrial.

Segundo Leitão-Junior (2005), podemos classificar os testes de banco de dados em: (i) teste de aplicações de banco de dados; (ii) teste de projeto de banco de dados; e (iii) geração de bases de dados para testes. Na primeira, estão incluídos programas com SQL embutida e rotinas escritas em SQL que utilizam extensões de fluxo de controle, tais como regras ativas; o segundo tem como foco avaliar o esquema da base de dados frente às especificações funcionais para a base de dados; e o terceiro envolve a construção de bases de dados na forma de dados de teste para as categorias de teste anteriores.

Algumas iniciativas, na maior parte feitas pelo meio industrial, mostram a preocupação com a avaliação do desempenho dos bancos de dados e, geralmente, não tratam diretamente a aplicação dos métodos de teste, pois não são pesquisas acadêmicas; seu interesse maior é a qualidade dos bancos de dados [Carey et al., 1993; TPC, 2007].

2.4.1. Teste de Aplicações de Banco de Dados

Os testes para aplicações de banco de dados relacionais (ABDR) possuem características similares ao teste de programas convencionais. As técnicas de testes definidas para programas convencionais podem ser aplicadas em ABDRs a partir de algumas adaptações, tendo em vista a presença de comandos SQL não existentes nos programas convencionais [Spoto, 2000]. Para realizar um teste para ABDRs normalmente busca-se dentro da linguagem hospedeira os códigos em SQL, que usualmente são encontrados na forma de comandos ou rotinas. Esta busca pode ser efetuada de diversas maneiras, tais como: pela leitura do código fonte da linguagem hospedeira ou pela observação da saída da execução do programa em relação aos acessos aos bancos de dados. Uma vez encontrados os comandos SQL, estes se tornam o objeto de teste e é então projetado um conjunto de casos de testes para exercitar estes comandos.

Leitão-Junior et al. (2005) investigam os relacionamentos entre tipos de defeitos e categorias de falhas em comandos de manipulação escritos em SQL. Uma lista de tipos de defeitos é apresentada a partir da estrutura dos comandos SELECT, INSERT, DELETE e UPDATE, e estes tipos são associados ao conceito de categoria de falha de manipulação, definido no artigo.

Suárez-Cabal e Tuya (2004) apresentam a definição de uma medida de cobertura para comandos de consulta SQL (SELECT), visando a usá-la na forma de critério de adequação para o teste de aplicações que acessam bases de dados. Para tanto, a abordagem utiliza as condições presentes nos predicados para a seleção de tuplas, buscando cobrir os valores *verdadeiro* e *falso* em cada combinação. As consultas são normalizadas segundo o padrão SQL3 [ANSI SQL]; são localizados, na cláusula FROM, os predicados atribuídos à operação de junção; e, na cláusula WHERE, os demais predicados de seleção de tuplas. Uma árvore de cobertura é construída visando a auxiliar a avaliação das condições, sendo que cada nível da árvore representa uma condição da consulta, iniciando pela cláusula FROM (quando existem operações de junção), seguidapelas condições da cláusula WHERE, na mesma ordem em que estão postas na consulta. Segundo os autores a árvore de cobertura é um recurso poderoso para a inserção de tuplas quando se objetiva alcançar uma dada cobertura e para reduzir o tamanho de uma base conservando a mesma medida de cobertura da base de dados original. O artigo menciona a existência de situações impossíveis na árvore, devido às restrições no esquema da base de dados, as quais impactam o cálculo de cobertura. A necessidade de se reduzir o número de nós da árvore de cobertura também é mencionada pelos autores.

Kapfhammer e Soffa (2003) propõem uma família de critérios de adequação para o teste intra-procedimental de aplicações de banco de dados. Os critérios são baseados em informações do fluxo dos dados persistentes, manipuláveis pela aplicação, para os diversos graus de granulosiidade da análise de fluxo de dados persistentes.

2.4.2. Teste de Projetos para Base de Dados

O principal objetivo dos testes de projetos para base de dados é a avaliação do esquema da base de dados frente às suas especificações funcionais. Os testes para base de dados diferem do conceito de teste normalmente encontrado na literatura, na qual teste significa

exercitar um programa com os dados de entrada e comparar o resultado computacional com o esperado, baseado nas especificações [Pressman, 2005]; isto é, o programa de aplicação e os procedimentos são o foco dos testes e os dados nas bases de dados são os dados de testes. No teste de banco de dados o esquema da base de dados, não considerado no teste tradicional, é o foco de testes e os programas de aplicação e procedimentos passam a ser dados de testes, pois representam a forma pela qual o esquema da base de dados é exercitado.

Aranha et al (2000) apresentam uma família de critérios de testes com o intuito de exercitar as relações e relacionamentos da estrutura lógica de um banco de dados. No teste de relação o principal objetivo é revelar defeitos na estrutura e na definição dos atributos em relação a uma unidade da base de dados (tabela). O teste de relacionamentos tem como principal objetivo revelar defeitos nos relacionamentos entre as unidades da base de dados, pelo exercício das chaves (primárias e estrangeiras).

Robbert e Maryanski (1991) propõem uma ferramenta que gera um plano de testes baseado no dicionário de dados e em outras definições do banco de dados; possui o intuito de orientar o testador sobre quais testes devem ser conduzidos. Estas orientações incluem diversos tipos de informação, tais como valores válidos e inválidos e relações a serem exercitadas. Após a execução dos testes, os testadores precisam avaliar os resultados para garantir, por exemplo, que valores inválidos não estão sendo inseridos. O principal intuito da ferramenta é o auxílio aos testes de restrições, aplicados na fase de projeto de banco de dados.

Zhang et al. (2001) estudam a geração de instâncias de banco de dados que é feita a partir da semântica de comandos SQL embutidos em aplicações de banco de dados. Este estudo cria um conjunto de restrições a partir de uma ferramenta, que apresenta uma propriedade contra a qual o programa deve ser testado. A saída da ferramenta apresenta um conjunto de restrições que pode ser utilizado por solucionadores de restrição [Zhang, 2000] para a geração de instâncias de banco de dados.

2.4.3. Geração de Bases de Dados para Testes

A geração de bases de dados para testes vem sendo estudada tanto pelo meio acadêmico como pelo meio industrial, devido à grande influência que as bases de dados têm nos

resultados dos testes. Esta influência ocorre porque a base de dados faz parte dos dados de entrada dos casos de testes, que visam a exercitar a aplicação que acessa a base de dados; diferentes estados da base exercitam a aplicação de formas diferentes. Quanto maior a diversidade de persistências de dados (registros com aspectos diferentes em uma base de dados) de uma base de dados maior a sua capacidade de revelar defeitos e maior a cobertura da aplicação em teste.

Chays et al. (2000) exploram a geração de bases de dados pela aplicação de uma técnica similar ao particionamento de equivalência [Ostrand e Balcer, 1988] para aplicações que utilizam a SQL na manipulação de dados. A automatização desta técnica é feita por meio da ferramenta AGENDA (*A Test Generator for Database Applications*) [Chays et al., 2002; Chays e Deng, 2003].

A partir da constatação de que algumas das técnicas de teste tradicionais não consideram os comandos de manipulação de dados da SQL na geração de casos de testes, Chan e Cheung (1999) propõem o mecanismo WHODATE (*White Box Database Applicaton Testing*) que visa a transformar comandos SQL em fragmentos de programação (definidos em linguagens de programação como C), permitindo assim o uso de técnicas caixa-branca convencionais para a geração de casos de testes.

Daou et al. (2001) fazem uma análise de impacto, da perspectiva do teste de regressão, em relação às alterações feitas nos módulos de banco de dados. O foco do estudo é extrair uma quantidade de casos de teste que exercitem os componentes alterados nos módulos do banco de dados. Uma ferramenta foi construída e utilizada para apoiar a análise. Outra iniciativa para os testes de regressão para bases de dados é feita por Willmor e Embury (2005), que propõem um algoritmo para a seleção de dados que considera o número de interações da aplicação e o estado da base de dados para a seleção dos dados no teste de regressão.

Davies et al. (2000) propõem uma abordagem para a geração de bases de dados visando à produção de arquivos texto com uma combinação de valores válidos e inválidos, do ponto de vista das restrições da base de dados, com o objetivo de criar um mecanismo para o teste das restrições de integridade de dados, definidas no esquema de banco de dados.

Mannila e Räihä (1986) propõem a construção de bases de teste para comandos de consulta individuais, construídos por meio da geração de dados representativos das dependências funcionais obtidas das especificações das relações atribuídas a cada consulta.

Tsai et al. (1990) propõem a geração de bases de dados para testes pela utilização de álgebra relacional; a análise de mutantes foi usada para avaliar parcialmente a capacidade de detecção de defeitos dos casos de teste para banco de dados.

2.5. Análise de Mutantes

A análise de mutantes ou teste de mutação foi originalmente proposta por DeMillo (1978); é um critério que utiliza a inserção de pequenas alterações no programa em teste, resultando em programas ligeiramente diferentes do original, chamados mutantes. O objetivo é encontrar um conjunto de teste T capaz de revelar as diferenças de comportamento existentes entre um programa P e seus mutantes. Quando o mutante apresenta um comportamento diferente de P, diz-se que o mutante está "morto". Se, para todo o conjunto de testes T, o mutante apresentar o mesmo comportamento de P, diz-se que o mutante está "vivo" e uma análise deverá ser feita para determinar se: (i) o mutante é equivalente a P; ou (ii) o conjunto de testes T não é bom o suficiente para matar o mutante.

A aplicação do critério de mutação requer a existência de um modelo executável: (i) dada uma entrada, a saída obtida do programa original é comparada com as saídas resultantes de cada programa mutante; e (ii) os programas mutantes são originados pela aplicação de operadores de mutação ao programa original, caracterizando o modelo de defeitos correspondentes ao programa executável.

A análise de mutantes possui eficiência baseada no efeito de acoplamento, pelo qual defeitos complexos são acoplados por meio de defeitos simples: um conjunto de dados de teste que detecta todos os defeitos simples de um programa é capaz de detectar uma alta porcentagem de defeitos complexos [Offutt, 1992]. Assim, mutantes complexos estão acoplados a mutantes simples, tal que um conjunto de teste que detecta mutantes simples também consegue detectar uma alta quantidade de mutantes complexos [Offutt, 1992].

Operadores de mutação estabelecem as regras para as alterações que podem ser aplicadas ao programa original P. A aplicação de um operador pode gerar muitos mutantes, pois P pode conter várias entidades que estão no domínio de um operador; o operador deve ser aplicado a cada uma dessas entidades [DeMillo, 1980].

O teste de mutação apóia o testador na descoberta de um conjunto de testes que seja capaz de matar todos os mutantes criados pelos operadores de mutação aplicados ao programa original P.

O *escore de mutação* é a razão entre o número de mutantes gerados e o número de mutantes mortos (exceto aqueles considerados equivalentes) [DeMillo, 1980]. Este escore, que varia entre 0 e 1, proporciona uma medida de cobertura do teste baseado em defeitos, indicando a qualidade dos casos de testes escolhidos para exercitar o programa. Calcula-se o escore de mutação EscMut (P,T) da seguinte forma (notar que $M \neq E$):

$$EscMut\ (P,T) = \frac{K}{M - E}$$

Onde:

P: programa em teste;

T: conjunto de teste;

K: quantidade de mutantes mortos;

M: total de mutantes gerados;

E: total de mutantes equivalentes.

De acordo com Woodward (1993) e Harman et al. (2000), podemos considerar três níveis de mutação: forte (*strong*), fraca (*weak*) e firme (*firm*).

No nível de mutação forte, são feitas muitas mudanças pequenas, uma por vez, para um dado programa. Uma tentativa é então feita para prover dados de teste que distinguem cada um dos mutantes do programa original [Woodward, 1993]. Este nível corresponde à análise de mutantes originalmente descrita por DeMillo (1978).

O nível de mutação fraco [Howden, 1982] foi introduzido, em parte, para fazer frente ao alto custo do nível forte. Consideremos um programa P onde C é um componente de P e que a aplicação da mutação em C produza C'. Na mutação fraca, requer-se que o teste t seja constituído de forma que execuções de C ocorram na execução de P e que, em pelo

menos uma execução de C, ocorra um resultado diferente do da versão que sofreu a mutação C'. A implementação da mutação fraca depende da especificação do conjunto de componentes e do conjunto associado ao componente que sofreu a mutação. Um componente normalmente corresponde a estruturas computacionais de um programa [Howden, 1982].

O nível de mutação firme [Woodward e Halewood, 1988] situa-se entre os dois outros níveis de mutação. Neste critério, são selecionados, no programa, pontos onde as alterações são efetuadas e pontos onde os estados do programa original e do mutante são comparados. Tal abordagem pode diminuir o custo da execução dos mutantes, uma vez que, para determinar se um mutante foi morto, são necessárias apenas execuções parciais do programa.

Devido ao número elevado de mutantes que podem ser criados a partir de uma classe de operadores, dois problemas são destacados: o alto custo computacional para executar todos os mutantes gerados; e a análise dos mutantes "vivos" quanto à equivalência com o programa original [Offutt e Untch, 2000].

Existem diversas iniciativas para diminuir o custo da utilização deste critério, tais como: Mutação Aleatória (*Randomly Selected X% Mutation)* [DeMillo et al., 1979]; Mutação Seletiva (*Selective Mutation*) [Offut et al., 1993]; Mutação Restrita [Mathur, 1991]; determinação de conjuntos essenciais de Operadores de Mutação [Offut et al., 1996; Wong e Mathur, 1995; Barbosa et al., 2001]; e redução por ordenação de mutante [Rothermel e Elbaum, 2003].

Inicialmente, o teste de mutação foi proposto para testes de unidade; diversos estudos adaptaram o critério para outros tipos de teste, tal como a mutação de interface [Delamaro et al., 2001], nos quais o critério de testes de mutação foi estendido para testes de integração e de especificações.

2.5.1. Ferramentas para o Teste de Mutação

A automatização dos testes para a análise de mutação demanda um alto custo computacional para a execução dos mutantes, devido a sua enorme quantidade. Mesmo com a utilização de técnicas de redução, o custo computacional ainda é alto e a análise dos mutantes equivalentes em muitos casos deve ser feita de forma manual. Por ser

inviável tal análise sem a criação/utilização de ferramentas automatizadas ou algoritmos para a geração e execução dos mutantes, a maior parte dos estudos feitos nessa área faz uso de ferramentas. As mais recentes são as ferramentas para uso em aplicações WEB [Mansour e Houri, 2005; Offut e Xu, 2004] e para linguagens orientadas a objetos [Ma et al., 2005].

A ferramenta de testes *Proteum* [Delamaro e Maldonado, 1996] foi desenvolvida para o auxílio de testes de unidade; *Proteum/IM* [Delamaro e Maldonado, 1999] visa à aplicação dos testes de mutação no teste de integração para programas desenvolvidos em linguagem de programação C.

A ferramenta Jester [Jester, 2008] utiliza a análise de mutantes para realizar testes unitários de classes em Java; ela possui versões para outras linguagens de programação como a Pester (para linguagem Python) e a Nester (para linguagem C#).

2.5.2. Análise de Mutantes Aplicada a SQL

Os esforços para aplicar a análise de mutantes em comandos SQL ainda são escassos.

Chan et al. (2005) propõem operadores de substituição para a mutação da SQL. A abordagem explora a informação semântica do modelo de dados conceitual, tais como as relações entre os atributos armazenados e derivados para a criação dos mutantes de acordo com a família de operadores de substituição proposta pelos autores. A aplicação dos operadores de mutação ocorre nos comandos SQL embutidos na linguagem hospedeira. Primeiramente os comandos SQL embutidos são analisados e são identificadas quais entidades, relações e restrições podem ser afetadas pela família de operadores de mutação definida, isto é, a partir da análise dos comandos e do modelo conceitual é possível identificar quais operadores de mutação podem ser aplicados nos comandos SQL e quantos mutantes serão gerados pela aplicação deste operador. A mutação utilizada para a geração e execução dos mutantes é a mutação fraca. O artigo não mostra resultados quanto à aplicabilidade dos mutantes; não apresenta uma ferramenta ou uma abordagem de como automatizar a geração, execução e avaliação dos mutantes. Os operadores de mutação definidos pelo artigo são apenas para consultas (comando SELECT) e o motivo de tal escolha não é explicitado no trabalho.

Tuya et al. (2007) propõem operadores de mutação para comandos de consultas SQL (SELECT). Os operadores propostos envolvem mutações para: cláusulas principais da SQL, condições e expressões, manipulação de valores nulos e substituição de parâmetros, tuplas e constantes. Os mutantes foram gerados a partir de comandos estáticos de consulta SQL. A execução do experimento ocorreu com apenas uma base de dados que sofria alterações conforme a aplicação dos operadores; por exemplo, para a execução dos mutantes relativos à aplicação dos operadores de manipulação de nulos a base de dados sintética foi alterada para possuir valores nulos. A detecção de mutantes equivalentes e a criação de casos de testes adicionais são realizadas de forma manual. O artigo estende também o conceito de redução de custo, utilizando a técnica de redução do número de mutantes por mutação seletiva [Mathur, 1991] e por redução do tamanho do conjunto de teste por ordenação de mutantes [Rothermel e Elbaum, 2003; Rothermel et al., 2001; Elbaum et al., 2002]. O experimento não inclui comandos de consultas SQL complexos tais como subconsulta ou utilização de múltiplas tabelas e não utiliza dados reais; isto, segundo os autores, torna o experimento incerto em relação à representatividade de cada consulta (*query*) em termos das combinações de características que podem ser encontradas no mundo real, impossibilitando, desta forma, a comparação da aplicabilidade dos operadores em sistemas reais.

2.6. Considerações Finais

Neste capítulo as iniciativas de pesquisa em teste de banco de dados e análise de mutantes são analisadas e classificadas. Observa-se que os testes de aplicações que usam a SQL têm sido abordados pela comunidade acadêmica. Algumas pesquisas inovam na área de banco de dados [Deng et al., 2004] e [Elbaum et al., 2005], com abordagens para o teste de bases de dados para aplicações WEB (aplicações em ambiente distribuído e heterogêneo acessado por meio de navegadores de mercado) sem, no entanto, considerar os comandos SQL.

A análise de mutantes para banco de dados é explorada por Chan et al. (2005), que propõem mutação baseada nas características de um modelo conceitual para o esquema de bases de dados. Tuya et al. (2007) introduzem uma família de operadores de mutação para comandos de consulta (SELECT) da SQL. No entanto, ambas as abordagens

limitam-se aos comandos de consulta e ao emprego de dados de teste sintéticos e possuem um escopo diferente deste trabalho e por este motivo não será feita uma comparação com eles.

3. Teste de Mutação Aplicado a SQL

Neste capítulo são estendidos os conceitos de teste de mutação para aplicações que utilizam a linguagem de manipulação SQL. Os conceitos envolvem aspectos inerentes a aplicações de banco de dados e, especificamente, ao emprego do teste de mutação em tais aplicações. Em adição, são estudados e definidos os cenários do teste de mutação aplicado a SQL. Tais cenários exploram as mutações fraca e forte e estabelecem estratégias para a avaliação de bases de dados de teste.

3.1. Unidade de Programa com SQL

Uma unidade de programa em teste aceita valores de entrada parametrizados e tem como uma de suas saídas os comandos de acesso às bases de dados, tais como comandos da SQL. Cada comando SQL produzido é, portanto, uma representação do procedimento de manipulação pretendido. Como conseqüência, registros nas instâncias do banco de dados são manipulados de acordo com os comandos SQL.

Para testar a aplicação de banco de dados, entretanto, é preciso lidar com essa manipulação de dados. Testadores entram com uma combinação de instâncias do banco de dados e com valores de entrada parametrizados (se existirem) aplicáveis para executar a unidade de programa [Chan et al., 2005].

Uma unidade de programa pode ter como saída comandos de manipulação da SQL, tais como SELECT, INSERT, UPDATE e DELETE. Estes comandos irão, por sua vez, selecionar ou modificar registros em suas instâncias alvos do banco de dados. No âmbito do banco de dados, a saída do programa é: a seqüência de comandos da SQL obtida pela execução do programa; ou a instância de banco de dados resultante da aplicação dessa seqüência de comandos da SQL à instância de banco de dados anterior à execução do programa [Chan et al., 2005].

Uma unidade de programa é dita *Unidade de Programa com SQL (UPS)*, se possuir comandos da SQL para o acesso à base de dados. Para a avaliação de UPSs, identificam-se duas abordagens de teste:

Ti. o resultado do teste é a saída O da UPS e o estado final DB_f (instância de saída) da base de dados;

Tii. o resultado do teste é a saída *O* da UPS e a seqüência *S* de comandos da SQL, tal que a aplicação de *S* à instância de entrada da base de dados produz o estado final DB_f da base de dados.

Ambas as abordagens podem ser aplicadas ao teste de UPSs. Na segunda abordagem (Tii), vale ressaltar que a seqüência *S* é oriunda da execução da UPS a partir do dado de teste: valores de entrada e instância de banco de dados.

A noção de resultado, com respeito ao teste de Unidades de Programa com SQL (UPSs), pode ser estendida ao teste de programas. Em ambos os casos, a execução isolada da seqüência *S* de comandos da SQL (*script SQL*), que é oriunda da execução de um programa *P* ou de uma unidade do programa *U* (escritos em linguagem hospedeira com SQL embutida), resulta na mesma instância de banco de dados obtida a partir da execução de *P* ou de *U*.

As definições da próxima seção são estabelecidas para o teste de programas. Contudo, os conceitos explorados são aplicáveis ao teste de unidade e ao teste de programas.

3.2. Definições

Definição 1: ***Dado de teste*** para a execução de programa *P*, tal que *P* representa uma aplicação de banco de dados, é definido pelo conjunto *ID = {parâmetros de entrada, instância* DB_i *da base de dados}*.

Definição 2: ***Caso de teste*** para a execução de programa *P*, tal que *P* representa uma aplicação de banco de dados, é definido pelo conjunto *TC = {parâmetros de entrada, instância* DB_i *da base de dados, S}* ou *TC = {parâmetros de entrada, instância* DB_i *da base de dados, instância* DB_o *de banco de dados}*, tal que DB_o é obtida aplicando-se a seqüência *S* de comandos SQL à instância DB_i. Esta definição presume que *P* não possui valores de saída.

Definição 3: ***Conjunto de operadores de mutação*** é definido pelo conjunto $MO = \{mo_1, mo_2,..., mo_x\}$, composto por operadores de mutação aplicáveis à seqüência *S* de comandos SQL, tal que $x > 0$.

Definição 4: Aplicando-se o dado de teste *ID* ao programa *P*, pode-se obter o ***conjunto de mutantes*** definido por $M = \{mo_{11}, mo_{12},..., mo_{i1}, mo_{i2},..., mo_{ij},..., mox_1, mox_2,...\}$ oriundo da seqüência *S*, tal que: (i) mo_{ij} é o j-ésimo mutante obtido a partir do operador de mutação $mo_i \in MO$; e (ii) $|M| \geq x$, sendo *x* a cardinalidade do conjunto *MO*.

Definição 5: A partir do conjunto *M*, é derivado o ***conjunto de mutantes plausível*** *M**, tal que $|M^*| \leq |M|$. Um mutante é plausível se for sintaticamente válido.

Ao aplicar um dado de teste *ID = {parâmetros de entrada, instância DBi da base de dados}* ao programa *P*, é obtida a seqüência *S* de comandos SQL. Da seqüência *S*, é derivado o conjunto $MO = \{mo_1, mo_2,..., mo_x\}$ de operadores de mutação, do qual se constroem *x* mutantes (tal que $x > 0$). Aplica-se DB_i a cada $mo_i \in MO$ e determina-se o número *y* de mutantes mortos, $y \leq |MO|$. O *escore de mutação* EscMut determina o percentual de mutantes mortos dentre os mutantes gerados. Se $EscMut=1$, significa que todos os mutantes foram mortos (situação ideal). Dado o conjunto de teste C, deve-se selecionar o subconjunto *C** de *C* que é composto dos casos de teste de *C* que resultaram nos maiores valores para *EscMut*.

Em síntese, a capacidade para revelar defeitos da base de dados reflete-se diretamente na qualidade dos casos de testes, pois a base de dados faz parte dos dados de entrada, os quais determinam a forma de exercitar a aplicação. Vale ressaltar que o foco primário é avaliar a habilidade que um conjunto de casos de testes possui de revelar defeitos em uma aplicação de banco de dados, ou até mesmo em uma seqüência de comandos SQL (como o caso da linguagem PL-SQL), conforme definido na próxima seção.

3.3. Cenários

A aplicação do teste de mutação em aplicações que utilizam a SQL é realizada pela definição de cenários para a derivação de mutantes e para a execução da atividade de teste propriamente dita. Para cada cenário, é pertinente definir o objeto em teste e o nível de mutação empregado (mutação fraca ou mutação forte).

A partir dos conceitos apresentados nas Seções 3.1 e 3.2, definimos dois cenários para a aplicação de operadores de mutação na linguagem SQL. No primeiro cenário a mutação é feita nos comandos SQL embutidos na linguagem hospedeira; no segundo ela é feita em comandos SQL isolados; os cenários são discutidos nas seções 3.3.1 e 3.3.2.

3.3.1. Cenário 1

Neste cenário, seqüências SQL são analisadas como um átomo, caracterizando o aspecto de Mutação Forte [Woodward, 1993] para o teste. Um exemplo deste cenário seria o teste de uma aplicação que possui comandos SQL embutidos nela, sendo estes comandos criados dinâmicamente pela linguagem hospedeira durante a execução do software ou fixos na linguagem.

Uma seqüência *S'* de comandos SQL é obtida pela aplicação dos dados de teste *ID'* (dados de entrada e instância inicial da base de dados) a uma UPS. A seqüência *S'* e um conjunto de mutantes, os quais foram gerados a partir de *S'*, são executados usando a instância de entrada da base de dados em *ID'*. A instância de banco de dados produzida pela execução de *S'* é comparada com as instâncias de banco de dados oriundas das execuções dos mutantes de *S'*. O resultado dessa comparação produz o escore de mutação *EscMut*, referente ao dado de teste *ID'*.

O escore de mutação EscMut obtido permite comparar o dado de teste *ID'* em relação a outros dados de teste para o teste da UPS. O foco da avaliação é a comparação de dados de teste, onde o teste de mutação dará suporte à análise da eficiência dos dados de teste. Durante a análise poderemos considerar diversos aspectos, tais como a capacidade de revelar classes de defeitos e a cobertura de comandos SQL a serem exercitados. Vale ressaltar que qualquer variação na instância de entrada da base de dados caracterizará outro dado de teste, denominado *ID''* (cuja aplicação à UPS produz a seqüência *S''* de comandos SQL, possivelmente distinta de *S'*), o qual é passível de comparação com *ID'*.

3.3.2. Cenário 2

Neste cenário, seqüências de comandos SQL são também consideradas como objetos do teste, mas são analisadas em termos de seus componentes, caracterizando o aspecto de Mutação Fraca [Woodward, 1993] para o teste. Um exemplo deste cenário é o teste de procedimentos ou de códigos em linguagem PL-SQL onde as sequência de comandos SQL não estão embutidas em uma linguagem hospedeira.

Geralmente, um componente da seqüência é constituído por um único comando de acesso à base de dados, tais como SELECT e INSERT, mas pode ser composto por um grupo de comandos que denotem algum aspecto particular (por exemplo, um subcaminho). Os mutantes são gerados a partir da aplicação de operadores de mutação aos comandos do componente em questão.

Vale ressaltar que esta abordagem é pertinente à comparação de instâncias de base de dados para: (i) o teste de uma UPS particular, pois apóia a seleção de bases potencialmente reveladoras de defeito pela análise de componentes de seqüências SQL; (ii) o teste de seqüências SQL oriundas de outras fontes, tais como *scripts* SQL de uso geral, *triggers* e *stored procedures*.

3.4. Considerações Finais

Este capítulo apresenta conceitos pertinentes ao emprego de teste de mutação para aplicações de banco de dados que usam a SQL. As definições exploram a noção de resultado do teste e dão suporte aos experimentos descritos no Capítulo 5.

Os aspectos explorados são aplicáveis ao teste de unidades de programa e ao teste de programas. Foram definidos dois cenários de teste, atrelados à mutação forte e à mutação fraca. Em ambos os cenários, o foco da avaliação é o conjunto de teste, permitindo a comparação da qualidade das bases de dados com respeito à sua relativa habilidade de detecção de defeitos.

No Cenário 1, a aplicação da mutação dá suporte à avaliação da eficácia dos casos de testes em linguagem hospedeira, pois neste cenário o objeto em teste é uma seqüência de comandos SQL obtidos a partir da execução do programa (ou da unidade de programa) em teste. Assim, os parâmetros de entrada e a instância inicial da base de dados serão

avaliados. No Cenário 2, a análise é desvinculada de um programa (ou unidade de programa), e o objeto em teste é um comando SQL ou um componente de uma seqüência de comandos SQL.

4. Operadores de Mutação em SQL

A definição dos operadores de mutação em SQL visa ao estudo da capacidade das bases de dados em revelar defeitos, estejam eles no esquema da base de dados ou nos comandos SQL embutidos dentro de linguagem hospedeira que acessam o banco de dados.

Os operadores de mutação propostos nesse trabalho visam à cobertura da maioria dos comandos (cuja sintaxe é passível de mutação) do padrão SQL 3 [ANSI, 2003]. Embora alguns estudos, como em Ponighaus (1995), mostrem que o comando SELECT é mais utilizado nas aplicações comerciais, não há registros sobre a relevância dos demais comandos. Essa constatação não significa que inexistam defeitos nos demais comandos da linguagem SQL e nem que os defeitos encontrados nos demais comandos sejam poucos ou não sejam tão graves quanto os defeitos relativos aos comandos de consulta. São propostos operadores para as quatro categorias de comandos especificadas no padrão SQL 3 (DML, DCL, DDL e DQL), visando a uma maior cobertura dos comandos de manipulação da SQL e buscando atingir, desta forma, a caracterização da maioria dos defeitos relacionados a SQL.

Este capítulo introduz os operadores de mutação definidos para a SQL. O Apêndice B estende a especificação dos operadores, apresentando para cada operador de mutação: um comando da linguagem SQL; possíveis mutantes para esse comando; e esclarecimentos com respeito à aplicação do operador de mutação.

4.1. Definição das Categorias

Os operadores de mutação são divididos em cinco categorias. As categorias são definidas conforme a funcionalidade dos comandos SQL que sofrerão alterações, exceto pela categoria de “Miscelânea”, que possui operadores que podem ser utilizados em diversos comandos com diversas funcionalidades. Alguns exemplos são: um operador de mutação que é definido para um operador matemático, tal como “+”, será colocado na categoria de operadores de mutação para operadores de SQL; e um operador de mutação que é definido para alterar o parâmetro de entrada ou saída de uma *stored procedure*, tais como “*in*” e “*out*”, é definido na categoria de operadores de mutação para funções e procedimentos. Todos os operadores definidos para diversos comandos básicos da SQL,

como “INSERT” e “DELETE”, são definidos na categoria de miscelânea, pois podem caracterizar tipos de defeitos diferentes, dependendo da funcionalidade do comando ao qual foram aplicados.

As categorias de operadores de mutação definidas são:

- operadores de mutação para operadores de SQL;
- operadores de mutação miscelânea;
- operadores de mutação de fluxo de dados;
- operadores de mutação para controle de transação;
- operadores de mutação para funções e procedimentos.

No Apêndice B encontram-se exemplos da aplicação dos operadores de mutação.

Os defeitos que os operadores visam a caracterizar foram baseados na definição de classes de defeitos de Leitão-Junior et al. (2005).

4.2. Operadores de Mutação para Operadores de SQL

Os operadores de mutação para operadores de SQL visam a caracterizar os defeitos que normalmente são criados pelo programador pelo uso incorreto dos operadores de SQL.

Foram definidos operadores de mutação para os cinco tipos mais comuns em SQL: são: operadores matemáticos, operadores de comparação, operadores lógicos, operadores conjuntivos e operadores de negação. Estes operadores de mutação podem caracterizar os seguintes defeitos:

- desvios de fluxos;
- excesso ou ausência de repetições em estruturas condicionais;
- número de tuplas incorreto na seleção, inserção, deleção ou atualização; por exemplo, caso um atributo seja excluído indevidamente em uma seleção, o número de tuplas poderá ser diferente do esperado.
- inserção, atualização ou exclusão indevida de dados;
- visualização ou cálculo de dados de forma incorreta; por exemplo: na somatória de dois campos, se o operador estiver incorreto será visualizado um valor diferente do esperado.

A Tabela 4-1 ilustra os seis operadores desta categoria, sendo que a coluna Nome apresenta o identificador e nome do operador; na mesma linha faz-se uma breve descrição de sua aplicação.

Tabela 4-1 - Operadores de Mutação para Operadores de SQL

Nome	Descrição
tOpMt Troca de Operador Matemático	Pode ser aplicado em qualquer parte do comando SQL que permita a sua aplicação; por exemplo, na lista de atributos de um comando SELECT. O operador realiza 12 permutas entre os principais operadores matemáticos (+, -, *, /).
tOpCp Troca de Operador de Comparação	Pode ser aplicado em qualquer parte do comando SQL que permita a sua aplicação, embora ele seja mais utilizado em um predicado usado para seleção de tuplas, como no caso de uma cláusula WHERE. Este operador realiza 30 permutas ente os operadores de comparação (=, <>, >, <. >=, <=).
tOpCj Troca de Operador Conjuntivo	Pode ser aplicado em qualquer parte do comando SQL que permita a sua aplicação, embora ele seja mais utilizado em um predicado usado para seleção de tuplas, como no caso de uma cláusula WHERE.
tOpLg Troca de Operador Lógico	Normalmente aplicado em comandos de seleção. Ele possui diversas permutas, uma vez que conta-se com uma grande variedade de operadores lógicos. Este trabalho limita-se aos operadores lógicos mais utilizados e que possuem sintaxe igual à sua troca.
iNot Inserção de Operador de Negação	Pode ser utilizado em diversas condições e em conjunto com outros operadores, como os operadores lógicos IS NULL, BETWEEN, IN, LIKE, EXISTS, UNIQUE, ALL, ANY e EQUAL.
rNot Retirada de Operador de Negação	Pode ser utilizado em diversas condições e em conjunto com outros operadores.

4.3. Operadores de Mutação de Miscelânea

Os operadores do tipo miscelânea são caracterizados pela sua aplicação em diversos comandos com diversas funcionalidades. Todos os operadores que são definidos para diversos comandos bases da SQL como "*insert*" e "*delete*" estão inseridos na categoria de miscelânea, pois podem caracterizar tipos de defeitos diferentes.

Estes operadores de mutação podem caracterizar os seguintes defeitos:

- inserção, atualização ou deleção errônea de persistência de dados;
- visualização ou cálculo errôneo de dados;
- definição errônea de atributos e variáveis;
- número de tuplas incorreto.

Para caracterizar estes defeitos são criados 15 operadores, conforme a Tabela 4-2:

Tabela 4-2- Operadores de Mutação de Miscelânea

Nome	Descrição
tPoAt Troca de Posição de Atributo	A aplicação dar-se-á em uma lista ordenada de atributos, em que um atributo troca de posição com um outro da mesma lista.
rAtr Retirada de Atributo	A aplicação dar-se-á em uma lista ordenada de atributos ou numa lista de atributos na qual serão feitas atribuições de valores (por exemplo, em um UPDATE) e pode ser utilizado em vários comandos.
iAtr Inserção de Atributo	A aplicação dar-se-á em uma lista ordenada de atributos ou numa lista de atributos na qual serão feitas atribuições de valores (por exemplo, em um UPDATE).
tAt Troca de Atributo	A troca pode ser feita por qualquer atributo existente no banco de dados e presente em outros comandos SQL da mesma aplicação.
tPoVr Troca de Posição de Valor	A aplicação é feita de diversas formas, como em uma lista ordenada de valores, em que um valor troca de posição com um outro da mesma lista ou em uma cláusula de condição e pode ser utilizados em vários comandos.
tVr Troca de Valor	A aplicação é feita pela troca de um determinado tipo de valor por outro diferente, por exemplo, a troca de um NULL por um valor alfanumérico ou de um valor numérico por um alfanumérico, também pode ser utilizados em vários comandos.
tTpVar Troca de Tipo de Variável	A aplicação dar-se-á na declaração da variável; a troca pode ser feita por qualquer tipo de variável existente no banco de dados e já declarada em outros comandos SQL da mesma aplicação.
tNmTb Troca de Nome da Tabela	A troca pode ser feita por qualquer nome de tabela existente no banco de dados e presente em outros comandos SQL da mesma aplicação.
tNmRole Troca de Nome de Role	A troca pode ser feita por qualquer nome de role existente no banco de dados e presente em outros comandos SQL da mesma aplicação.
iRole Inserção de Role	A inserção pode ser feita por qualquer role existente no banco de dados e presente em outros comandos SQL da mesma aplicação.
rRole Retirada de Role	A retirada pode ser feita em qualquer role presente no comando.
tNmCursor Troca de Nome do Cursor	A troca pode ser feita por qualquer cursor existente no banco de dados e/ou presente em outros comandos SQL da mesma aplicação.
tFuAg Troca de Função de Agregação	A aplicação é feita de diversas formas como em uma lista ordenada de atributos ou em uma cláusula WHERE. As trocas entre funções de agregação são muitas; as mais utilizadas são SUM, AVG, COUNT, MAX e MIN e pode ser utilizados em diversos comandos.
tInSec Troca de Intersecção	É aplicado às intersecções UNION, INTERSECT e EXCEPT que podem ser utilizadas tanto entre dois comandos distintos como dentro de um SELECT. É indiferente para este trabalho em qual das duas formas as intersecções são trocadas, permitindo-se a troca entre qualquer uma delas.
tJoin Troca de Join	É utilizado nos comandos do tipo SELECT.

4.4. Operadores de Mutação para Fluxo de Dados

Esta categoria visa a alterar o fluxo de dados dos comandos SQL por meio das modificações em estruturas condicionais e de repetição. Os operadores definidos têm por objetivo caracterizar os defeitos que são normalmente criados pelo programador, quando

este utiliza as estruturas condicionais de maneira incorreta; e pelos enganos cometidos nas condições das estruturas. Estes operadores de mutação podem caracterizar os seguintes defeitos:

- execução incorreta de comandos, como a inversão de códigos dentro de uma condição.
- ausência/excesso de execução de comandos, como a falta de um comando em uma condição ou até mesmo um comando de um bloco de repetição que deveria ser externo ao bloco.

Para caracterizar estes defeitos são definidos seis operadores conforme descrição na Tabela 4-3:

Tabela 4-3 - Operadores de Mutação para Fluxo de Dados

Nome	Descrição
tBlCmEstRe Troca de bloco de comandos	A aplicação dar-se-á pela troca de um bloco de comandos nas estrutras de condição e repetição.
rCmBlRep Retirada de comando do bloco de repetição/condição	A aplicação dar-se-á pela retirada de um bloco de comandos nas estrutras de condição e repetição.
iCmBlRep Inserção de comando do bloco de repetição/condição	A aplicação dar-se-á pela inserção de um bloco de comandos nas estrutras de condição e repetição.
tPosLeav Troca de posição do leave no bloco de comandos	A aplicação dar-se-á pela troca de posição do LEAVE em um bloco de comandos..
rLeave Retirada de Leave	A aplicação dar-se-á pela retirada do LEAVE em um bloco de comandos.
iLeave Inserção de Leave	A aplicação dar-se-á pela inserção do LEAVE em um bloco de comandos.

4.5. Operadores de Mutação para Controle de Transações.

Esta categoria visa à criação de operadores para o controle das transações e para as permissões de acesso por parte dos usuários. Os operadores definidos visam a caracterizar os enganos que normalmente são cometidos pelos usuários no momento da execução dos comandos e na definição do esquema de permissões de acesso. Estes operadores de mutação podem caracterizar as seguintes falhas:

- perda de dados;
- execução de comandos não autorizados por determinados usuários;
- ausência/excesso de permissão de execução de comandos para usuários.

Para caracterizar estes defeitos são definidos doze operadores, conforme descrição da Tabela 4-4.

Tabela 4-4 - Operadores de Mutação para Controle de Transações

Nome	Descrição
ICM Inserção de COMMIT	A aplicação dar-se-á pela inserção do COMMIT em/entre um bloco de comandos.
rCm Retirada de COMMIT	A aplicação dar-se-á pela retirada do COMMIT em/entre um bloco de comandos.
iRb Inserção de ROLLBACK	A aplicação dar-se-á pela inserção do ROLLBACK em/entre um bloco de comandos.
rRb Retirada de ROLLBACK	A aplicação dar-se-á pela retirada do LEAVE em/entre um bloco de comandos.
tCmRb Troca de COMMIT por ROLLBACK	Atua entre comandos, isto é, a substituição do comando inteiro é feita entre um comando de manipulação SQL e outro.
tRbCm Troca de ROLLBACK por COMMIT	Atua entre comandos, isto é, a substituição do comando inteiro é feita entre um comando de manipulação SQL e outro.
tNmSP Troca de Nome do SAVEPOINT	A troca pode ser feita por qualquer SAVEPOINT existente no banco de dados e/ou presente em outros comandos SQL da mesma aplicação.
tPerm Troca de Permissão	É ultilizado para a troca de permissão de diversos comandos.
tPriv Troca de Privilégio	É utilizado nos comandos GRANT e REVOKE. Este operador pode efetuar a troca entre quaisquer tipos de privilégio. Os principais privilégios são SELECT, DELETE, UPDATE, INSERT, USAGE e ALL PRIVELEGE.
tGrRe Troca de GRANT por REVOKE	A aplicação dar-se-á pela troca do comando GRANT por REVOKE.
tReGr Troca de REVOKE por GRANT	A aplicação dar-se-á pela troca do comando REVOKE por GRANT.
tNmUsr Troca de Nome do Usuário	A troca pode ser feita por qualquer nome de usuário existente no banco de dados e/ou presente em outros comandos SQL da mesma aplicação.

4.6. Operadores de Mutação para Funções, Procedimentos e *Triggers*

Esta categoria visa à criação de operadores para as informações de controle de funções, procedimentos e *triggers*. Os operadores definidos visam à caracterização do uso incorreto de valores e chamadas externas. Estes operadores de mutação podem caracterizar os seguintes defeitos:

- execução errônea de função, *trigger*, *view* ou procedimento;
- retorno de valores incorretos;
- ausência de retorno de valores;
- execução de *trigger* em evento incorreto.

Para caracterizar estes defeitos são definidos cinco operadores, conforme descrição da Tabela 4-5.:

Tabela 4-5 - Operadores de Mutação para Funções, Procedimentos e Triggers

Nome	Descrição
tNm Troca de Nome da função, procedimento, VIEW ou TRIGGER	Visa à caracterização de um erro muito comum dos programadores, que é a chamada incorreta de uma função, procedimento, view ou trigger durante a execução de algum procedimento. Este operador pode ser aplicado em diversos pontos e comandos da SQL.
tPoReFu Troca de Posição de Retorno da Função	A aplicação dar-se-á pela troca de posição do retorno de uma função.
tReFu Retirada de Retorno da Função	A aplicação dar-se-á pela retirada do retorno de uma função.
tPaPro Troca de Parâmetros da PROCEDURE (in por out ou out por in)	A aplicação dar-se-á pela permuta de IN e OUT de uma procedure.
tEv Troca de Evento na TRIGGER	Pode efetuar troca entre quaisquer eventos aplicáveis em uma trigger.

4.7. Considerações Finais

Neste capítulo são apresentados os operadores de mutação definidos para a aplicação da análise de mutantes na SQL. São definidas 5 categorias de operadores:

- Operadores de Mutação para Operadores de SQL: caracterização dos defeitos criados pelo uso incorreto dos operadores da SQL;
- Operadores de Mutação de Miscelânea: caracterização dos defeitos criados pelo uso incorreto das partes dos comandos SQL mais utilizados durante a criação de um comando SQL;
- Operadores de Mutação para Fluxo de Dados: caracterização dos defeitos criados pelo uso incorreto das estruturas condicionais e de repetição;
- Operadores de Mutação para Controle de Transações: caracterização dos defeitos criados pela manipulação incorreta dos comandos e uso incorreto das definições do esquema de permissões de acesso;
- Operadores de mutação para Funções, Procedimentos e *Triggers*: caracterização do uso incorreto de valores e chamadas externas.

5. Experimentos

Neste capítulo apresentam-se três experimentos conduzidos com o objetivo de investigar a aplicabilidade e a habilidade de detecção de defeitos dos operadores de mutação apresentados. Os experimentos foram realizados dentro dos cenários propostos no Capítulo 3.

São inseridos defeitos nos quatro principais comandos de manipulação da SQL: INSERT, SELECT, UPDATE e DELETE. Estes comandos foram escolhidos por serem os mais comuns em sistemas comerciais, nos quais os experimentos são realizados. Os operadores de mutação aplicados foram os operadores para SQL (Seção 4.2) e de miscelânea (Seção 4.3), por representarem os operadores mais aplicáveis aos comandos INSERT, SELECT, UPDATE e DELETE, isto é, os operadores que melhor se encaixam sintaticamente às construções desses comandos. O motivo da escolha dos operadores de mutação dentro dessas categorias é detalhado em cada experimento.

O número elevado de mutantes criados a partir de uma classe de operadores torna necessário o desenvolvimento e o uso de suporte automatizado à criação, à execução e à análise de mutantes (mortos, vivos e equivalentes). Todos os experimentos foram apoiados por uma ferramenta de mutação desenvolvida para dar suporte às atividades de geração, execução e análise de mutantes. Como conseqüência, um objetivo secundário foi validar a adequação da ferramenta aos propósitos de geração e execução de mutantes e análise de resultados. A ferramenta é descrita no Apêndice A.

Todos os experimentos foram realizados em uma versão do aplicativo (software em testes) que possuiam defeitos já identificados por testes funcionais. Desta forma, é possível comparar se as bases de dados e os operadores de mutação foram capazes de revelar tais defeitos. A análise desta comparação é feita no Apêndice C.

5.1. Programas em Teste

Os experimentos foram executados em três sistemas reais, que se distinguem entre si nos seguintes quesitos: equipe desenvolvedora, área de negócio, quantidade de usuários e bases de dados.

Todos os sistemas utilizam a plataforma Windows e o Sistema Gerenciador de Banco de Dados MS SQL Server 2000.

Os sistemas estão em produção, apresentando atividades de manutenção e de evolução pelo desenvolvimento de novas funcionalidades. A descrição de cada um dos sistemas, bem como as bases de dados escolhidas para a execução do experimento, são detalhadas nas próximas seções.

5.2. Seleção dos Operadores

Para a seleção dos operadores de mutação nos três experimentos duas considerações foram levadas em conta:

- Os comandos para os quais os testes foram realizados são SELECT, INSERT, DELETE e UPDATE por serem os mais comuns em sistemas comerciais, isto é, não são aplicados testes para os demais comandos. Conseqüentemente apenas os operadores que possam ser aplicados aos comandos escolhidos foram selecionados.
- Experiência do testador envolvido na execução dos experimentos em relação aos tipos de erros mais freqüentemente cometidos pela equipe de desenvolvimento. A experiência do testador também é utilizada em alguns casos de redução da aplicação dos operadores.

5.3. Procodimento de Realização dos Experimentos

O testador forneceu uma combinação de instâncias do banco de dados (base de produção ou de testes) e valores de entrada parametrizados (dados de entrada dos casos de testes projetados pelo testador). A unidade de programa teve como saída comandos de manipulação da SQL que, por sua vez, selecionam ou modificam registros em suas instâncias alvos do BD. A ferramenta armazena informações destas alterações e modificações juntamente com os comandos executados. Algumas das informações armazenadas são: a quantidade de persistências afetadas, ordens das tuplas, e quantidade de leituras e escritas. Os comandos que não sofrem mutação (que não são SELECT, INSERT, DELETE e UPDATE) também foram registrados, pois a ferramenta precisa executá-los para garantir a integridade do experimento.

5.4. Experimento Controle

O experimento ocorreu no ambiente proposto pelo Cenário 2 (Seção 3.2.2), utilizando as seguintes bases de dados: (i) Base de Dados 1 – *Adventure Works,* base de dados exemplo do sistema MS SQL Server (ii) Base de Dados 2 – Base 1 com redução de 50% das persistências de dados. O procedimento foi o mesmo para as duas bases.

5.4.1. Contextualização

O experimento Controle utiliza um software cuja principal função é exemplificar padrões de desenvolvimento como telas, componentes e relatórios. Este sistema é utilizado internamente em uma empresa multinacional de tecnologia de informação. A base de dados utilizada neste sistema é a *Adventure Works*, uma base de exemplo do *MS SQL Server* 2005.

Os casos de testes utilizados neste experimento foram criados por um analista de testes com perfil sênior (o profissional é experiente e tem alto nível de qualificação em sua área).

Para este experimento foram utilizados 42 comandos de manipulação SQL retirados de *procedures* do sistema em teste.

Para a execução deste experimento foram consideradas as seguintes bases de dados:

Base de Dados 1 – *Adventure Works*: Base de dados disponível no banco de dados de exemplo, sem nenhuma alteração.

Base de Dados 2 – *Adventure Works* reduzido: a Base de Dados 1 sofreu uma redução de persistências de dados de 50%. Esta redução foi feita por meio de uma ferramenta proprietária (File Aid C/S da Compuware) que busca reduzir o volume de dados das bases sem prejuízo da qualidade dos dados de testes.

5.4.2. Seleção dos Operadores para o Experimento Controle

A Tabela 5-1 mostra os operadores selecionados para o Experimento Controle, levando em consideração os critérios definidos na Seção 5.2 . São indicados o tipo de operador selecionado, as permutas utilizadas e são feitas as observações em relação ao uso do operador.

Tabela 5-1 - Operadores Selecionados para o Experimento Controle.

Operador	Permutas	Observações
tInSec	tInEx, tInUn, tExIn, tExUn.	Foram aplicadas apenas as trocas de INTERCEPT e EXCEPT pelo fato de as características do software em testes apontá-los como os principais operadores deste cenário. Todas as trocas possíveis para estes operadores foram utilizadas.
tOpMt	tAdSu	Todas as trocas possíveis foram aplicadas.
tOpCp	tIgDes, tMaIg, tMaIg, tMeIg e tMeMa.	Foram aplicadas apenas as trocas de igualdade, maior e menor pelo fato de as características do software em teste apontá-los como principais operadores deste cenário. Todas as trocas possíveis para estes operadores foram utilizadas.
tOpCj	tAndOr, tOrAnd	Todas as trocas possíveis foram aplicadas.
iNot	N.A.	Todas as trocas possíveis foram aplicadas.
rNot	N.A.	Todas as trocas possíveis foram aplicadas.
tPoAt	N.A.	Aplicado para 10% dos atributos.
tAt	N.A.	Aplicado para 10% dos atributos.
tVr	N.A.	Aplicado para 10% dos valores em condições
tPoVr	N.A.	Aplicado para 10% dos atributos em atualizações.
tNmTb	N.A.	Aplicado para 10% das tabelas.
tJoin	tInLe, tInRi, tInFu, ttInCr	Foram aplicadas apenas as trocas de INNER JOIN pelo fato de as características do software em teste apontá-lo como o principal tipo de JOIN deste cenário. Todas as trocas possíveis para este JOIN foram utilizadas.
tFuAg	Troca de Função de Agregação	Foram aplicadas trocas de AVG e MIN por MAX, COUNT, MIN, AVG e SUM.
tOpLg	tAllAny, tAnyAll, tExUn	Todas as trocas possíveis foram aplicadas.

Devido ao grande número de mutantes gerados para este experimento alguns operadores foram aplicados em apenas dez por cento dos casos. Baseando-se no Experimento Equipamentos (descrito na Seção 5.5) foi tomado como premissa que a redução dos operadores não alterará os resultados deste experimento em relação à eficácia do teste. O critério de seleção dos mutantes foi uma escolha feita pelo testador que, com sua experiência, apontou quais seriam os casos mais interessantes de se aplicar aos operadores de mutação, levando em consideração as características do software.

Após a execução dos mutantes as informações sobre as alterações e seleções feitas são armazenadas e comparadas com os comandos de manipulação original. A ferramenta avaliou automaticamente os mutantes e decidiu se eles estavam mortos ou vivos. No caso de mutantes vivos a ferramenta mostrou os resultados tanto do comando original quanto do mutante, para que o testador decida se o mutante é equivalente ou não. A ferramenta permite que o mutante seja re-executado isoladamente para sua avaliação.

O escore de mutação foi então gerado para cada caso de teste executado no experimento.

5.4.3. Resultados

O experimento foi gerado com 42 comandos de manipulação SQL, os comandos são os mesmos para ambas as bases. Neste cenário não existe variação na quantidade de comandos SQL, uma vez que os comandos de manipulação não são provenientes de uma linguagem hospedeira, tornando-os invariáveis. Os resultados gerados para as duas bases de dados podem ser vistos no Apêndice C deste trabalho.

O gráfico na Figura 5-1 ilustra os valores dos escores de mutação para o Experimento Controle. Na abscissa estão indicados os 42 casos de testes executados para este experimento; na ordenada é mostrado o escore de mutação alcançado para cada caso de teste e as barras representam os resultados alcançados para cada uma das bases de dados utilizadas no experimento.

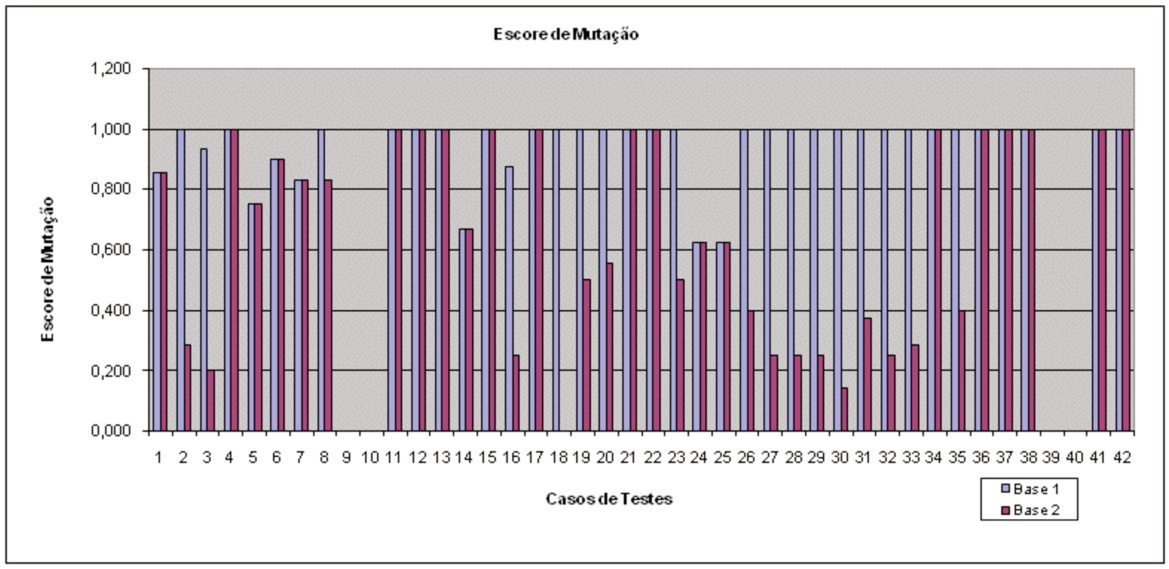

Figura 5-1 - Diferença entre os Escores de Mutação no Experimento Controle

Em relação ao estudo das bases de dados podemos notar que os valores do escore de mutação alcançados pela Base de Dados 1 são maiores do que os alcançados pela Base de Dados 2 em 16 dos 42 comandos em teste, mostrando que, para o conjunto de operadores de mutação aplicados nos comandos de manipulação SQL, a Base de Dados 1 possui maior capacidade relativa em revelar defeitos.

O gráfico da Figura 5-2 ilustra a diferença do escore de mutação entre as duas bases de dados; quando o valor da ordenada (escore) for positivo entendemos que a Base de Dados 1 obteve um escore de mutação melhor (mais próximo de 1) do que a Base de Dados 2; quando o valor for negativo, entendemos que a Base de Dados 2 obteve um escore de mutação melhor (mais próximo de 1) do que a Base de Dados 1.

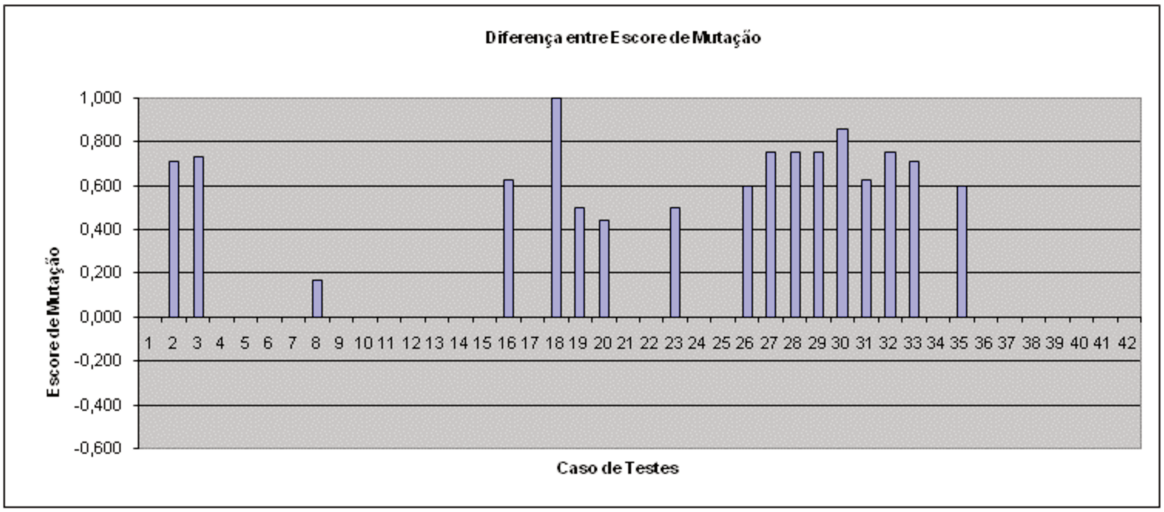

Figura 5-2 - Diferença entre os Escores de Mutação no Experimento Controle

A Base de Dados 1 consumiu aproximadamente 60 horas de processamento para a execução dos mutantes e a Base de Dados 2 consumiu aproximadamente 30 horas.

A redução da base de dados mostrou uma grande perda na qualidade dos seus dados, uma vez que a Base de Dados 1 obteve melhores escores de mutação em 16 comandos de manipulação (38%).

O operador de troca de *join* conseguiu atingir escores de mutação diferentes de zero, demonstrando que a base de dados é forte o suficiente para revelar este tipo de defeito.

Novamente, os mutantes que aumentavam a quantidade de campos selecionados como no caso dos operadores *tOpCp* e *tJoin*, consumiram muito tempo para a sua execução.

As trocas de funções de agregação (que deveriam obter altos escores de mutação por representarem, em muitos casos, mudanças bruscas no cálculo de valores) não atingiram escores altos para nenhuma das duas bases. O motivo para tal resultado é a ausência de

persistências de dados que satisfaçam as condições dos comandos nos quais as funções de agregações estavam inseridas.

5.5. Experimento Equipamentos

O experimento ocorreu no ambiente proposto pelo Cenário 1 (Seção 3.2.1), utilizando as seguintes bases de dados: (i) Base 1 – Ambiente de Testes (ii) Base 2 – Ambiente de Produção.

5.5.1. Contextualização

A aplicação em teste é dedicada à gestão de equipamentos de informática.

As principais funcionalidades deste software são:

- controle de entrada e saída de equipamentos de informática (software e hardware);
- controle de entrada/saída interna e externa de materiais;
- controle de custo, amortização, emissão de notas, faturas e outras características relacionadas aos equipamentos e áreas responsáveis.

O software foi desenvolvido sob demanda para uma empresa de médio porte atuante no comércio varejista. O sistema está implantado e em funcionamento em 25 filiais, sendo utilizado por aproximadamente quatro usuários por filial.

A equipe de desenvolvimento para a produção do software possui um perfil pleno (não é composta de profissionais inexperientes) e o analista de testes responsável pela geração dos casos de testes possui perfil Junior (profissionai inexperiente).

Para este experimento foram utilizados 20 casos de testes projetados pelo analista de testes da equipe desenvolvedora do software. Os casos de testes projetados são funcionais e utilizam um cenário da aplicação em teste (entende-se por um cenário um caso de uso), visando ao teste de regras de negócio e de funcionamento básico (Inclusão. Alteração, Navegação e Exclusão).

A base de dados do sistema é unificada para todas as filiais. Para a execução deste experimento foram consideradas as seguintes bases de dados:

Base 1 - Ambiente de Testes: Base de Dados utilizada por desenvolvedores e testadores para executar testes unitários, funcionais, de integração e de sistemas. Esta base possui uma quantidade de dados muito menor do que a da base de produção.

Base 2 - Ambiente de Produção: Base de Dados da produção, sem nenhuma alteração por parte dos testadores ou desenvolvedores. As persistências de dados são reais.

5.5.2. Seleção dos Operadores para o Experimento Equipamentos

Os operadores de mutação selecionados para este experimento foram aplicados aos comandos SQL coletados pela ferramenta e os mutantes gerados são executados.

A Tabela 5-2 mostra os operadores selecionados, conforme os critérios definidos para tal escolha (Seção 5.2). São indicados o tipo de operador selecionado, as permutas utilizadas e são feitas observações em relação ao uso do operador.

Tabela 5-2 - Operadores Selecionados para o Experimento Equipamentos.

Operador	Permutas	Observações
TOpMt	tMuAd, tMuSu, tMuDi	Foram aplicadas apenas as trocas de multiplicação pelo fato das características do software em testes apontarem a multiplicação como o principal operador deste cenário. Todas as trocas possíveis para este operador foram utilizadas.
TOpCp	tIgDes, tIgMa, tIgMe, tIgMag, tIgMeg, tMaIg, tMaDes, tMaMe, tMaMag, tMaMeg, tMeIg, tMeDes, tMeMa, tMeMag, tMeMeg,	Foram aplicadas apenas as trocas de igualdade, maior e menor pelo fato das características do software em teste apontá-los como principais operadores deste cenário. Todas as trocas possíveis para estes operadores foram utilizadas.
TOpCj	tAndOr, tOrAnd	Todas as trocas possíveis foram aplicadas.
INot	N.A.	Todas as trocas possíveis foram aplicadas.
RNot	N.A.	Todas as trocas possíveis foram aplicadas.
TPoAt	N.A.	Aplicado para 10% dos atributos.
Tat	N.A.	Aplicado para 10% dos atributos.
TVr	N.A.	Aplicado para 10% dos valores em condições
TPoVr	N.A.	Aplicado para 10% dos atributos em atualizações.
TNmTb	N.A.	Aplicado para 10% das tabelas.
TJoin	tInLe, tInRi, tInFu, ttInCr	Foram aplicadas apenas as trocas de INNER JOIN pelo fato de as características do software em teste apontá-lo como o principal tipo de JOIN deste cenário. Todas as trocas possíveis para este join foram utilizadas.
TFuAg	Troca de Função de Agregação	Foi aplicada apenas a troca de COUNT por SUM pelo fato de as características do software em teste apontá-lo como o principal tipo de troca para este cenário. Todas as trocas possíveis para este operador foram utilizadas.

Para a seleção dos operadores a serem aplicados, a mutação seletiva [Mathur,1991; Offutt et al., 1996] foi adaptada. A idéia básica da mutação seletiva consiste em selecionar um número reduzido de operadores de mutação que sejam realmente diferentes dos demais. Se o operador que gera um grande número de mutantes puder ser removido, então o custo de executar os mutantes também será reduzido. O processo consiste em desenvolver um conjunto de testes efetivos para matar os mutantes excluindo alguns operadores; desta forma os casos de teste são, então, executados para o conjunto inteiro de mutantes. Se o escore de mutação obtido depois da execução for próximo de um, os operadores excluídos podem ser realmente removidos, pois não são úteis para detectar novos defeitos.

Para este experimento, a seleção foi feita da seguinte forma: foram gerados 100% dos mutantes e os testes foram executados. Em uma segunda fase, foram gerados mutantes com apenas 10 por cento dos operadores selecionados, conforme mostrado na Tabela 5-2. Os escores de mutação foram então comparados; os valores foram muito próximos, indicando que os mutantes excluídos podem ser removidos, pois contribuem pouco para a detecção de novos defeitos. Os resultados alcançados por este experimento constam de material não publicado e corroboram a conclusão de outras pesquisas como a de Offutt et. al (1996).

Devido ao grande número de mutantes gerados para este experimento, alguns operadores foram aplicados em apenas dez por cento dos casos. O critério de seleção de quais operadores sofreriam a redução considerou também a escolha do testador que, com sua experiência, apontou quais seriam os casos mais interessantes de se aplicar aos operadores de mutação, levando em consideração as características do software e o conhecimento da equipe de desenvolvimento.

5.5.3. Procedimento de Realização do Experimento Equipamentos

A realização do Experimento Equipamentos seguiu o procedimento definido na Seção 5.3.

Para exemplificar analisaremos o caso de teste 002 deste experimento. Com a execução deste caso de teste os seguintes comandos SQL foram capturados (interceptados durante a execução) pela ferramenta de mutação:

```
SELECT [meq_codigo], [meq_descricao]
  FROM [Controle_EquipamentoBase2].[dbo].[tb_marca_equipamento]

SELECT [teq_codigo], [teq_descricao]
  FROM [Controle_EquipamentoBase2].[dbo].[tb_tipo_equipamento]

INSERT INTO [Controle_EquipamentoBase2].[dbo].[tb_modelo_equipamento]
        ([meq_codigo], [teq_codigo], [mod_descricao])
    VALUES
        (1, 1, '11123')

INSERT INTO [Controle_EquipamentoBase2].[dbo].[tb_modelo_equipamento]
        ([meq_codigo], [teq_codigo], [mod_descricao])
    VALUES
        (3, 1, 'PPPPPPPPPP')

INSERT INTO [Controle_EquipamentoBase2].[dbo].[tb_modelo_equipamento]
        ([meq_codigo], [teq_codigo], [mod_descricao])
    VALUES
        (1, 2, 'X@2')

INSERT INTO [Controle_EquipamentoBase2].[dbo].[tb_modelo_equipamento]
        ([meq_codigo], [teq_codigo], [mod_descricao])
    VALUES
        (1, 5, 'ótico')

INSERT INTO [Controle_EquipamentoBase2].[dbo].[tb_modelo_equipamento]
        ([meq_codigo], [teq_codigo], [mod_descricao])
    VALUES
        (3, 5, 'Sem fio')

INSERT INTO [Controle_EquipamentoBase2].[dbo].[tb_modelo_equipamento]
        ([meq_codigo], [teq_codigo], [mod_descricao])
    VALUES
        (3, 3, 'HP PhotoSmart 3110')

INSERT INTO [Controle_EquipamentoBase2].[dbo].[tb_modelo_equipamento]
        ([meq_codigo], [teq_codigo], [mod_descricao])
    VALUES
        (3, 4, 'Teclado HP cor Preta')
```

Para a aplicação do operador de mutação tPoVr no segundo comando INSERT deste caso de teste, a execução do mutante ocorrerá conforme ilustrado a seguir:

```
INSERT INTO [Controle_EquipamentoBase2].[dbo].[tb_modelo_equipamento]
        ([meq_codigo], [teq_codigo], [mod_descricao])
    VALUES
        (1, 3, 'PPPPPPPPPP') //mutante aplicado: troca de posição de atributo
(tPoVr)
```

Após a execução dos mutantes, as informações sobre as alterações e seleções feitas são armazenadas e comparadas com o comando de manipulação original. Para que um mutante seja executado, todos os comandos gerados antes do mutante são executados em sua forma original. A ferramenta avaliou automaticamente os mutantes e decidiu se eles estavam mortos ou vivos. No caso de mutantes vivos a ferramenta mostrou os resultados tanto do comando original quanto do mutante, para que o testador decida se o mutante é equivalente ou não. A ferramenta permite que o mutante seja re-executado isoladamente para análise. Um escore de mutação foi então gerado para cada caso de teste executado no experimento.

5.5.4. Resultados

Foram gerados 79 comandos para a Base de Dados 1 e 85 comandos para a Base de Dados 2, a partir do mesmo conjunto de casos de teste. A variação da quantidade de comandos de manipulação para as duas bases dá-se devido ao fato das instâncias das bases de dados exercitarem partes diferentes do software em teste, como já foi observado anteriormente (seção 2). Os resultados gerados para as duas bases de dados podem ser vistos detalhadamente no Apêndice C deste trabalho.

A Figura 5-3 mostra o escore de mutação alcançado para cada caso de teste executado em ambas as bases. Na abscissa são indicados os 20 casos de testes executados para este experimento, na ordenada é visto o escore de mutação alcançado para cada caso de teste e as barras representam os resultados alcançados para cada uma das duas bases de dados utilizadas no experimento.

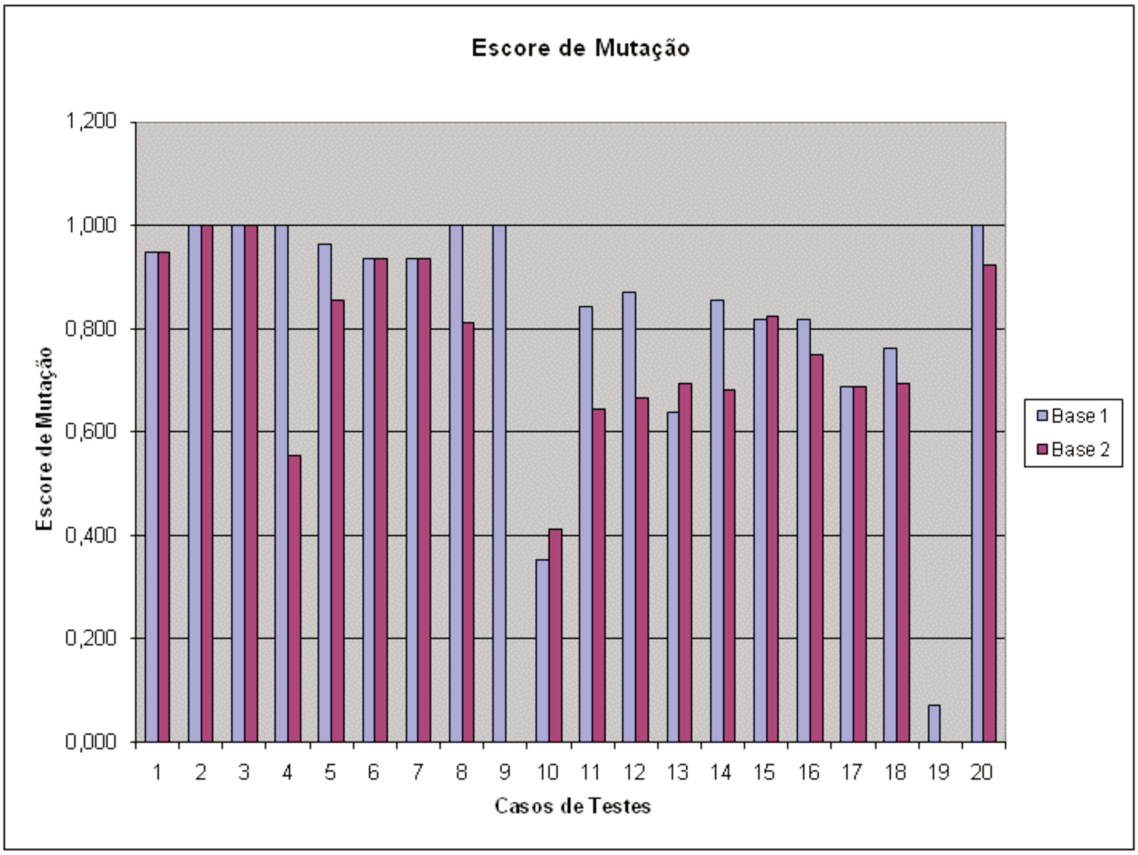

Figura 5-3 - Escores de Mutação do Experimento Equipamentos

Em relação ao estudo das bases de dados podemos notar que os valores do escore de mutação alcançados pela Base de Dados 1 são maiores do que os alcançados pela Base de Dados 2 em 11 dos 20 casos de testes, mostrando que, para o conjunto de operadores de mutação aplicado nos comandos de manipulação SQL, a Base de Dados 1 possui uma melhor capacidade de detecção de defeitos.

A Figura 5-4 mostra a diferença do escore de mutação entre as duas bases de dados. Na abscissa são indicados os 20 casos de testes executados para este experimento; quando o valor da ordenada (escores) for positivo entendemos que a Base de Dados 1 obteve um escore de mutação melhor (mais próximo de 1) do que a Base de Dados 2, quando o valor for negativo, entendemos que a Base de Dados 2 obteve um escore de mutação melhor (mais próximo de 1) do que a Base de Dados 1.

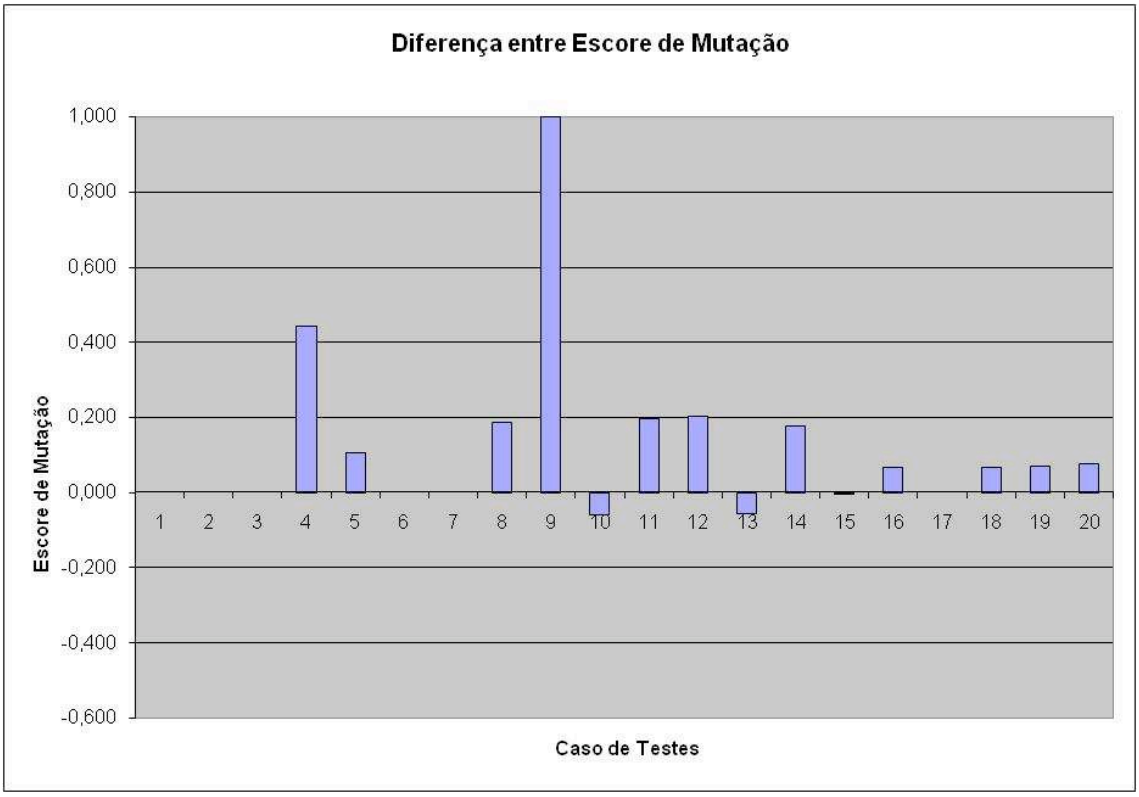

Figura 5-4 - Diferença entre os Escores de Mutação no Experimento Equipamentos

O conjunto de casos de teste utilizado neste experimento revelou 13 defeitos quando foram executados na aplicação original. Os defeitos revelados envolviam defeitos na implementação de regras de negócio e no funcionamento básico do software (inclusão, alteração, exclusão e navegação). A relação entre os defeitos revelados pela base e seus casos de teste pode ser vista no Apêndice C.

Os operadores de mutação que caracterizam os tipos de defeitos encontrados no programa original alcançaram altos escores de mutação em ambas as bases; por exemplo, o operador de mutação *tAt* atingiu o valor 1 para ambas as bases.

Mesmo com a utilização de poucos operadores, a execução do experimento mostrou um custo operacional elevado. Os testes utilizando as Bases de Dados 1 e 2 consumiram aproximadamente 48 e 120 horas para a execução dos mutantes, respectivamente. A diferença de tempo é devida à quantidade de persistências de dados na base de produção; nota-se que a maior parte dos operadores teve apenas dez por cento das possibilidades de

mutantes executadas; isto é, foram selecionados dez por cento de todos os mutantes possíveis de criação/execução.

Os mutantes que mais consumiram tempo para a sua execução foram aqueles que aumentavam a quantidade de campos selecionados como o operador *tOpCp* quando aplicado em condições. Apesar de este operador aumentar ou diminuir a quantidade de registros a serem selecionados ele não apresentou um bom índice de mutantes mortos; por exemplo, para as trocas de igualdade, os escores de mutação nas Bases de Dados 1 e 2 para os 20 casos de testes foram 0,4 e 0,52, respectivamente. Podemos concluir também que, para este operador, a Base de Dados 1 é relativamente mais reveladora de defeitos, pois conseguiu matar mais mutantes para o operador do que a Base de Dados 2, sendo uma melhor opção para detectar a presença desse tipo de defeito.

Os operadores de troca de posição, tais como o *tPoAt* e o *tPoVr*, indicaram que as bases de dados possuem um esquema bem definido, uma vez que todos os operadores de mutação foram mortos pelas bases, mostrando que as restrições de integridade das relações de dados estão preparadas para rejeitar dados inválidos. Por outro lado, a aplicação destes operadores para comparar a qualidade dos dados entre duas bases é ineficiente, pois, uma vez que a definição do esquema da base é a mesma, os resultados serão iguais independentemente dos dados que estão nestas bases.

Os mutantes de alguns operadores de mutação são extremamente difíceis de matar, pois criam relações entre tabelas que não são usuais no dia-a-dia do uso deste programa. Um bom exemplo foi o caso do operador *tJoin*, que não foi morto na maior parte dos casos de teste, pois os dados não exercitaram todas as relações de todas as formas possíveis e, portanto, situações não usuais para o sistema não possuíam dados suficientes para revelar a troca do *Join*.

A seleção dos operadores de troca de *Join* foi feita cautelosamente, pois este operador mostrou pouquíssimos índices de mutantes mortos e foi o operador que utilizou mais tempo para o processamento; algumas vezes o teste teve de ser interrompido devido ao enorme tempo de processamento que os mutantes gerados por este operador consumiram. O principal motivo para tal custo é o fato de alguns comandos com INNER JOIN terem sido efetuados com campos sem índices e quando a troca é feita por um *Join* - que amplia

a seleção dos dados (por exemplo LEFT OUTER JOIN) com um campo não indexado - o tempo de processamento sobe drasticamente. O aumento de persistências selecionadas pela troca do tipo de join pode também causar o aumento do tempo de processamento.

O fato da Base de Dados 1 alcançar, em alguns casos, escores de mutação maiores do que a Base de Dados 2 ocorre porque a ausência de persistências pode modificar o estado da base de dados acarretando resultados diferentes. Por exemplo, a ausência da persistência durante a execução de uma consulta ao banco de dados pode revelar um defeito que, caso a persistência existisse, não seria revelado.

5.6. Experimento Materiais

O experimento ocorreu no ambiente proposto pelo Cenário 1 (Seção 3.2.1), utilizou o mesmo procedimento de realização do Experimento Equipamentos (Seção 5.5.3) e ocorreu nas seguintes bases de dados: (i) Base de Dados 1 – Ambiente de produção; (ii) Base de Dados 2 – Ambiente de Produção Reduzido.

5.6.1. Contextualização

O segundo experimento utilizou um software de controle de empréstimo de materiais.

As principais funcionalidades deste software são:

- controle de entrada/saída e reserva de materiais (livros, CDs, revistas, etc) para treinamento dos funcionários;
- integração com outros sistemas da empresa, tal como cadastro dos funcionários no sistema de RH;
- envio de e-mail referente a reservas, atrasos, etc.

O software foi desenvolvido sob demanda para uma empresa varejista de médio porte. O sistema está implantado e em funcionamento em 25 filiais, sendo utilizado por aproximadamente 40 usuários em cada filial.

A equipe de desenvolvimento para produzir e testar este software possui um perfil sênior (todos os profissionais são experientes e com alto nível de qualificação em suas áreas).

Para este experimento foram utilizados 20 casos de teste projetados pelo analista de testes da equipe desenvolvedora do software. Os casos de teste projetados são testes funcionais, que utilizam um cenário da ferramenta (entende-se por um cenário um caso de uso) e visavam ao teste de regras de negócio e do funcionamento básico (Inclusão, Alteração, Navegação e Exclusão).

A base de dados do sistema é unificada para todas as filiais. Para a execução deste experimento foram consideradas as seguintes bases de dados:

Base de Dados 1 - Ambiente de Produção: Base de dados da produção sem nenhuma alteração por parte dos testadores ou desenvolvedores. As persistências de dados são reais.

Base de Dados 2 – Ambiente de Produção Reduzido: A Base de Dados 1 sofreu uma redução de persistências de dados de 30%. Esta redução foi feita por meio de uma ferramenta proprietária que, segundo o fabricante, busca reduzir o volume de dados das bases sem, no entanto, perder a qualidade dos dados de teste segundo critérios embutidos na ferramenta proprietária.

5.6.2. Seleção dos Operadores para o Experimento Materiais.

A Tabela 5-3 mostra os operadores selecionados para o Experimento Materiais, segundo os critérios definidos para tal escolha (Seção 5.2). São indicados o tipo de operador selecionado, as permutas utilizadas e são feitas as observações em relação ao uso do operador.

Devido ao grande número de mutantes gerados para este experimento, alguns operadores foram aplicados em apenas dez por cento dos casos. Baseando-se no experimento anterior (Seção 5.5), foi tomado como premissa que a redução não alterará os resultados deste experimento.

O critério de seleção dos dez por cento dos mutantes foi uma escolha feita pelo testador que, com sua experiência, apontou quais seriam os casos mais interessantes de se aplicar os operadores de mutação, levando em consideração as características do software e o conhecimento da equipe de desenvolvimento.

Tabela 5-3 - Operadores Selecionados para o Experimento Materiais.

Operador	Permutas	Observações
tInSec	tInEx, tInUn, tExIn, tExUn.	Foram aplicadas apenas as trocas de INTERCEPT e EXCEPT pelo fato das características do software em teste apontá-los como o principais operadores deste cenário. Todas as trocas possíveis para estes operadores foram utilizadas.
tOpCp	tIgDes, tIgMa, tIgMe, tIgMag, tIgMeg, tMaIg, tMaDes, tMaMe, tMaMag, tMaMeg, tMeIg, tMeDes, tMeMa, tMeMag, tMeMeg,	Foram aplicadas apenas as trocas de igualdade, maior e menor pelo fato das características do software em teste apontá-los como principais operadores deste cenário. Todas as trocas possíveis para estes operadores foram utilizadas.
tOpCj	tAndOr, tOrAnd	Todas as trocas possíveis foram aplicadas.
iNot	N.A.	Todas as trocas possíveis foram aplicadas.
rNot	N.A.	Todas as trocas possíveis foram aplicadas.
tPoAt	N.A.	Aplicado para 10% dos atributos.
tAt	N.A.	Aplicado para 10% dos atributos.
tVr	N.A.	Aplicado para 10% dos valores em condições
tPoVr	N.A.	Aplicado para 10% dos atributos em atualizações.
tNmTb	N.A.	Aplicado para 10% das tabelas.
tJoin	tInLe, tInRi, tInFu, ttInCr	Foram aplicadas apenas as trocas de INNER JOIN pelo fato das características do software em teste apontá-lo como o principal tipo de JOIN deste cenário. Todas as trocas possíveis para este join foram utilizadas.
tFuAg	Troca de Função de Agregação	Foi aplicada apenas a troca de COUNT por SUM pelo fato das características do software em teste apontá-lo como o principal tipo de troca para este cenário. Todas as trocas possíveis para este operador foram utilizadas.

5.6.3. Resultados

Foram gerados 71 comandos para a Base de Dados 1 e 66 comandos para a Base de Dados 2, a partir dos mesmos dados de testes. A variação da quantidade de comandos de manipulação para as duas bases dá-se devido ao fato de as instâncias das bases de dados exercitarem partes diferentes do software em teste. Os resultados gerados para as duas bases de dados podem ser vistos detalhadamente no Apêndice C.

A Figura 5.5 mostra o escore de mutação alcançado para cada caso de teste executado em ambas as bases. Na abscissa são indicados os 20 casos de testes executados para este experimento, na ordenada é mostrado o escore de mutação alcançado para cada caso de teste e as barras representam os resultados alcançados para cada uma das bases de dados utilizadas no experimento.

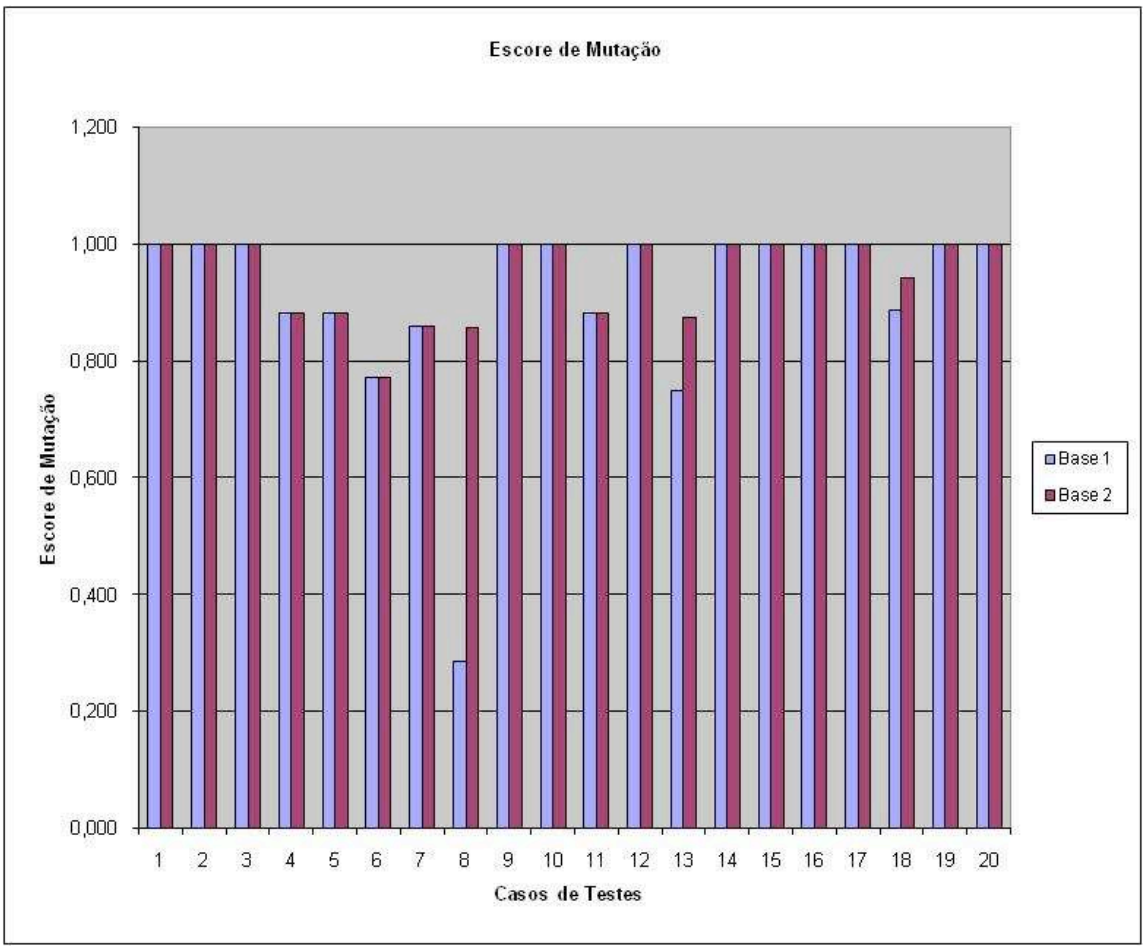

Figura 5-4 - Escores de Mutação do Experimento Materiais

Em relação ao estudo das bases de dados podemos notar que os valores do escore de mutação alcançados pela Base de Dados 2 são superiores em três casos de teste em relação à Base de Dados 1, mostrando que para o conjunto de operadores de mutação aplicado nos comandos de manipulação SQL, a Base de Dados 2 possui melhor capacidade de revelar defeitos.

O gráfico da Figura 5-6 mostra a diferença do escore de mutação entre as duas bases de dados; quando o valor da ordenada (escore) for positivo entendemos que a Base de Dados 1 obteve um escore de mutação melhor (mais próximo de 1) do que a Base de Dados 2, quando o valor for negativo, entendemos que a Base de Dados 2 obteve um escore de mutação melhor (mais próximo de 1) do que a Base de Dados 1.

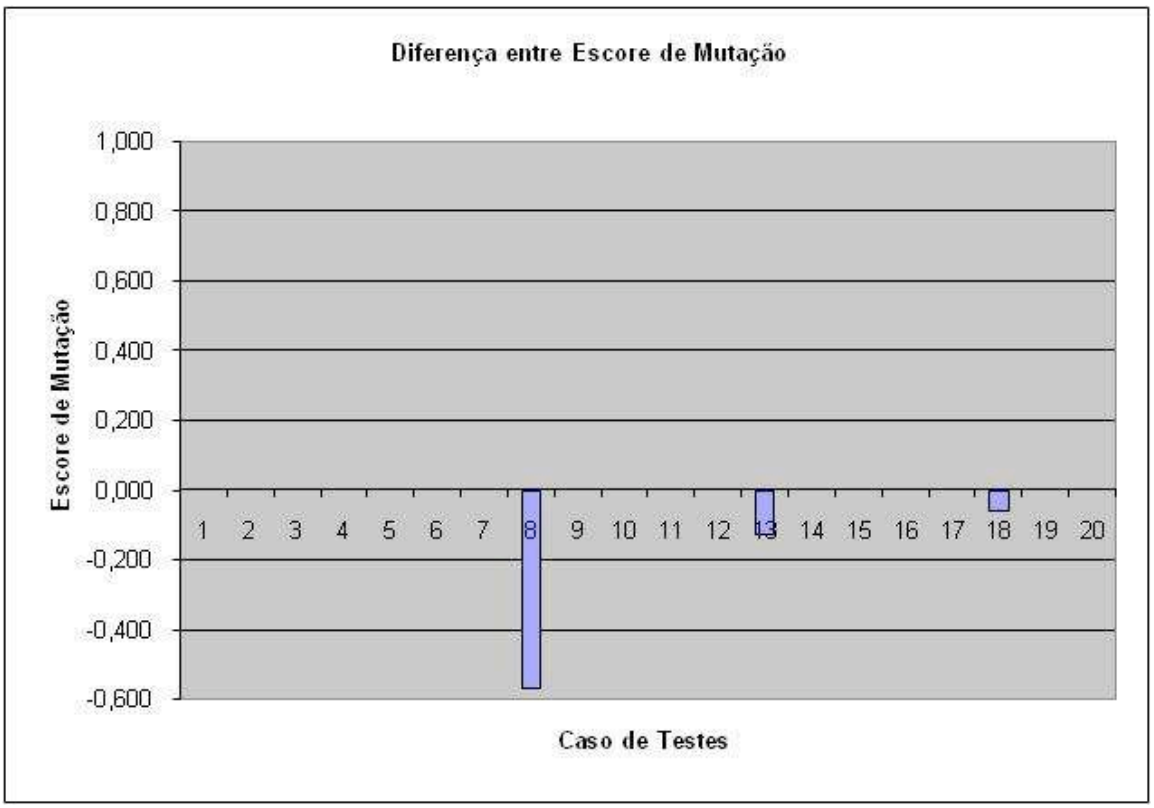

Figura 5-5 - Diferença entre os Escores de Mutação no Experimento Materiais

O conjunto de casos de testes utilizado neste experimento revelou 13 defeitos na aplicação: os defeitos revelados envolviam defeitos na implementação de regras de negócio e no funcionamento básico do software (inclusão, alteração, exclusão e navegação). A relação entre os defeitos revelados e seus casos de testes pode ser vista no Apêndice C.

Baseando-se nos defeitos encontrados no programa original foram identificados, dentre os operadores de mutação criados e executados, quais poderiam representar os mesmos tipos de defeitos encontrados no programa original. Este conjunto de operadores identificados não alcançou altos escores de mutação em ambas as bases, ou seja, não foram revelados em sua totalidade, como o operador de mutação *tAt* que caracteriza um tipo de defeito encontrado no programa original, que alcançou escores de mutação de 0,66 a 1.

Novamente a execução do experimento demonstrou um custo operacional elevado. As Bases de Dados 1 e 2 consumiram aproximadamente 96 e 50 horas de processamento

para a execução dos mutantes, respectivamente. Esta diferença é devida ao volume relativo de persistências de dados na base de produção.

Os mutantes que consumiram mais tempo para a sua execução foram aqueles que aumentam a quantidade de campos selecionados, tais como os operadores *tOpCp* e *tJoin*.

Os mutantes oriundos do operador de troca de *join* permaneceram vivos em todos os casos, tal como ocorreu no Experimento Equipamentos. A mesma constatação é verificada para os operadores *dtPoAt* e *tPoVr*.

Os mutantes criados pelos operadores de troca de intersecção (*tInSec*) foram mortos em todos os casos, demonstrando que ambas as bases são fortes em diversidade de dados, isto é, possuem persistências de dados que abrangem uma larga faixa de valores para o tipo de atributo ao qual elas pertencem.

Ambas as bases obtiveram elevados escores de mutação, mostrando que a redução da base em produção não afetou a qualidade dos dados utilizados nos testes. Entretanto, podemos observar que a base de dados reduzida (Base de Dados 2) exercitou aproximadamente 8% menos código do que a base de produção (a Base de Dados 1 executou cinco comandos de manipulação a mais do que a Base de Dados 2). Esta diferença afetou também a quantidade de defeitos revelados por cada base; a base de dados reduzida revelou dois defeitos a menos que a base de produção.

Os resultados alcançados neste experimento são conflitantes em diversos pontos, o que nos leva à conclusão de que o estudo sobre a eficiência de uma base de dados em revelar defeitos envolve diversos fatores e não apenas o volume das persistências de dados. Um bom exemplo disto é o Caso de Testes número oito, que conseguiu revelar um defeito quando executado na Base de Dados 2, mas não obteve nenhum defeito revelado na Base de Dados 1. O escore de mutação para a Base de Dados 2 também foi mais alto do que na Base de Dados 1 com a aplicação dos mesmos operadores de mutação. Tendo em vista que a Base de Dados 2 é uma redução da Base de Dados 1, os seus resultados deveriam ser inferiores ou iguais ao da base de dados original; ao contrário, os resultados foram melhores para a Base de Dados 2 em todos os sentidos. Neste caso, a ausência de algumas persistências de dados fez com que o programa tivesse um comportamento diferente para cada uma das bases de dados gerando este resultado inesperado.

5.7. Considerações Finais

Na comparação entre bases de dados de um mesmo experimento no Cenário 1 (Seção 3.2.1) deve-se considerar que a seleção da base de dados de entrada pode interferir na escolha da aplicação hospedeira em relação ao comando de manipulação a ser executado. Este aspecto pode determinar que os comandos de manipulação executados para um determinado caso de teste da linguagem hospedeira na Base de Dados 1 sejam diferentes dos comandos executados a partir da Base de Dados 2.

O Experimento Controle mostrou que no Cenário 2 (Seção 3.2.2) a comparação entre bases de dados traz resultados que podem realmente classificar qual base de dados é mais forte para um determinado conjunto de comandos SQL. Isto se deve ao fato de os comandos SQL serem os mesmos para todas as bases de dados.

Independentemente da comparação entre as bases de dados, diversas informações sobre elas podem ser classificadas por meio da aplicação dos operadores propostos, tais como:

- Capacidade da base de dados em revelar defeitos, medida pelo escore de mutação dos operadores de mutação;
- Qualidade das restrições das relações de dados, determinada pelos operadores de troca de posição como *tPoVr* e *tPoAtr*.

O operador de troca de *join* conseguiu atingir no Experimento Controle, ao contrário dos outros dois experimentos, escores de mutação diferentes de zero, mostrando que, se a base de dados for forte o suficiente para revelar este tipo de defeito, ele será revelado; o que leva à conclusão de que nos dois últimos experimentos as bases de dados eram inadequadas à detecção desse tipo de defeito. Mesmo assim a seleção dos operadores de troca de *join* deve ser feita cautelosamente, pois esse operador demonstrou reduzido índice de mutantes mortos em todos os experimentos e foi o operador que mais utilizou tempo de processamento, provocando muitas vezes o aborto do experimento. O principal motivo é o fato de alguns comandos que usam INNER JOIN serem efetuados com campos sem índices e quando a troca é feita, por exemplo, por um LEFT OUTER JOIN - que amplia a seleção dos dados com um campo não indexado - o tempo de processamento torna-se elevado.

O fato da base de dados reduzida alcançar, em alguns casos, escores de mutação maiores do que os da base de dados de produção ocorre porque a ausência de persistências pode modificar o estado da base de dados acarretando resultados diferentes. Por exemplo, a ausência da persistência durante a execução de uma consulta ao banco de dados pode revelar um defeito que, caso a persistência existisse, não seria revelado.

6. Conclusões

Apesar dos progressos que o meio industrial e acadêmico vêm fazendo na direção da qualidade de aplicações de banco de dados, ainda existe um campo de pesquisas em aberto. Mais especificamente, com respeito à utilização da análise de mutantes para comandos da linguagem SQL, apenas dois estudos foram identificados [Tuya, 2007; Chan, 2005]; e mesmo assim, tais pesquisas limitam-se a: (i) estudo de comandos de consulta da linguagem SQL que não sofrem alterações durante a execução; e (ii) experimentos utilizando apenas dados sintéticos e sem suporte à automação via ferramenta.

Neste trabalho é investigada a extensão do critério de análise de mutantes a aplicações que utilizam comandos da linguagem SQL para acessar banco de dados. Um conjunto de operadores de mutação é proposto visando à caracterização de defeitos em diversos comandos da linguagem SQL e até mesmo no esquema da base de dados. Em adição, é feito um estudo dos cenários onde a análise de mutantes pode ser aplicada, possibilitando até mesmo a aplicação em comandos que sofrem alterações durante a execução, por meio da mutação fraca.

O objetivo principal foi contribuir para uma melhor qualidade dos testes de aplicações de banco de dados, visando à escolha de bases de dados mais reveladoras de defeitos pela aplicação da análise de mutantes em comandos da linguagem SQL.

6.1. Síntese do Trabalho

No âmbito da SQL, estudou-se a extensão do critério de análise de mutantes para comandos de manipulação (embutidos ou não em uma linguagem hospedeira). Os cenários onde este critério pode ser aplicado para a linguagem SQL são investigados. A partir dessa investigação dois cenários são definidos: (i) a aplicação da análise de mutantes em comandos SQL embutidos em uma linguagem hospedeira - onde as instâncias da base de dados fazem parte do conjunto de dados de entrada dos casos de testes, que exercitarão a linguagem hospedeira; e (ii) a aplicação da análise de mutantes em comandos SQL que não estão embutidos em uma linguagem hospedeira, sendo possível a utilização do nível da mutação fraca.

O estudo dos defeitos relacionados aos comandos SQL, que podem ser modelados com operadores de mutação, resulta na criação de cinco categorias de operadores que visam à caracterização de defeitos para uma ampla quantidade de comandos da linguagem SQL. As categorias são:

- operadores de mutação para operadores de SQL;
- operadores de mutação miscelânea;
- operadores de mutação de fluxo de dados;
- operadores de mutação para controle de transação;
- operadores de mutação para funções e procedimentos.

Para validar os operadores de mutação uma ferramenta que suporta a geração, a execução e a análise dos mutantes foi implementada. Esta ferramenta pode capturar comandos SQL independentemente da linguagem hospedeira onde os comandos estão embutidos, garantindo o funcionamento da ferramenta em ambos os cenários de aplicação da análise de mutantes descrita neste trabalho.

Cada experimento foi executado para duas bases de dados da aplicação, sendo sempre a base de produção (base real da aplicação) e uma base de dados reduzida (ambiente de testes ou redução orientada). Foram analisados dois experimentos no Cenário 1 (Seção 3.2.1) e um experimento no Cenário 2 (Seção 3.2.2). Os experimentos mostram que a comparação direta da capacidade em revelar defeitos entre as bases de dados é possível no Cenário 2, onde os comandos SQL não sofrem mutação e são sempre os mesmos, independentemente dos dados de entrada e das instâncias de dados. A capacidade da base em revelar defeitos, ou seja, o quão reveladora uma base de dados é em relação a um determinado conjunto de operadores de mutação, pode ser avaliada em qualquer cenário.

6.2. Contribuições

As principais contribuições deste trabalho estão resumidas abaixo:

- extensão do critério de análise de mutantes para aplicações que acessam banco de dados, visando à qualidade das aplicações de banco de dados (Seção 3.1);

- definição e investigação empírica dos cenários de aplicação do teste de mutação para a SQL (Seção 3.2 e Capítulo 5);
- definição de um conjunto de operadores, utilizados na caracterização de defeitos em comandos da linguagem SQL que acessam bases de dados, que suportam a identificação da capacidade que uma base de dados possui em revelar defeitos (Capítulo 4);
- desenvolvimento de uma ferramenta que automatiza a geração, a execução e a análise dos mutantes gerados pela técnica proposta e auxilia a avaliação dos mutantes equivalentes, independentemente da linguagem hospedeira que possui os comandos SQL embutidos (Apêndice A);
- condução de experimentos com o objetivo de validar a ferramenta e a eficácia da extensão do critério de análise de mutantes (Capítulo 5);
- investigação empírica dos operadores de mutação propostos (Capítulo 5 e Apêndice C).

Alguns trabalhos futuros são:

- estender a análise experimental a outras aplicações reais e dados reais;
- utilizar a técnica apresentada neste trabalho para a geração de bases de dados;
- implantar a técnica nas empresas que colocaram à disposição os programas e dados reais para o experimento; inicialmente, motivar a avaliação de todas as bases de dados reduzidas derivadas de bases de dados de produção, para analisar a sua eficácia na atividade de teste;
- aperfeiçoar a ferramenta para aplicação da técnica de análise de mutantes em aplicações SQL de banco de dados, permitindo um melhor desempenho na execução dos testes, análise dos resultados e melhor visualização dos resultados obtidos;

- aplicar os conceitos de testes de banco de dados na criação de testes funcionais com o intuito de melhorar a qualidade dos testes funcionais sem, no entanto, projetar testes de banco de dados para as aplicações;
- criar bases de dados sintéticos que possuam uma grande diversidade de persistências de dados com apoio dos operadores de mutação criados neste trabalho;
- desenvolver uma técnica capaz de mensurar o tamanho de aplicações SQL de banco de dados, estabelecendo variáveis que sejam capazes de estimar o esforço necessário para testá-la;
- estudar cenários envolvendo técnicas de testes, processos e ferramentas com o intuito de aprimorar o tempo despendido com teste de software.

7. Referências Bibliográficas

Aranha, M. C. L. F. M.; Mendes, N. C.; Jino, M. and Toledo, C. M. T. (2000) RDBTool: Uma Ferramenta de Apoio ao Teste de Bases de Dados Relacionais, *XI CITS: Conferência Intl. de Tecnologia de Software*, Curitiba, Paraná, agosto.

ANSI/ISO (ISO/IEC 9075-2:2003) Information technology - Database languages - SQL - Part 2: Foundation (SQL/Foundation), *American National Standarts Instituite / International Organization for Standardization*, agosto.

Barbosa, E. F.; Maldonado, J. C. and Vincenzi, A.M.R. (2001). Towards the determination of sufficient mutant operator for C. *STVR- Software Testing, Verification and Reliability*, Vol.11, nº 2, pages 113-136.

Beizer, B. (1990). Software Testing Techniques, 2nd ed., *International Thomson Computer Press*, Junho.

Bueno, P.M.S. (1999). Geração Automática de Dados e Tratamento de Não Executabilidade no Teste Estrutural de Software, Dissertação de Mestrado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, Junho.

Carey, M.J.; DeWitt, D.J. and Naughton, J.F. (1993). The 007 Benchmark. *Proc. of the 1993 ACM SIGMOD Intl. Conf. on Management of data*, New York, NY, May, pages 12-21.

Chan, M.Y. and Cheung, S. C. (1999). Testing Database Applications with SQL Semantic. *Proc. of 2nd Intl. Symposium on Cooperative Database Systems for Advanced Applications (CODAS'99)*, Wollongong, Austrália, March, pages 363-374.

Chan, W.K.; Cheung, S.C. and Tse, T.H (2005). Fault-based testing of database application programs with conceptual data model. *Proc. of the 5th Intl. Conference on Quality Software*, *IEEE Computer Society Press,* Los Alamitos, CA, pages 187–196.

Chays, D.; Dan, S.; Frankl, P. G.; Vokolos, F. I. and Weyuker, E. J (2000). A Framework for Testing Database Applications. *Proc. of the 2000 ACM SIGSOFT Intl. Symposium on Software Testing and Analysis*, Portland, Oregon, pages 147-157.

Chays, D.; Deng, Y.; Frankl, P. G.; Dan, S.; Vokolos, F. I. and Weyuker, E. J (2002). *AGENDA: A Test Generator for Relational Database Applications. Technical report, Department of Computer Science*, Polytechnic University, Brooklyn, Long Island, Westchester, Agosto.

Chays, D. and Deng, Y. (2003). Demonstration of AGENDA Tool Set for Testing Relational Database Applications. *Proc. of the 25th Intl. Conference on Software Engineering,* Portland, Oregon, pages 802-803.

Codd, E.F. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, Vol. 13, n 6, June pages 377-387.

Daou, B.**;** Haraty, R. A. and Mansour, N. (2001). Regression Testing of Database Applications. *Proc. of the 2001 ACM Symposium on Applied Computing*, Las Vegas, Nevada, pages 285-289.

Davies, R. A.; Beynon, R. J. A. and Jones, B. F. (2000). Automating the Testing of Databases. *Proc. of the 1st Intl. Workshop on Automated Program Analysis, Testing and Verification, IEEE Computer Society*, 16, June, pages 15-20.

Delamaro, M. E.; Maldonado, J. C. and Mathur, A. P. (2001). Interface Mutation: An Approach for Integration Testing. *IEEE Transactions on Software Engineering*, Vol. 27, n 3, pages 228-247.

Delamaro, M. E. and Maldonado, J. C. (1999) Interface mutation: Assessing testing quality at interprocedural level. *Proc. of the XIX International Conference of the Chilean Computer Science Society*, pages 78-86.

Delamaro, M. E. and Maldonado, J. C. (1996). Proteum: A tool for the assessment of test adequacy for C programs. *Proc. of the Conference on Performability in Computing Systems,* Brunswick, NJ, pages 79-95.

DeMillo, R.A.; Acree, A.T.; Budd, A.T. and Sayward, F.G.(1979). Mutation Analysis. *Georgia Institute of Technology*, Atlanta, GA.

DeMillo, R.A. (1980). Mutation analysis as a tool for software quality assurance. *Proc. of COMPSAC 80*, Chicago, IL, October.

DeMillo, R.A. (1987). Software Testing and Evaluation. *The Benjamin/Cumming Publishing Company Inc*.

DeMillo, R.A.; Lipton, R.J. and Sayward, F.G.(1978). Hints of test data selection: Help for the practicing programmer. *IEEE Computer*, Vol.11, n 4, pages 34-43.

Deng, Y.; Frankl, P. and Wang, J.(2004). Testing web database applications. *Workshop on Testing Analysis and Verification of Web Services*. *ACM Press*, New York, pages 1–10.

Deutsch, M.S. (2002). *Verification and Validation in* PRESSMAN, Roger. Engenharia de Software. São Paulo: Makron Books, page 464.

Elbaum, S.; Rothermel, G.; Karre, S. and Fisher, M. (2005). Leveraging user session data to support web application testing. *IEEE Transactions on Software Engineering*, 31 (3), pages 187–202.

Elbaum, S.; Malishevssky, A.G. and Rothermel, G.(2002). Test case priorization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28 (2), pages 159–182.

Elmasri, R. and Navathe, S. B. (2006). Fundamentals of Database Systems, 5th Ed., Addison Wesley.

Harman, M.; Hierons, R. and Danicic, S.(2000). The Relationship Between Program Dependence and Mutation Analysis. *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, San Jose, CA, October, pages 5-13.

Howden, W.E. (1982). Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, USA, New Jersey, July, pages 371– 379.

IEEE número 610.12-1990 *IEEE Computer Society,* **1991.**

Jeng, B. and **Weyuker**, E. J.(1989). Some Observations on Partition Testing. *Proc. of the ACM SIGSOFT 89, Third Symposium on Software Testing, Analysis and Verification.* Key West, FL, December, pages 13-15.

Jester – The JUnit Test Tester – Página da internet acessada em 08/09/2008: http://www.xpdeveloper.net/xpdwiki/Wiki.jsp?page=JesTer

Kapfhammer, G. M. and **Soffa,** M. L.(2003). A Family of Test Adequacy Criteria for Database-Driven Applications. *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering,* (ESEC/FSE 2003), Helsinki, Finlândia, September.

Leitão-Junior, P. S.; Vilela, P. R. S. and Jino, M. (2005). Mapping Faults to Failures in SQL Manipulation Commands. *Proc. of the 3rd ACS/IEEE Intl. Conference on Computer Systems and Applications (AICCSA-05)*, Egito, Cairo, Janeiro.

Leitão-Junior, P. S; Cardoso, V. M. and Jino, M.(2002). Aspectos de Fluxo de Controle, de Fluxo de Dados e de Instrumentação no Contexto de Banco de Dados Ativos. Relatório Técnico, Faculdade de Engenharia Elétrica e de Computação, UNICAMP.

Leitão-Junior, P. S.; Vilela, P.R.S. and Jino, M.(2008). Data Flow Testing of SQL-based Active Database Applications. *In: The Third International Conference on Software Engineering Advances*, pages 26-31, Sliema, Malta.

McCabe, T.(1976). A Complexity Measure. *IEEE Transactions on Software Enginnering*, Vol SE-02 n 4, pages 308-320, Manchester, UK, Dezembro.

Maldonado, J.C.(1991). Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software. Tese de Doutorado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, Campinas.

Ma, Y.S.; Offutt, J.; Kwon, Y.R. and MuJava.(2005). An automated class mutation system. *Software Testing, Verification and Reliability,* 15 (2), pages 97–133.

Mannila, H. and **Räihä,** K.J.(1986). Test data for relational queries. *Proc. of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, ACM Press*, New York, NY, USA, pages 217–223.

Mansour, N. and Houri, M.(2005). Testing web applications. *Information and Software Technology,* 48 (1), pages 31–42.

Mathur, A. P.(1991). Performance, effectiveness and reliability issues in software testing. *15th Annual International Computer Software and Application Conference*, Tokio, Japan, *IEEE Computer Society Press*, pages 604-605.

Myers, G.J.(2004). *The Art of Software Testing*.Wiley, New York, 2004.

Offutt, A. J.; Rothermel, G. and Zapf, C.(1993). An experimental evaluation of selective mutation. *15th International Conference on Software Engineering, Baltimore MD. IEEE Computer Society Press*, pages 100-107.

Offutt, A. J.; Lee, A.; Rothermel, G.; Untch, R.H. and Zapf, C.(1996). An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, Vol.5, n 2, pages 99-118.

Offutt, A. J.(1992). Investigations of the Software Testing Coupling Effect. *ACM Trans. Software Eng. and Methodology,* Vol. 1, n 1, pages 3-18.

Offutt, A.J. and Xu, W.(2004). Generating test cases for web services using Data Perturbation. *Workshop on Testing, Analysis and Verification of Web Services, ACM Press,* New York, NY, USA, pages 1–10.

Offutt, A.J. and Untch, R.H.(2000). Mutation 2000: Uniting the Orthogonal*, Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, San Jose, CA, October, pages 45-55.

Ostrand, T.J. and **Balcer,** M.J. (1988). The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6), June, pages 676-686.

Pressman, R. S. (2005). *Software Engineering*: *A Practitioner's Approach*, 6th Ed., McGraw-Hill.

Pönighaus, R.(1995). 'Favourite' SQL-Statements – an empirical analysis of SQL-Usage in commercial applications. *Proc. of the 6th Intl. Conference on Information Systems and Management of Data, Lecture Notes in Computer Science*, Vol. 1006, Springer, pages 75–91.

Rapps, S. and Weyuker, E.J.(1985). Selecting Software Test Data using Data Flow Information. *IEEE Transactions on Software Engineering*, Vol.SE-11, n 4, EUA, New Jersey, pages 367-375.

Robbert, M. A. and Maryanski, F.J.(1991). Automated Test Plan Generator for Database Application Systems. *Proc. of the ACM SIGSAMLL/PC Symposium on Small Systems*, Toronto, Ontario, pages 100-106.

Rothermel, G. and Elbaum, S.(2003). Putting your best tests forward. *IEEE Software,* 20 (5), pages 74–77.

Rothermel, G.; Untch, R.H. and Harrold, M.J.(2001). Prioritizing test cases for regression testing, *IEEE Transactions on Software Engineering,* 27 (10), pages 929–948.

Spoto, E.S.(2000). Teste Estrutural de Programas de Aplicação de Banco de Dados Relacional. Tese de Doutorado, Faculdade de Engenharia Elétrica e de Computação, UNICAMP, Campinas, Dezembro.

Spoto, E.S.; Leitão-Junior, P.S.; Jino, M. and Maldonado, J. C.(2005). Teste Estrutural de Integração de Programas de Aplicação de Banco de Dados Relacional. *IV Simpósio Brasileiro de Qualidade de Software (SBQS 2005)*, Porto Alegre, Junho.

Sommerville, I. (2003). Engenharia de Software*,* 6ª Ed., Addison Wesley, 2003.

TPC. (2007) Transaction Processing Performance Council. *TPC Benchmark C Standard Specification Revision 5.9*. http://www.tpc.org/tpcc/spec/tpcc_current.pdf , June.

Tsai, W.T.; Volovik, D. and Keefe, T.F.(1990). Automated test case generation for programs specified by relational algebra queries, *IEEE Transactions on Software Engineering,* Vol.16 (3), pages 316–324.

Suarez-Cabal, M.J. and **Tuya,** J.(2004). Using an SQL Coverage Measurement for Testing Database Applications, *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, New York, NY, USA, pages 253-262.

Tuya, J.; Suárez-Cabal, M. and de la Riva, C.(2007). Mutation Database queries. *Information and Software Technology,* 79, pages 398-417.

Weyuker, E. J.(1982). On Testable non Testable Programs. *The Computer Journal*, Vol. 25, n 4, November, pages 465-470.

Willmor, D. and **Embury,** S.M.(2005). A safe regression test selection technique for database driven applications. *Proc. of the IEEE International Conference on Software Maintenance*, Manchester, UK, September, pages 421-430.

Wong, E. W. and **Mathur,** A. P. (1995). Reducing the cost of mutation testing: An empirical study. *The Journal of Systems and Software*, Vol. 31, n 3, pages 185-196.

Woodward, M.R.(1993). Mutation Testing - its Origin and Evolution. *Information and Software Technology*, Vol. 35, n 3, March, pages 163–169.

Woodward, M.R. and **Halewood,** K.(1988). From weak to strong, dead or alive? An analysis of some mutation testing issues. *Proc. of the 2nd Workshop on Software Testing, Verification and Analysis*. Banff, Canada, July, pages 152-158.

Zhang, J. (2000). Specification Analysis and Test Data Generation by Solving Boolean Combinations of Numeric Constraints. *Proc. of the 1st Asia-Pacific Conference on Quality Software*, (APAQS), October, pages 267-274.

Zhang, J.; Xu, C. and Cheung, S. C.(2001). Automatic Generation of Database Instances for White-box Testing. *Proc. of the 25th Annual International Computer Software and Applications Conference* (COMPSAC'01), Chicago, Illinois.

Apêndice A - Descrição da Ferramenta de Suporte ao Teste de Mutação para SQL

Devido ao número elevado de mutantes criados a partir de uma classe de operadores, torna-se extensiva a execução e análise dos mutantes vivos x equivalentes, o que exige a automação destas tarefas. Para executar os experimentos deste trabalho algumas tarefas foram automatizadas com o auxílio de uma ferramenta desenvolvida para criar, executar e analisar os mutantes.

Neste apêndice apresenta-se o comportamento da ferramenta nos cenários onde a análise de mutantes pode ser aplicada (Seção 3.3), os seus principais módulos e as suas características técnicas.

A.1. Síntese de Funcionamento

A ferramenta é capaz de capturar um conjunto de seqüência S de comandos SQL de uma determinada unidade de programa e as instâncias do banco de dados que a aplicação pretende acessar ou modificar, esta captura independe do cenário (Seção 3.3.1 e 3.3.2) em que os testes serão executados.

Após a fase de captura, a ferramenta procura em um conjunto de operadores de mutação MO (pré-definido na ferramenta pelo testador) por um subconjunto de operadores de mutação aplicáveis ao conjunto S e deriva um conjunto M* de mutantes possíveis = $\{mo_{11}, mo_{12}, ..., mo_{i1}, mo_{i2}, ..., mo_{ij}, ..., mo_{x1}, mo_{x2}, ... \}$.

A ferramenta executa os mutantes e retorna os seus respectivos resultados juntamente com a decisão de se o mutante esta *vivo* ou *morto*. Para os mutantes *vivos* a análise de uma possível equivalência deve ser feita manualmente (auxiliada pela ferramenta) e a decisão poderá ser armazenada na ferramenta.

A.2. Módulos da Ferramenta

Distinguem-se quatro etapas na automação da análise de mutantes para SQL. A primeira, em geral, concentra-se na definição dos operadores de mutação a serem aplicados nos comandos que serão capturados; a segunda etapa concentra-se na captura dos comandos SQL executados pela linguagem hospedeira, bem como os dados de entrada, o estado da

base de dados e os resultados obtidos pela execução dos comandos SQL; a terceira etapa concentra-se na geração e execução dos mutantes e a quarta e última etapa concentra-se na análise dos resultados.

As próximas seções descrevem as atividades suportadas pela ferramenta.

A.2.1. Definição dos Operadores de Mutação

Para que a definição dos operadores seja feita para um determinado experimento, é necessária a criação de um projeto de teste que indique qual será o banco de dados alvo dos comandos SQL. A Figura A-1 ilustra a tela de cadastro de Projetos.

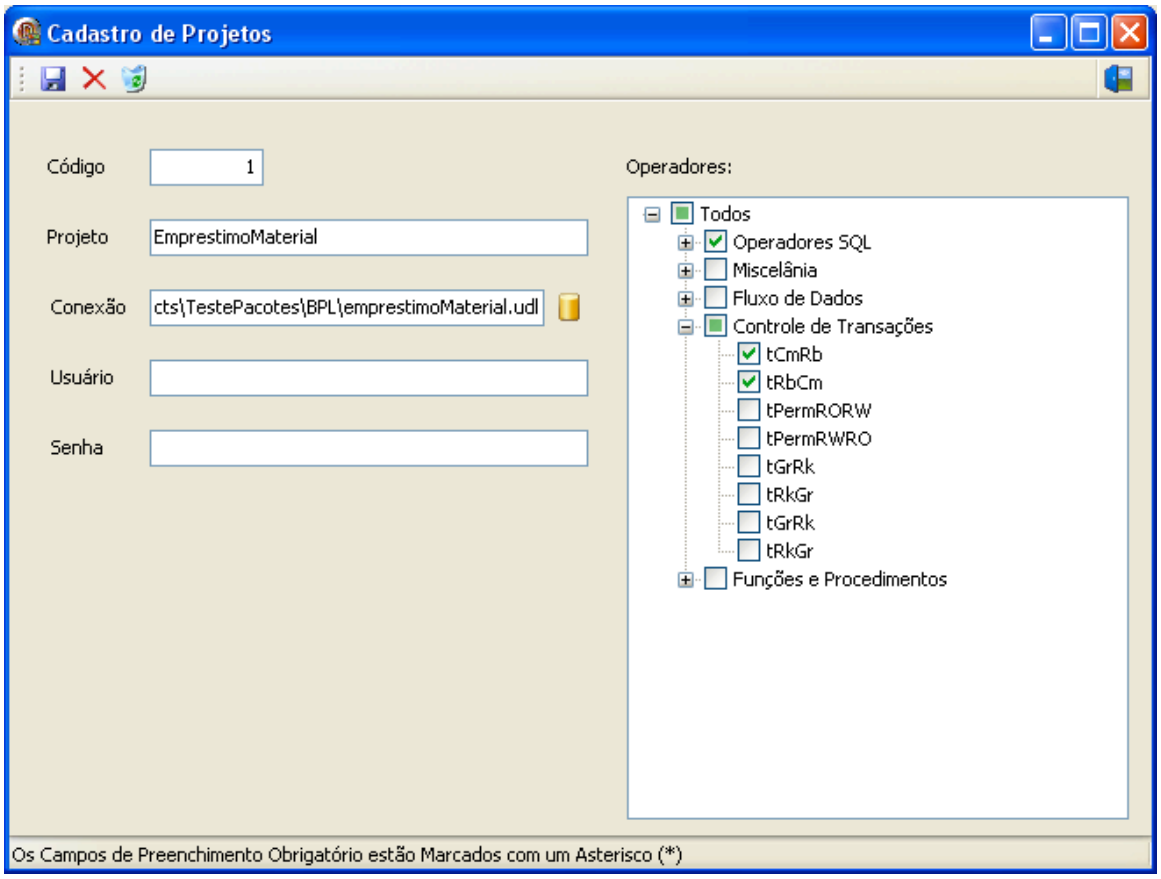

Figura A-1 - Cadastro de Projetos

Após as definições relativas ao banco de dados, é possível escolher quais os operadores de mutação serão aplicados para o projeto criado. Para tanto, basta clicar nas categorias e/ou operadores desejados, visualizados na estrutura de árvore no lado esquerdo da tela de Cadastro de Projetos (Figura A-1).

Os operadores de mutação disponibilizados pela ferramenta são todos aqueles citados neste trabalho (pré-cadastrados na ferramenta). Caso seja necessário criar novas categorias de operadores e operadores de mutação, a ferramenta possui um módulo para esse cadastro. Este módulo é limitado à criação de operadores de troca simples, isto é, substituição de um valor por outro. A Figura A-2 ilustra o cadastro de novos operadores.

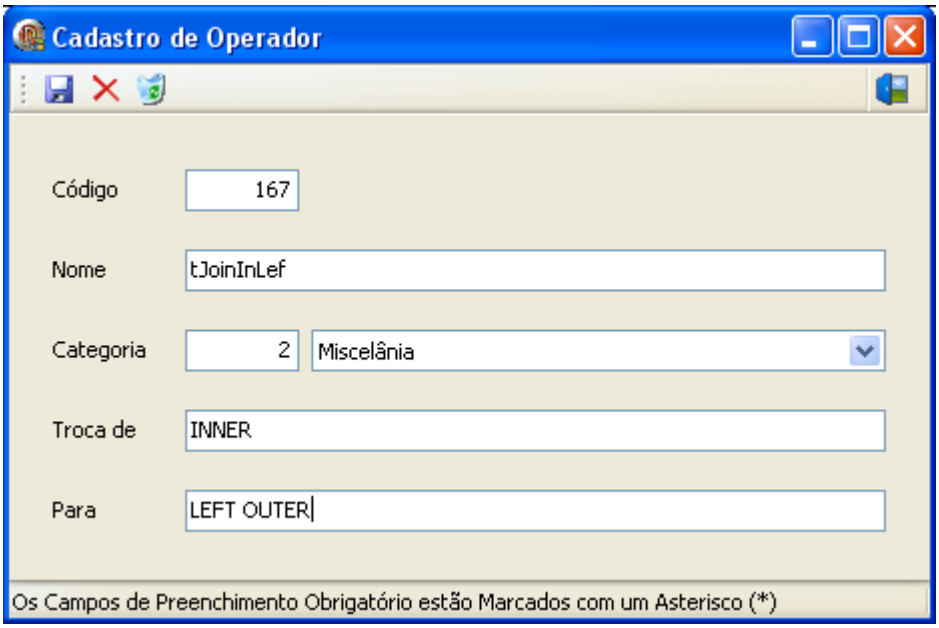

Figura A-2 - Cadastro de Operadores

A.2.2. Captura dos Comandos SQL

Para que a captura dos comandos SQL seja feita pela ferramenta basta iniciá-la antes da execução dos casos de testes; para que tal captura seja efetuada basta escolher o projeto na caixa de seleção de projetos e clicar no botão “Inicia Captura”, conforme ilustrado na Figura A-3.

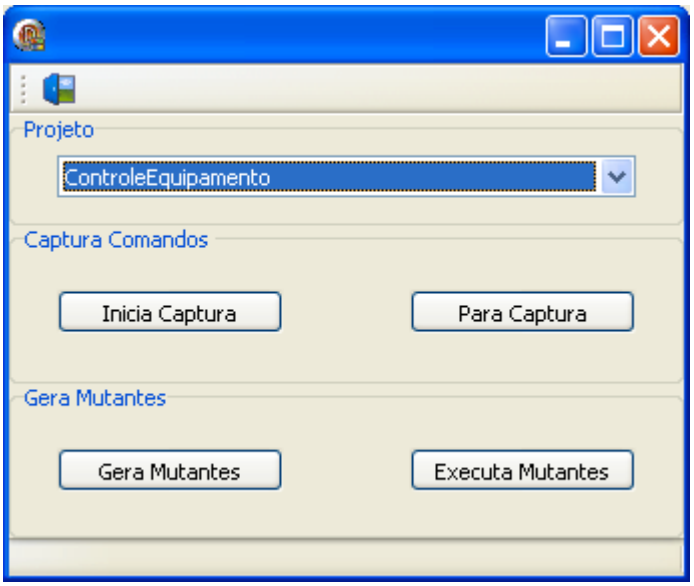

Figura A-3 - Captura e Geração

Ao termino da execução do primeiro caso de teste é necessário clicar no botão "Para Captura", então, uma tela será exibida para o cadastramento de uma identificação para os dados coletados pela ferramenta, conforme ilustrado na Figura A – 4.

Figura A - 4 - Seleção de Grupo

O procedimento pode ser repetido para a execução de outros casos de teste ou grupos de comandos. Os dados identificados neste passo podem representar apenas um comando SQL, como no caso da execução no Cenário 2 (Seção 3.3.2) ou um conjunto de comandos SQL como no caso da execução no Cenário 1 (Seção 3.3.1).

Além dos comandos executados, serão capturados os resultados dos comandos, o estado da base de dados - antes e depois da execução dos comandos pertencentes ao grupo identificado na tela de seleção de grupos - e qualquer informação pertinente à base de dados que possa servir para a análise dos dados.

A.2.3. Geração e Execução dos Mutantes

Após a captura de todos os grupos de comandos SQL em que os operadores serão aplicados, é necessário clicar no botão "Gerar Mutantes", ilustrado na Figura A – 4. Após a finalização da geração dos mutantes, clicando no botão "Executar Mutantes" todos os mutantes serão executados e seus resultados serão exibidos, conforme Figura A-5. Estas duas tarefas provavelmente consumirão um tempo razoável de execução, pois estarão analisando os comandos capturados e criando, em tempo de execução, todos os mutantes possíveis para a aplicação dos operadores de mutação selecionados. No caso do Cenário 1 - em que o critério de mutação é o Forte (*Strong Mutation*) - será consumido um tempo maior na execução, uma vez que, todos os comandos SQL capturados devem ser executados, e não apenas o comando que sofreu mutação, como é o caso do Cenário 2 (Seção 3.3.1) que utiliza o critério de mutação Fraca (*Weak Mutation*).

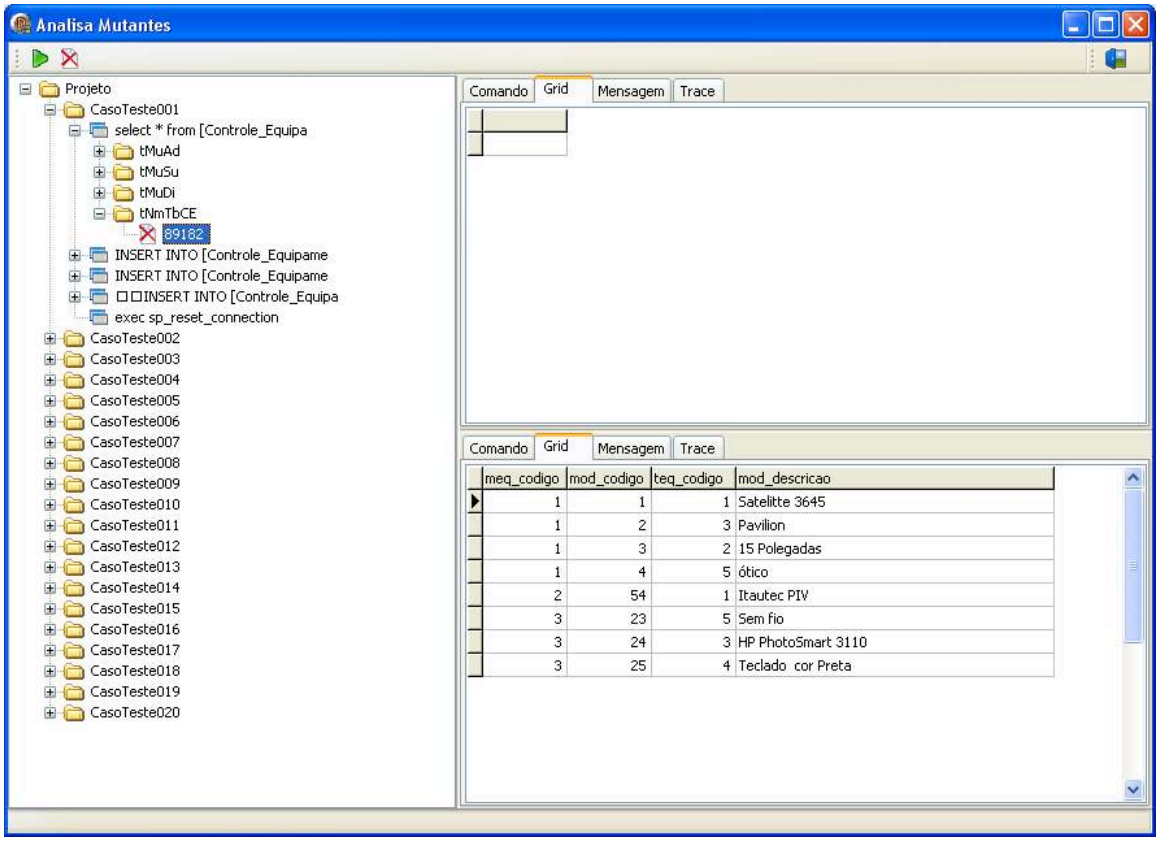

Figura A - 5 - Análise de Mutantes

A.2.4. Análise dos Resultados

A interface para a análise dos mutantes é apresentada na Figura A-5. Os dados dos mutantes são armazenados na base de testes, permitindo que cada grupo de comandos e seus mutantes sejam re-executados a qualquer tempo. Na estrutura de árvore ilustrada ao lado esquerdo da interface (Figura A-5) é possível visualizar as seguintes informações:

- identificação do grupo de comandos: demonstra qual foi a seqüência de comandos executados para aquele caso de teste;
- operadores de mutação aplicados ao comando;
- mutantes gerados para cada operador;
- resultado da avaliação do mutante (morto: ícone em vermelho, vivo: ícone de identificação em verde).

Do lado direito da mesma interface (Figura A-5) é possível visualizar diversos dados relativos à análise dos mutantes, tais como:

- comando SQL Original x Mutantes: é possível analisar os dados que foram utilizados para a execução do comando, o local exato da aplicação do operador e a sintaxe completa dos mesmos;
- *grid*: é possível visualizar o resultado do comando e suas respectivas persistências de dados para o comando original e seu mutante;
- mensagem: é possível visualizar as mensagens que o sistema gerenciador de banco de dados emitiu após a execução do comando para o comando original e seu mutante;
- *trace*: é possível visualizar informações com relação ao número de tuplas selecionadas, modificadas, número de operações de escrita e de leitura, entre outras informações.

Diversos tipos de relatórios podem ser visualizados através da ferramenta, tais como:

- Informações do projeto, como operadores aplicados, quantidade de mutantes gerados;

- escores de mutação por operador, projeto, caso de teste, comando SQL ou categoria de operadores;
- maior e menor índice de mortos, vivos e equivalentes por operador, projeto, caso de teste, comando SQL ou categoria.

A.3. Características Técnicas da Ferramenta

A ferramenta pode ser utilizada com qualquer linguagem hospedeira que utilize comandos SQL embutidos, não se restringindo a uma linguagem de programação especifica como C ou Java, porém, a base de dados acessada pela linguagem hospedeira deve estar obrigatoriamente em um sistema gerenciador de banco de dados Microsoft SQL Server 2005, pois a ferramenta utiliza um mecanismo de *trace* pertencente a este SGBD.

A ferramenta foi desenvolvida em Borland Delphi 2006 e sua base de dados foi desenvolvida em MS SQL Server 2005.

Apêndice B - Exemplos dos Operadores de Mutação Aplicados a SQL

Neste apêndice são demonstrados exemplos da aplicação dos operadores de mutação aplicados aos comandos SQL. Textos explicativos são inseridos para elucidar o funcionamento dos comandos.

Este apêndice é organizado em cinco seções que representam as cinco categorias de operadores de mutação e cada subseção representa um operador de mutação.

Para facilitar a visualização da aplicação dos operadores nos comando SQL os exemplos dos mutantes exibirão apenas a linha modificada.

B.1. Operadores de Mutação para Operadores de SQL

Nesta seção são apresentados os exemplos para os operadores de mutação da categoria operadores de mutação para operadores de SQL.

B.1.1. Troca de Operador Matemático (tOpMt)

```
SELECT [PurchaseOrderID],
    [PurchaseOrderDetailID],
    [ ProductID],
    QTD_TOTAL = [ReceivedQty]+[RejectedQty]
FROM [AdventureWorks].[Purchasing].[PurchaseOrderDetail]
```

Para o comando acima é possível aplicar três permutas de tOpMt, criando três mutantes:

Mutante 1 (tAdSu)

```
    QTD_TOTAL = [ReceivedQty]-[RejectedQty]
```

Mutante 2 (tAdMu)

```
    QTD_TOTAL = [ReceivedQty]*[RejectedQty]
```

Mutante 3 (tAdDi)

```
    QTD_TOTAL = [ReceivedQty]/[RejectedQty]
```

O comando acima executa uma seleção na tabela *PurchaseOrderDetail*, trazendo três atributos da tabela e um campo calculado que representa a soma do atributo *ReceivedQty* e *RejectedQty*.

No mutante 1 pode-se notar que a seleção trará os mesmos três atributos selecionados no comando original, mas o campo calculado representará a subtração do atributo *RejectedQty* ao valor do atributo *ReceivedQty*.

Já no mutante 3, nota-se que a seleção trará os mesmos três atributos selecionados no comando original, mas o campo calculado representará a divisão do atributo *ReceivedQty* ao valor do atributo *RejectedQty*.

B.1.2. Troca de Operador de Comparação (tOpCp)

```
SELECT [SalesOrderID], [RevisionNumber],[OrderDate],
    [DueDate],[ShipDate],[Status],[OnlineOrderFlag],
    [SalesOrderNumber],[PurchaseOrderNumber],
    [AccountNumber],[CustomerID],[ContactID],
    [SalesPersonID],[TerritoryID],[BillToAddressID],
    [ShipToAddressID],[ShipMethodID],[CreditCardID],
    [CreditCardApprovalCode],[CurrencyRateID],
    [SubTotal],[TaxAmt],[Freight],[TotalDue],
    [Comment],[rowguid],[ModifiedDate], ABS(Convert(int, [ShipDate] -
[OrderDate]))
FROM [AdventureWorks].[Sales].[SalesOrderHeader]
WHERE [TaxAmt]+[Freight] > 0.05*[TotalDue]
AND   ABS(Convert(int, [ShipDate] - [OrderDate])) < 10

Mutante 1 (tMaIg)
WHERE [TaxAmt]+[Freight] = 0.05*[TotalDue]

Mutante 2 (tMaDes)
WHERE [TaxAmt]+[Freight] <> 0.05*[TotalDue]
```

O comando acima executa uma seleção na tabela *SalesOrderHeader*, trazendo 27 atributos da tabela e a criação de um novo campo, sendo representando pela diferença entre os atributos *ShipDate* e *OrderDate*. Há duas condições para esta seleção: a primeira é o resultado da soma dos atributos *TaxAmt* e *Freight* ser maior que o resultado do atributo *TotalDue* multiplicado por 0.05, a segunda condição definida é que a diferença entre os atributos *ShipDate* e *OrderDate* seja menor do que dez.

No mutante 1 pode-se notar que a seleção trará os mesmos vinte e sete atributos selecionados no comando original, mas a condição mostra que a soma dos atributos *TaxAmt* e *Freigh*t devem ter seu resultado igual ao resultado do atributo *TotalDue* multiplicado por 0.05.

No mutante 2 nota-se que a seleção trará os mesmos vinte e sete atributos selecionados no comando original, mas a condição mostra que a soma dos atributos *TaxAmt* e *Freight* devem ter seu resultado diferente do resultado do atributo *TotalDue* multiplicado por 0.05.

B.1.3. Troca de Operador Conjuntivo (tOpCj)

```
SELECT [SalesOrderID], [RevisionNumber],[OrderDate],
    [DueDate],[ShipDate],[Status],[OnlineOrderFlag],
    [SalesOrderNumber],[PurchaseOrderNumber],
    [AccountNumber],[CustomerID],[ContactID],
    [SalesPersonID],[TerritoryID],[BillToAddressID],
    [ShipToAddressID],[ShipMethodID],[CreditCardID],
    [CreditCardApprovalCode],[CurrencyRateID],
    [SubTotal],[TaxAmt],[Freight],[TotalDue],
    [Comment],[rowguid],[ModifiedDate], ABS(Convert(int, [ShipDate] -
[OrderDate]))
FROM [AdventureWorks].[Sales].[SalesOrderHeader]
WHERE [TaxAmt]+[Freight] > 0.05*[TotalDue]
AND   ABS(Convert(int, [ShipDate] - [OrderDate])) < 10

Mutante 1 (tAndOr)
OR   ABS(Convert(int, [ShipDate] - [OrderDate])) < 10
```

O comando acima executa uma seleção na tabela *SalesOrderHeader*, trazendo vinte e sete atributos da tabela e a criação de um novo campo, sendo representando pela diferença entre os atributos *ShipDate* e *OrderDate*. Há duas condições para esta seleção: a primeira é o resultado da soma dos atributos *TaxAmt* e *Freight* ser maior que o resultado do atributo *TotalDue* multiplicado por 0.05, a segunda condição definida é que a diferença entre os atributos *ShipDate* e *OrderDate* seja menor do que dez.

No mutante 1 observa-se que a seleção trará os mesmos vinte e sete atributos selecionados no comando original, porém a condição mostra que a soma dos atributos

TaxAmt e *Freight* deve ter seu resultado maior do que o resultado do atributo *TotalDue* multiplicado por 0.05 ou que a diferença entre os atributos *ShipDate* e *OrderDate* seja menor do que dez.

B.1.4. Troca de Operador Lógico (tOpLg)

```
SELECT p1.ProductModelID
FROM Production.Product p1
GROUP BY p1.ProductModelID
HAVING MAX(p1.ListPrice) >= ALL
(SELECT 2 * AVG(p2.ListPrice)
FROM Production.Product p2
WHERE p1.ProductModelID = p2.ProductModelID)

Mutante 1 (tAllAny)
HAVING MAX(p1.ListPrice) >= ANY
```

O comando acima executa a seleção na tabela *Product* p1, retornando o atributo *ProductModelID* agrupado pelo mesmo com o critério de seleção, sendo o valor máximo do atributo ListPrice maior ou igual a todos do critério de seleção da tabela Product p2 onde a média do atributo ListPrice é multiplicado por dois.

No mutante 1 nota-se que a consulta retorna os mesmos atributos da consulta original, porém, há a substituição do comando ALL pelo comando ANY que irá recuperar os registros da consulta principal que satisfaçam a comparação com qualquer registro recuperado na subconsulta.

B.1.5. Inserção de Operador de Negação (iNot)

```
SELECT *
FROM Sales.SpecialOffer
WHERE (DiscountPct > 0.22
OR    MaxQty is null)
AND   (Year(EndDate) - Year(StartDate)) < 5 AND   Category <> 'Customer'
ORDER BY MaxQty DESC, Description, DiscountPct DESC

Mutante 1 (iNot)
OR    MaxQty is NOT null)
```

O comando acima executa uma seleção na tabela *SpecialOffer*, retornando todos os seus atributos, atendendo às condições de que o valor do atributo *DiscountPct* seja maior que

0.22, ou de que o valor do atributo *MaxQty* seja nulo, de que a diferença dos atributos *EndDate* e *StartDate* seja menor do que cinco e também de que o atributo *Category* seja diferente de '*Customer*'.

No mutante 1 pode-se notar que a seleção retornará todos os mesmos atributos, porém a condição do atributo *MaxQty* não deverá ser nulo.

B.1.6. Retirada de Operador de Negação (rNot)

```
SELECT *
FROM Sales.SpecialOffer
WHERE (DiscountPct > 0.22
OR    MaxQty is not null)
AND   (Year(EndDate) - Year(StartDate)) < 5 AND   Category <> 'Customer'
ORDER BY MaxQty DESC, Description, DiscountPct DESC

Mutante 1 (iNot)
OR    MaxQty is null)
```

O comando acima executa uma seleção na tabela *SpecialOffer*, retornando todos os seus atributos, atendendo às condições de que o valor do atributo *DiscountPct* seja maior que 0.22, ou de que o valor do atributo *MaxQty* não seja nulo, de que a diferença dos atributos *EndDate* e *StartDate* seja menor do que cinco e também de que o atributo *Category* seja diferente de '*Customer*'.

No mutante 1 observa-se que a seleção retornará todos os mesmos atributos, porém a condição do atributo *MaxQty* deverá ser nulo.

B.2. Operadores de Mutação de Miscelânea

Nesta seção são apresentados os exemplos para os operadores de miscelânea.

B.2.1. Troca de Posição de Atributo (tPoAt)

```
SELECT [SalesOrderID], [RevisionNumber],[OrderDate],
    [DueDate],[ShipDate],[Status],[OnlineOrderFlag]
FROM [AdventureWorks].[Sales].[SalesOrderHeader]

Mutante 1
        , [CustomerID],[DueDate],[ShipDate],[Status],[OnlineOrderFlag]
```

Mutante 2
, [DueDate], [Status], [ShipDate], [OnlineOrderFlag]

O comando acima executa uma seleção na tabela *SalesOrderHeader*, retornando sete atributos.

No mutante 1 nota-se que a seleção retornará os mesmos atributos na condição original e o atributo *CustomerID* também será selecionado.

No mutante 2 nota-se que a seleção retornará os mesmos atributos na condição original, porém, a ordem dos atributos está sendo alterada.

B.2.2. Retirada de Atributo (rAtr)

SELECT [SalesOrderID], [RevisionNumber],[OrderDate]
,[DueDate],[ShipDate],[Status],[OnlineOrderFlag]
FROM [AdventureWorks].[Sales].[SalesOrderHeader]

Mutante 1
, [ShipDate], [Status],[OnlineOrderFlag]

Mutante 2
, [DueDate], [Status], [OnlineOrderFlag]

O comando acima executa uma seleção na tabela *SalesOrderHeader*, retornando sete atributos.

No mutante 1 pode-se notar que a seleção está retirando da seleção o atributo *DueDate*.

No mutante 2 nota-se também que a seleção está retirando o atributo *ShipDate*.

B.2.3. Inserção de Atributo (iAtr)

SELECT [SalesOrderID], [RevisionNumber],[OrderDate]
,[DueDate],[ShipDate],[Status],[OnlineOrderFlag]
FROM [AdventureWorks].[Sales].[SalesOrderHeader]

Mutante 1
, [ShipDate], [Status],[OnlineOrderFlag]

Mutante 2
, [DueDate], [Status], [OnlineOrderFlag]],[CurrencyRateID]

O comando acima executa uma seleção na tabela *SalesOrderHeader*, retornando sete atributos.

No mutante 1 nota-se que a seleção está retirando da seleção o atributo *DueDate*.

No mutante 2 nota-se que a seleção está retirando o atributo *ShipDate* e incluindo o atributo *CurrencyRateID*.

B.2.4. Troca de Posição de Valor (tPoVr)

```
INSERT INTO [AdventureWorks].[dbo].[EmployeeOne]
        ([LastName], [FirstName], [Title])
    VALUES
        ('José','Silva','housekeeper')

Mutante 1
        ('Silva','José', 'housekeeper')

Mutante 2
     ('José', 'housekeeper', 'Silva')
```

O comando acima executa uma inserção de valores na tabela *EmployeeOne*.

No mutante 1 e 2 observa-se que é trocada a ordem de inserção dos dados dos atributos.

B.2.5. Troca de Valor (tVr)

```
INSERT INTO [AdventureWorks].[dbo].[EmployeeOne]
        ([LastName], [FirstName], [Title])
    VALUES
        ('José','Silva','housekeeper')

Mutante 1
        ('José', null, 'housekeeper')

Mutante 2
     ('José', 'Silva', 234)
```

O comando acima executa uma inserção de valores na tabela *EmployeeOne.*

No mutante 1 nota-se que é trocado o tipo do dado de alfanumérico para nulo.

No mutante 2 nota-se que é trocado o tipo do dado alfanumérico por um valor numérico.

B.2.6. Troca de Tipo de Variável (tTpVar)

```
SET NOCOUNT ON

DECLARE @vendor_id int, @vendor_name nvarchar(50),
@message varchar(80), @product nvarchar(50)

Mutante 1
DECLARE @vendor_id nvarchar(50), @vendor_name int,
```

O comando acima declara quatro variáveis sendo @vendor_id do tipo inteiro, @vendor_name, @message e @product do tipo caracter.

No mutante 1 nota-se que o tipo das variáveis @vendor_id e @vendor_name estão sendo alteradas para o tipo caracter e inteiro respectivamente.

B.2.7. Troca de Nome de Tabela (tNmTb)

```
SELECT p.Name, sod.SalesOrderID
FROM Production.Product p
FULL OUTER JOIN Sales.SalesOrderDetail sod
ON p.ProductID = sod.ProductID
WHERE p.ProductID IS NULL
OR sod.ProductID IS NULL
ORDER BY p.Name

Mutante 1
        FROM Sales.SalesOrderDetail p
        FULL OUTER JOIN Production.Product sod
```

O comando acima faz uma seleção nas tabelas *Product* e *SalesOrderDetail* retornando dois atributos contendo produtos que não possuem vendas relacionadas ao atributo *ProductID*.

No mutante 1 nota-se que a seleção retornará os mesmos dois campos, porém, a ordem do relacionamento das tabelas foi alterada.

B.2.8. Troca de Nome de ROLE (tNmRole)

```
CREATE ROLE Records_customer
CREATE ROLE Records_vendor
GRANT SELECT to Records_customer
```

GRANT INSERT to Records_vendor

Mutante 1
CREATE ROLE Records_customer
CREATE ROLE Records_vendor
GRANT SELECT to Records_vendor
GRANT INSERT to Records_vendor

O comando acima cria duas funções nomeadas *Records_customer* e *Records_vendor*, atribuindo privilégios de seleção e inserção, respectivamente, a elas.

No mutante 1 pode-se notar que os dois privilégios são atribuídos à função *Records_vendor*.

B.2.9. Inserção de ROLE (iRole)

CREATE ROLE Records_customer
CREATE ROLE Records_vendor
GRANT SELECT to Records_customer
GRANT INSERT to Records_vendor

Mutante 1
CREATE ROLE Records_customer
CREATE ROLE Records_vendor
GRANT SELECT to Records_customer, Records_vendor
GRANT INSERT to Records_vendor

O comando acima cria duas funções nomeadas *Records_customer* e *Records_vendor*, atribuindo privilégios de seleção e inserção, respectivamente, a elas.

No mutante 1 pode-se notar que o privilégio de seleção é atribuído à função *Records_vendor* e *records_vendor*.

B.2.10. Retirada de ROLE (rRole)

CREATE ROLE Records_customer
CREATE ROLE Records_vendor
GRANT SELECT to Records_customer, Records_vendor

Mutante 1
GRANT SELECT Records_vendor

O comando acima cria duas funções nomeadas *Records_customer* e *Records_vendor*, atribuindo privilégios de seleção a ambas.

No mutante 1 observa-se que o privilégio de seleção é atribuído apenas à função *Records_vendor*.

B.2.11. Troca de Nome de Cursor (tNmCursor)

```
DECLARE product_cursor CURSOR FOR
SELECT v.Name
FROM Purchasing.ProductVendor pv, Production.Product v
WHERE pv.ProductID = v.ProductID AND
pv.VendorID = @vendor_id-- Variable value from the outer cursor

OPEN product_cursor
FETCH NEXT FROM product_cursor INTO @product

IF @@FETCH_STATUS <> 0
PRINT '        <<None>>'

WHILE @@FETCH_STATUS = 0
BEGIN

SELECT @message = '        ' + @product
PRINT @message
FETCH NEXT FROM product_cursor INTO @product

END

CLOSE product_cursor
DEALLOCATE product_cursor
```

O comando acima declara o cursor product_cursor, o carrega na memória e o percorre até o final, exibindo na tela o atributo *name* da tabela production.product, e desaloca da memória o cursor product_cursor

```
 Mutante 1
DEALLOCATE vendor_cursor
```

No mutante 1 o comando declara o cursor product_cursor, o carrega na memória e o percorre até o final, exibindo na tela o atributo *name* da tabela production.product, e desaloca da memória o cursor vendor_cursor.

B.2.12. Troca de Função de Agregação (tFuAg)

```
Select SalesOrderID, Minimo = MIN(UnitPrice),
     MAX(UnitPrice) as Maximo, AVG(UnitPrice) as 'Preco Medio',
     'Nr. Itens' = COUNT(UnitPrice), SUM( OrderQTY) QTDE_Pedida,
     'Variacao %'= (MAX(UnitPrice)/MIN(UnitPrice))*100
from Sales.SalesOrderDetail
GROUP BY SalesOrderID
HAVING MAX(UnitPrice)/MIN(UnitPrice) > 1
ORDER BY [Preco Medio] Desc, SalesOrderID , Maximo

Mutante 1
Select SalesOrderID, Minimo = MAX(UnitPrice),

Mutante 2
Select SalesOrderID, Minimo = AVG(UnitPrice),
```

O comando acima mostra a seleção da tabela *SalesOrderDetail* agrupada pelo atributo *SalesOrderID* com o critério de apresentar os registros com a variação do valor máximo e mínimo do atributo *UnitPrice* que for maior do que 1, retornando os campos valor mínimo, máximo, média e quantidade sobre o atributo *UnitPrice*, a soma da quantidade do atributo *OrderQty* e a variação do valor máximo e mínimo do atributo *UnitPrice*.

No mutante 1 percebe-se que o comando de seleção foi alterado para retornar o valor máximo do atributo *UnitPrice* ao invés do seu valor mínimo.

No mutante 2 percebe-se que o comando de seleção foi alterado para retornar a média do atributo *UnitPrice* ao invés do seu valor mínimo.

B.2.13. Troca de Intersecção (tInSec)

```
SELECT *
FROM HumanResources.Employee e1
INTERSECT
SELECT *
FROM HumanResources.Employee e2

Mutante 1 (tInEx)
EXCEPT

Mutante 2 (tInUn)
UNION
```

O comando acima mostra a seleção da tabela *Employee* trazendo todos os seus atributos, o comando *intersect* entre as duas seleções retornará todos os elementos que intercedem

tanto na primeira seleção quanto na segunda, sendo que os registros duplicados serão omitidos.

No mutante 1 observa-se que a utilização do comando *except* retornará todas as tuplas da seleção e1, mas que não estejam presentes na seleção e2.

No mutante 2 observa-se que a utilização do comando *union* retornará todas as tuplas da seleção e1 e também todas as tuplas da seleção e2, omitindo os registros duplicados.

B.2.14. Troca de Join (tJoin)

```
SELECT Manager = G.[EmployeeID], G.[Title], G.[Gender], Employee =
E.[EmployeeID], E.[Title], E.[Gender]
FROM [AdventureWorks].[HumanResources].[Employee] G
        INNER JOIN [AdventureWorks].[HumanResources].[Employee] E
ON G.EmployeeID = E.ManagerID
WHERE G.Gender = 'M'
AND   E.Gender = 'F'

Mutante 1 (tJoinInLe)
LEFT OUTER JOIN
[AdventureWorks].[HumanResources].[Employee] E

Mutante 2 (tJoinInRi)
RIGHT OUTER JOIN [AdventureWorks].[HumanResources].[Employee] E
```

O comando acima mostra a seleção da Tabela *Employee* trazendo seis atributos com a condição de relacionamento entre os atributos *EmplyeeID* e *ManagerID* outra condição para a seleção é que somente o atributo *Gender* tenha seu valor igual a ‘M’ ou ‘F’.

No mutante 1 nota-se a troca do comando INNER JOIN pelo comando LEFT OUTER JOIN, onde todas as linhas da tabela *Employee G* serão apresentadas mesmo que não sejam localizadas na seleção da *Tabela Employee E*.

No mutante 2 percebe-se a troca do comando INNER JOIN pelo comando RIGHT OUTER JOIN, onde todas as linhas da tabela *Employee G* serão apresentadas somente se localizadas na seleção da tabela *Employee E*.

B.3. Operadores de Mutação para Fluxo de Dados

Nesta seção são apresentados os exemplos para os operadares de mutação para Fluxo de Dados.

B.3.1. Troca de Bloco de Comandos (tBlCmEstRep)

```
IF @cost <= @compareprice
BEGIN
   PRINT 'These products can be purchased for less than'
END
ELSE
   PRINT 'The prices for all products in this category exceed'

Mutante 1 (tBlCmEstRep)
IF @cost <= @compareprice
BEGIN
   PRINT 'The prices for all products in this category exceed'
END
ELSE
   PRINT 'These products can be purchased for less than
```

O comando acima imprime a mensagem 'These products can be purchased for less' no caso da condição cost <= @compareprice ser verdadeira, caso contrário a mensagem exibida será 'The prices for all products in this category exceed'.

No mutante 1 oberva-se a troca do bloco de comandos, sendo então exibida na tela a mensagem 'These products can be purchased for less' no caso da condição cost <= @compareprice não ser verdadeira.

B.3.2. Retirada de Comando do Bloco de Repetição/Condição (rCmBlRep)

```
IF @cost <= @compareprice
BEGIN
   PRINT 'These products can be purchased for less'
END
ELSE
   PRINT 'The prices for all products in this category exceed'

Mutante 1 (tBlCmEstRep)
IF @cost <= @compareprice
BEGIN
   PRINT 'The prices for all products in this category exceed
   $'+ RTRIM(CAST(@compareprice AS varchar(20)))+'.'
END
```

O comando acima imprime a mensagem 'These products can be purchased for less' no caso da condição cost <= @compareprice ser verdadeira, caso contrário, a mensagem exibida será 'The prices for all products in this category exceed'.

No mutante 1 observa-se a retirada do bloco de comandos no caso da condição cost <= @compareprice não ser verdadeira.

B.3.3. Inserção de Comando do Bloco de Repetição/Condição (iCmBlRep)

```
IF @cost <= @compareprice
BEGIN
   PRINT 'These products can be purchased for less than'
END
ELSE
   cost = 15
   PRINT 'The prices for all products in this category exceed'

Mutante 1 (tBlCmEstRep)
IF @cost <= @compareprice
BEGIN
   PRINT 'These products can be purchased for less than'
   cost = 15
END
ELSE
   cost = 15
   PRINT 'The prices for all products in this category exceed'
```

O comando acima imprime a mensagem 'These products can be purchased for less' no caso da condição cost ≤ @compareprice ser verdadeira, caso contrário, a mensagem exibida será 'The prices for all products in this category exceed' e será atribuído o valor 15 à variável cost.

No mutante 1 observa-se que o valor 15 será atribuído a variável *cost* independente de satisfazer ou não a condição.

B.3.4. Troca de Posição do Leave no Bloco de Comandos (tPosLeave)

O LEAVE será substituído neste exemplo pelo BREAK, pois, no MS SQL Server a função definida no padrão SQL3 do LEAVE é feita pelo BREAK.

```
USE AdventureWorks;
GO
WHILE (SELECT AVG(ListPrice) FROM Production.Product) < $300
BEGIN
  UPDATE Production.Product
```

```
    SET ListPrice = ListPrice * 2
  SELECT MAX(ListPrice) FROM Production.Product
  IF (SELECT MAX(ListPrice) FROM Production.Product) > $500
    BREAK
  IF (SELECT MAX(ListPrice) FROM Production.Product) < $300
       PRINT 'Product less than 300''
END
PRINT 'Too much for the market to bear';

Mutante 1 (tPosLeave)
USE AdventureWorks;
GO
WHILE (SELECT AVG(ListPrice) FROM Production.Product) < $300
BEGIN
  UPDATE Production.Product
    SET ListPrice = ListPrice * 2
  SELECT MAX(ListPrice) FROM Production.Product
  IF (SELECT MAX(ListPrice) FROM Production.Product) > $500
  IF (SELECT MAX(ListPrice) FROM Production.Product) < $300
       PRINT 'Product less than 300''
    BREAK
END
PRINT 'Too much for the market to bear';
```

No comando acima nota-se uma estrutura de repetição onde ocorre a atualização da tabela Production. Product enquanto a média do campo ListPrice da mesma tabela for menor que $300. Caso o valor máximo deste campo seja maior que $500 a estrutura de repetição é encerrada e a mensagem 'Too much for the market to bear' é exibida.

No mutante 1 percebe-se uma estrutura de repetição, onde ocorre a atualização da tabela Production.Product enquanto a média do campo *ListPrice* da mesma tabela for menor que $300. Caso o valor máximo deste campo seja menor que $300 a estrutura de repetição é encerrada após a exibição da mensagem 'Product less than 300'' e em seguida é exibida a mensagem 'Too much for the market to bear' é exibida.

B.3.5. Retirada de LEAVE (rLeave)

```
USE AdventureWorks;
GO
WHILE (SELECT AVG(ListPrice) FROM Production.Product) < $300
BEGIN
  UPDATE Production.Product
    SET ListPrice = ListPrice * 2
  SELECT MAX(ListPrice) FROM Production.Product
```

```
  IF (SELECT MAX(ListPrice) FROM Production.Product) > $500
    BREAK
END
PRINT 'Too much for the market to bear';

Mutante 1 (tPosLeave)
USE AdventureWorks;
GO
WHILE (SELECT AVG(ListPrice) FROM Production.Product) < $300
BEGIN
  UPDATE Production.Product
    SET ListPrice = ListPrice * 2
  SELECT MAX(ListPrice) FROM Production.Product
  IF (SELECT MAX(ListPrice) FROM Production.Product) > $500

END
PRINT 'Too much for the market to bear';
```

No comando acima observa-se uma estrutura de repetição, onde ocorre a atualização da tabela Production.Product enquanto a média do campo *ListPrice* da mesma tabela for menor que $300. Caso o valor máximo deste campo seja maior que $500 a estrutura de repetição é encerrada e a mensagem 'Too much for the market to bear' é exibida.

No mutante 1 observa-se uma estrutura de repetição, onde ocorre a atualização da tabela Production.Product enquanto a média do campo *ListPrice* da mesma tabela for menor que $300. Caso o valor máximo deste campo for maior que $500 a estrutura de repetição não será encerrada, devida a ausência do BREAK.

B.3.6. Inserção de LEAVE (iLeave)

```
USE AdventureWorks;
GO
WHILE (SELECT AVG(ListPrice) FROM Production.Product) < $300
BEGIN
  UPDATE Production.Product
    SET ListPrice = ListPrice * 2
  SELECT MAX(ListPrice) FROM Production.Product
  IF (SELECT MAX(ListPrice) FROM Production.Product) > $500
    BREAK
  IF (SELECT MAX(ListPrice) FROM Production.Product) < $300
       PRINT 'Product less than 300''

END
PRINT 'Too much for the market to bear';
```

```
Mutante 1 (tPosLeave)
USE AdventureWorks;
GO
WHILE (SELECT AVG(ListPrice) FROM Production.Product) < $300
BEGIN
  UPDATE Production.Product
    SET ListPrice = ListPrice * 2
  SELECT MAX(ListPrice) FROM Production.Product
  IF (SELECT MAX(ListPrice) FROM Production.Product) > $500
    BREAK
  IF (SELECT MAX(ListPrice) FROM Production.Product) < $300
       PRINT 'Product less than 300''
    BREAK
END
PRINT 'Too much for the market to bear';
```

No comando acima podemos observar uma estrutura de repetição, onde ocorre a atualização da tabela Production. Product enquanto a média do campo *ListPrice* da mesma tabela for menor que $300. Caso o valor máximo deste campo seja maior que $500 a estrutura de repetição é encerrada e a mensagem 'Too much for the market to bear' é exibida.

No mutante 1 observa-se uma estrutura de repetição, onde ocorre a atualização da tabela Production.Product enquanto a média do campo *ListPrice* da mesma tabela for menor que $300. Caso o valor máximo deste campo seja maior que $500 a estrutura de repetição é encerrada e a mensagem 'Too much for the market to bear' é exibida. Caso o valor máximo deste campo seja menor que $300 a estrutura de repetição é encerrada após a exibição da mensagem 'Product less than 300'' e em seguida é exibida a mensagem 'Too much for the market to bear'.

A.4. Operadores de Mutação para Controle de Transações.

Nesta seção são apresentados os exemplos para os operadares de mutação para Controle de Transações.

B.4.1. Inserção de COMMIT (iCm)

```
SELECT DISTINCT [GroupName]
FROM [AdventureWorks].[HumanResources].[Department]

SELECT [GroupName], Count(*) as QTDE
```

```
FROM [AdventureWorks].[HumanResources].[Department]
Group By GroupName
```

Mutante 1 (iCm)

```
SELECT DISTINCT [GroupName]
FROM [AdventureWorks].[HumanResources].[Department]
COMMIT
SELECT [GroupName], Count(*) as QTDE
FROM [AdventureWorks].[HumanResources].[Department]
Group By GroupName
```

Mutante 2 (iCm)

```
SELECT DISTINCT [GroupName]
FROM [AdventureWorks].[HumanResources].[Department]
SELECT [GroupName], Count(*) as QTDE
FROM [AdventureWorks].[HumanResources].[Department]
Group By GroupName
COMMIT
```

O comando acima realiza a seleção na Tabela Department trazendo o atributo *GroupName* sem suas duplicações e logo em seguida ocorre outra seleção da tabela Department trazendo os atributos *GroupName* e o campo QTDE informando a quantidade de registros agrupadas pelo campo *GroupName*.

No mutante 1 nota-se a realização das duas seleções, porém, há o comando COMMIT dentre elas - sua função é efetivar as transações correntes antes de aplicar a segunda seleção.

No mutante 2 observa-se a utilização do comando COMMIT somente no final da segunda seleção - sua função é, após aplicar as duas seleções, efetivar as transações correntes.

B.4.2. Retirada de COMMIT (rCm)

```
SELECT DISTINCT [GroupName]
FROM [AdventureWorks].[HumanResources].[Department]
COMMIT
SELECT [GroupName], Count(*) as QTDE
FROM [AdventureWorks].[HumanResources].[Department]
Group By GroupName
```

Mutante 1 (rCm)

```
SELECT DISTINCT [GroupName]
FROM [AdventureWorks].[HumanResources].[Department]
SELECT [GroupName], Count(*) as QTDE
FROM [AdventureWorks].[HumanResources].[Department]
Group By GroupName
```

O comando acima realiza a seleção na Tabela Department trazendo o atributo *GroupName* sem suas duplicações e em seguida ocorre o comando COMMIT que tem a finalidade de efetivar a realização da transação, a segunda seleção realizada na tabela Department traz os atributos *GroupName* e o campo QTDE informando a quantidade de registros agrupada pelo campo *GroupName*.

No mutante 1 percebe-se a retirada do comando COMMIT.

B.4.3. Inserção de ROLLBACK (iRb)

```
DELETE FROM [AdventureWorks].[dbo].[EmployeeOne]
    WHERE ([LastName] = 'Silva'

Mutante 1 (iRb)
DELETE FROM [AdventureWorks].[dbo].[EmployeeOne]
    WHERE ([LastName] = 'Silva'
ROLLBACK

Mutante 2 (iRb)
ROLLBACK
DELETE FROM [AdventureWorks].[dbo].[EmployeeOne]
    WHERE ([LastName] = 'Silva'
ROLLBACK
```

O comando acima mostra a exclusão de um registro na Tabela EmployeeOne, a condição para esta exclusão deve satisfazer o atributo *LastNam*e igual a ‘Silva’.

No mutante 1 percebe-se que o comando de exclusão é o mesmo do código original, porém, o comando ROLLBACK é aplicado no final do critério de exclusão, fazendo com que a ação executada seja desfeita logo após o código de exclusão.

No mutante 2 percebe-se também que o comando de exclusão é o mesmo do código original, porém, o comando ROLLBACK é aplicado no início e final do critério de exclusão. O ROLLBACK aplicado no início faz com que a última transação realizada

seja desfeita e em seguida executa o código de exclusão. O ROLLBACK no final do código faz com que a ação executada seja desfeita logo após o código de exclusão.

B.4.4. Retirada de ROLLBACK (rRb)

```
DELETE FROM [AdventureWorks].[dbo].[EmployeeOne]
    WHERE ([LastName] = 'Silva'
ROLLBACK

Mutante 1 (rRb)
DELETE FROM [AdventureWorks].[dbo].[EmployeeOne]
    WHERE ([LastName] = 'Silva'
```

O comando acima mostra a exclusão de um registro na Tabela EmployeeOne, onde a condição para esta exclusão deve satisfazer o atributo *LastName* igual a 'Silva' e em seguida é aplicado o comando ROLLBACK desfazendo a efetivação desta transação.

No mutante 1 observa-se que o comando de exclusão é o mesmo do código original porém o comando ROLLBACK não é utilizado deixando a transação sem ser efetivada.

B.4.5. Troca de COMMIT por ROLLBACK (tCmRb)

```
DELETE FROM [AdventureWorks].[dbo].[EmployeeOne]
    WHERE ([LastName] = 'Silva'
COMMIT

Mutante 1 (iRb)
DELETE FROM [AdventureWorks].[dbo].[EmployeeOne]
    WHERE ([LastName] = 'Silva'
ROLLBACK
```

O comando acima mostra a exclusão de um registro na Tabela EmployeeOne, onde a condição para esta exclusão deve satisfazer o atributo *LastName* igual a 'Silva' e em seguida é aplicado o comando COMMIT para efetivar esta transação.

No mutante 1 nota-se que o comando de exclusão é o mesmo do código original, porém, o comando COMMIT é substituído pelo comando ROLLBACK que é utilizado no final do critério de exclusão desfazendo a última transação realizada.

B.4.6. Troca de ROLLBACK por COMMIT (tRbCm)

```
DELETE FROM [AdventureWorks].[dbo].[EmployeeOne]
    WHERE ([LastName] = 'Silva'
ROLLBACK


Mutante 1 (iRb)
DELETE FROM [AdventureWorks].[dbo].[EmployeeOne]
    WHERE ([LastName] = 'Silva'
COMMIT
```

O comando acima mostra a exclusão de um registro na tabela EmployeeOne, onde a condição para esta exclusão deve satisfazer o atributo LastName igual a 'Silva' e em seguida é aplicado o comando ROLLBACK para desfazer esta transação.

No mutante 1 observa-se que o comando de exclusão é o mesmo do código original, porém, o comando ROLLBACK é substituído pelo comando COMMIT que é utilizado no final do critério de exclusão, efetivando a última transação realizada.

B.4.7. Troca de Nome do SAVEPOINT (tNmSP)

```
SAVEPOINT SP1
DELETE FROM [AdventureWorks].[dbo].[EmployeeOne]
    WHERE ([LastName] = 'Silva'
SAVEPOINT SP2
GRANT EXECUTE ON xp_readmail TO public
ROLLBACK SP1

Mutante 1 (itNmSP)
ROLLBACK SP2
```

O *script* acima mostra a criação de um ponto de transação SP1, a execução de um comando de exclusão, a criação de um ponto de transação SP2, a garantia de privilégio de execução sobre a tabela xp_readmail para o usuário *public* e o comando para desfazer as transações até o ponto SP1.

No mutante 1 observa-se que a única transação desfeita foi a garantia de privilégio.

B.4.8. Troca de Permissão (tPerm)

```
SET TRANSACTION READ WRITE

Mutante1 (tPerm)
SET TRANSACTION READ ONLY
```

O comando acima mostra que todos os comandos executados para a transação definida poderão fazer transações de leitura a escrita.

No mutante 1 percebe-se que apenas as transações de escritas serão permitidas.

B.4.9. Troca de Privilégio (tPriv)

use master
GRANT EXECUTE ON xp_readmail TO public
use Adventureworks

Mutante 1 (tExAl)
GRANT ALL PRIVELEGE. ON xp_readmail TO public

O comando acima garante o privilégio de execução sobre a tabela xp_readmail para o usuário public.

No mutante 1 é dado privilégio total sobre a tabela xp_readmail para o usuário *public*.

B.4.10. Troca de GRANT por REVOKE (tGrRe)

use master
GRANT EXECUTE ON xp_readmail TO public
use Adventureworks

Mutante 1 (tGrRe)
REVOKE EXECUTE ON xp_readmail TO public

O comando acima garante o privilégio de execução sobre a tabela xp_readmail para o usuário *public*.

No mutante 1 vê-se o privilégio sendo retirado.

B.4.11. Troca de REVOKE por GRANT (tReGr)

use master
REVOKE EXECUTE ON xp_readmail TO public
use Adventureworks

Mutante 1 (tGrRe)
GRANT EXECUTE ON xp_readmail TO public

O comando acima declaradamente retira (nega) o privilégio de execução sobre a tabela xp_readmail para o usuário *public*.

No mutante 1 vê-se o privilégio de execução sobre a tabela xp_readmail ser dado para o usuário *public*.

B.4.12. Troca de Nome de Usuário (tNmUsr)

```
use master
REVOKE EXECUTE ON xp_readmail TO public
use Adventureworks

Mutante 1 (tGrRe)
GRANT EXECUTE ON xp_readmail TO Mike
```

O comando acima declaradamente retira (nega) o privilégio de execução sobre a tabela xp_readmail para o usuário *public*.

No mutante 1 observa-se o privilégio de execução sobre a tabela xp_readmail ser dado para o usuário Mike.

B.5. Operadores de Mutação para Funções, Procedimentos e Triggers

Nesta seção são apresentados os exemplos para os operadares de mutação para Funções, procedimentos, *view* e *Triggers*.

B.5.1. Troca de Nome da Função, Procedimento, View ou Trigger (tNm)

```
IF OBJECT_ID ('Sales.reminder1', 'TR') IS NOT NULL
  DROP TRIGGER Sales.reminder1
Go
CREATE TRIGGER Customer_Sales
ON Sales.Customer
AFTER INSERT, UPDATE
AS RAISERROR ('Notify Customer Relations', 16, 10)


Mutante 1 (tNm)
  DROP TRIGGER Customer_Sales
```

O comando acima verifica se o *trigger* Sales reminder1 existe, se existir, ele será descartado e o *trigger* Customer_sales será criado.

No mutante 1 nota-se que, após a verificação, a *trigger* a ser descartada será a Customer_Sales.

B.5.2. Troca Posição de Retorno da Função (tPoReFu)

```
CREATE FUNCTION dbo.ISOweek (@DATE datetime)
RETURNS int
WITH EXECUTE AS CALLER
AS
BEGIN
    DECLARE @ISOweek int
    SET @ISOweek= DATEPART(wk,@DATE)+1
        -DATEPART(wk,CAST(DATEPART(yy,@DATE) as CHAR(4))+'0104')
--Special cases: Jan 1-3 may belong to the previous year
    IF (@ISOweek=0)
        SET @ISOweek=dbo.ISOweek(CAST(DATEPART(yy,@DATE)-1
            AS CHAR(4))+'12'+ CAST(24+DATEPART(DAY,@DATE) AS
CHAR(2)))+1
--Special case: Dec 29-31 may belong to the next year
    IF ((DATEPART(mm,@DATE)=12) AND
        ((DATEPART(dd,@DATE)-DATEPART(dw,@DATE))>= 28))
        SET @ISOweek=1
    RETURN(@ISOweek)
END;
GO
```

Mutante 1 (tPoReFu)

```
CREATE FUNCTION dbo.ISOweek (@DATE datetime)
RETURNS int
WITH EXECUTE AS CALLER
AS
BEGIN
    DECLARE @ISOweek int
    SET @ISOweek= DATEPART(wk,@DATE)+1
        -DATEPART(wk,CAST(DATEPART(yy,@DATE) as CHAR(4))+'0104')
--Special cases: Jan 1-3 may belong to the previous year
    RETURN(@ISOweek)
    IF (@ISOweek=0)
        SET @ISOweek=dbo.ISOweek(CAST(DATEPART(yy,@DATE)-1
            AS CHAR(4))+'12'+ CAST(24+DATEPART(DAY,@DATE) AS
CHAR(2)))+1
--Special case: Dec 29-31 may belong to the next year
    IF ((DATEPART(mm,@DATE)=12) AND
        ((DATEPART(dd,@DATE)-DATEPART(dw,@DATE))>= 28))
        SET @ISOweek=1
END;
GO
```

No comando acima notamos que o retorno da função é feito após a execução de todos os comandos inseridos nela.

No Mutante 1 o retorno da função é feito antes da execução de todos os comandos da função.

B.5.3. Retirada de Retorno da Função (rReFu)

```
CREATE FUNCTION dbo.ISOweek (@DATE datetime)
RETURNS int
WITH EXECUTE AS CALLER
AS
BEGIN
    DECLARE @ISOweek int
    SET @ISOweek= DATEPART(wk,@DATE)+1
        -DATEPART(wk,CAST(DATEPART(yy,@DATE) as CHAR(4))+'0104')
--Special cases: Jan 1-3 may belong to the previous year
    IF (@ISOweek=0)
        SET @ISOweek=dbo.ISOweek(CAST(DATEPART(yy,@DATE)-1
            AS CHAR(4))+'12'+ CAST(24+DATEPART(DAY,@DATE) AS
CHAR(2)))+1
--Special case: Dec 29-31 may belong to the next year
    IF ((DATEPART(mm,@DATE)=12) AND
        ((DATEPART(dd,@DATE)-DATEPART(dw,@DATE))>= 28))
        SET @ISOweek=1
    RETURN(@ISOweek)
END;
GO

Mutante 1 (tPoReFu)
CREATE FUNCTION dbo.ISOweek (@DATE datetime)
RETURNS int
WITH EXECUTE AS CALLER
AS
BEGIN
    DECLARE @ISOweek int
    SET @ISOweek= DATEPART(wk,@DATE)+1
        -DATEPART(wk,CAST(DATEPART(yy,@DATE) as CHAR(4))+'0104')
--Special cases: Jan 1-3 may belong to the previous year
    IF (@ISOweek=0)
        SET @ISOweek=dbo.ISOweek(CAST(DATEPART(yy,@DATE)-1
            AS CHAR(4))+'12'+ CAST(24+DATEPART(DAY,@DATE) AS
CHAR(2)))+1
--Special case: Dec 29-31 may belong to the next year
    IF ((DATEPART(mm,@DATE)=12) AND
```

```
        ((DATEPART(dd,@DATE)-DATEPART(dw,@DATE))>= 28))
        SET @ISOweek=1
END;
GO
```

No comando acima observa-se que o retorno da função é feito após a execução de todos os comandos inseridos nela.

No mutante 1 nota-se que o retorno da função foi retirado da declaração da mesma.

B.5.4. Troca de Parâmetros da Procedure (tPaPro)

```
EXECUTE HumanResources.usp_GetEmployees 'Ackerman', 'Pilar';

Mutante 1 (tPaPro)
EXECUTE HumanResources.usp_GetEmployees 'Pilar',
'Ackerman';
```

O comando acima demonstra a chamada da procedure usp_GetEmployees com a passagem dos parâmetros 'Pilar', 'Ackerman'.

No mutante 1 observa-se que a passagem dos parâmetros esta invertida.

B.5.5. Troca de Evento na Trigger (tEv)

```
CREATE TRIGGER reminder1
ON Sales.Customer
AFTER UPDATE
AS RAISERROR ('Notify Customer Relations', 16, 10)

Mutante 1 (tUpIn)
AFTER INSERT
```

O comando acima prepara a *trigger* reminder1 para ser executada após o evento de UPDATE da tabela Sales.Customer.

No mutante 1 o comportamento é alterado para o evento INSERT da tabela Sales.Customer.

Apêndice C - Resultados dos Experimentos

Este apêndice visa ao detalhamento dos resultados obtidos nos experimentos. A sua organização é feita pelo resultado dos experimentos Equipamento e Material, que representam a geração, execução e avaliação da aplicação dos mutantes no Cenário 1 (Seção 3.3.1), seguido pelos resultados do Experimento Controle que utiliza o Cenário 2 (Seção 3.3.2).

A descrição da condução dos experimentos encontra-se na Seção 5.4, 5.5 e 5.6 deste trabalho.

É chamado de sub-operador a permuta de um operador específico, por exemplo, o operador troca de Operador Matemático possui uma permuta troca de adição por subtração, que nas próximas seções receberá o nome de suboperador *troca de adição por subtração.*

Todos os experimentos foram realizados em uma versão do aplicativo (software em testes) que possuiam defeitos já identificados por testes funcionais. Desta forma, foi possíve comparar se as bases de dados e os operadores de mutação foram capazes de revelar tais defeitos.

C.1. Experimento Equipamentos

A Tabela C-1 representa os resultados obtidos na execução dos testes para as duas bases de dados utilizadas neste experimento. Podemos visualizar a quantidade de mutantes gerados, a quantidade de mortos, vivos e equivalentes e seus respectivos escores de mutação para cada caso de teste executado neste cenário.

Tabela C-1 - Resultados do Experimento Equipamentos.

Resultados do Experimento Equipamentos								
	Base 1				**Base 2**			
Qtde de Comandos	79				85			
Caso de Testes	**Mutantes**				**Mutantes**			
	Mortos	**Vivos**	**Equivalentes**	***Escores***	**Mortos**	**Vivos**	**Equivalentes**	***Escores***
1	18	1	0	0,947	18	1	0	0,947
2	20	0	0	1	12	0	0	1
3	14	0	0	1	9	0	0	1
4	18	0	0	1	10	8	0	0,556

5	27	1	0	0,964	24	4	0	0,857
6	15	1	0	0,938	15	1	0	0,938
7	15	1	0	0,938	15	1	0	0,938
8	18	0	0	1	13	3	1	0,813
9	3	0	0	1	0	3	0	0
10	6	11	0	0,353	7	10	0	0,412
11	107	20	34	0,843	115	63	0	0,646
12	114	17	41	0,87	222	111	0	0,667
13	16	9	0	0,64	16	7	2	0,696
14	126	21	38	0,857	126	59		0,681
15	9	2	1	0,818	14	3	2	0,824
16	9	2	1	0,818	9	3		0,75
17	11	5	7	0,688	11	5	7	0,688
18	16	5	4	0,762	16	7	2	0,696
19	3	40	1	0,07	0	43	1	0
20	13	0	0	1	12	1		0,923
Total	578	136	127		664	333	15	
Total de Mutantes	841				1012			

A Tabela C-2 representa a quantidade de defeitos encontrados com a execução dos casos de testes projetados para a aplicação que acessa o banco de dados. A coluna “Não Revelados” indica apenas que a base de dados em questão não revelou a mesma quantidade de defeitos revelados pela outra base no mesmo caso de testes e, em nenhum momento, podemos dizer quantos defeitos ainda existem não revelados na aplicação em teste.

Os defeitos reais apresentados na tabela C-2 são defeitos encontrados na aplicação por testes feitos na aplicação antes da execução deste experimento, desta forma, podemos visualizar se as instâncias das bases de dados foram capazes de revelar os defeitos que já existiam no aplicativo.

Tabela C-2 - Defeitos Encontrados a partir da Execução dos Casos de Teste.

Defeitos Reais – Experimento Equipamentos				
Caso de Testes	Base 1		Base 2	
	Revelados	Não Revelados	Revelados	Não Revelados
1	1		1	
2	0		0	
3	0		0	
4	0		0	
5	0		0	
6	2		0	2
7	1	1	2	
8	1		1	
9	0		0	
10	1	1	2	
11	0		0	
12	0	1	0	1
13	0		0	
14	1		1	
15	0		0	
16	1		1	
17	1		1	
18	1		1	
19	0		0	
20	0		0	
Total	10	3	10	3

A Tabela C-3 demonstra os operadores e seus escores de mutação para cada caso de teste. Para os operadores que possuem suboperadores ou que necessitam de alguma informação sobre o tipo de troca, a representação no campo Operador é feita pelo suboperador ou pelo suboperador, no intuito de dar exatidão e riqueza de informação.

A análise dos operadores de mutação deve ser feita considerando cada caso de teste como um cenário único e a comparação com outros casos de testes torna-se irrelevante, pois a persistência de dados em cada caso de testes pode ser diferente, o que alterará completamente o resultado em relação à eficiência de cada operador de mutação.

Tabela C-3 - Operadores e seus respectivos Escores de Mutação no Experimento Equipamentos.

Operadores Experimento Equipamentos					
Base 1			**Base 2**		
Caso de Teste	**Operador**	***Escores* de Mutação**	**Caso de Teste**	**Operador**	***Escores* de Mutação**
1	tMuAd	1	1	tNmTb	1
1	tMuDi	1	1	tMuAd	1
1	tMuSu	1	1	tMuDi	1
1	tNmTb	1	1	tMuSu	1
1	tAt	0,6666667	1	tAt	0,6666667
2	tAt	1	2	tAt	1
2	tPoAt	1	2	tNmTb	1
2	tNmTb	1	2	tOrAnd	1
2	tOrAnd	1	3	tAt	1
3	tAt	1	3	tNmTb	1
3	tPoAt	1	3	tVr	1
3	tNmTb	1	4	tAt	0,8333333
3	tVr	1	4	tIgDes	0,5
4	tAt	1	4	tIgMa	0,5
4	tIgDes	1	4	tIgMag	0,5
4	tIgMa	1	4	tIgMe	0
4	tIgMag	1	4	tIgMeg	0
4	tIgMe	1	4	tNmTb	1
4	tIgMeg	1	5	tAt	0,9285714
4	tNmTb	1	5	tNmTb	1
5	tAt	1	5	tIgDes	1
5	tNmTb	1	5	tIgMeg	0
5	tIgDes	1	5	tIgMe	0,5
5	tIgMeg	0,5	5	tIgMag	1
5	tIgMe	1	5	tIgMa	1
5	tIgMag	1	5	tOrAnd	1
5	tIgMa	1	6	tAt	1
5	tOrAnd	1	6	tNmTb	1
6	tAt	1	6	tIgDes	1
6	tNmTb	1	6	tIgMa	1
6	tIgDes	1	6	tIgMag	1
6	tIgMa	1	6	tIgMe	1
6	tIgMag	1	6	tIgMeg	0
6	tIgMe	1	7	tAt	1
6	tIgMeg	0	7	tNmTb	1
7	tAt	1	7	tIgDes	1
7	tNmTb	1	7	tIgMeg	0
7	tIgDes	1	7	tIgMe	1
7	tIgMeg	0	7	tIgMag	1
7	tIgMe	1	7	tIgMa	1
7	tIgMag	1	8	tOrAnd	0,6666667
7	tIgMa	1	8	tAt	1

8	tOrAnd	1	8	tNmTb	1
8	tAt	1	9	tOrAnd	0
8	tPoAt	1	10	tOrAnd	0,5
8	tNmTb	1	10	tIgDes	0,5
9	tOrAnd	1	10	tIgMa	0,5
10	tOrAnd	1	10	tIgMag	0,5
10	tIgDes	0	10	tIgMe	0
10	tIgMa	0	10	tIgMeg	0
10	tIgMag	0	10	tAndOr	1
10	tIgMe	0	11	tOrAnd	0,9166667
10	tIgMeg	0	11	tIgDes	0,9473684
10	tAndOr	0	11	tIgMeg	0
11	tOrAnd	1	11	tIgMe	0,8947368
11	tNmTb	0,5882353	11	tIgMag	0,0526316
11	tIgDes	1	11	tIgMa	0,9473684
11	tIgMeg	0	11	tNmTb	0,6470588
11	tIgMe	1	11	tAt	1
11	tIgMag	0	11	tInLef	0
11	tIgMa	1	11	tInRig	0,1428571
11	tAt	1	11	tAndOr	1
11	tInLef	0	12	tOrAnd	1
11	tInRig	0,1428571	12	tIgDes	1
12	tOrAnd	1	12	tIgMa	1
12	tIgDes	1	12	tIgMag	0
12	tIgMa	1	12	tIgMe	1
12	tIgMag	0	12	tIgMeg	0
12	tIgMe	1	12	tNmTb	0,6176471
12	tIgMeg	0	12	tAt	1
12	tNmTb	0,5882353	12	tInRig	0,0714286
12	tAt	1	12	tInLef	0
12	tInRig	0	12	tMeDe	1
12	tInLef	0	12	tMeIg	1
12	tMeDe	1	12	tMeMa	1
12	tMeIg	1	12	tMeMag	1
12	tMeMa	1	12	tMeMeg	0
12	tMeMag	1	12	tAndOr	1
12	tMeMeg	0	13	tAt	0,8
12	tAndOr	1	13	tNmTb	1
13	tAt	0,8	13	tOrAnd	0
13	tNmTb	1	13	tIgDes	0,5
13	tOrAnd	0	13	tIgMeg	0,5
13	tIgDes	0,5	13	tIgMe	0,5
13	tIgMeg	0,5	13	tIgMag	0,5
13	tIgMe	0,5	13	tIgMa	0,5
13	tIgMag	0,5	14	tOrAnd	1
13	tIgMa	0,5	14	tIgMa	1
14	tOrAnd	1	14	tIgMag	0,05
14	tIgMa	1	14	tIgMe	1

14	tIgMag	0,05	14	tIgMeg	0,1
14	tIgMe	1	14	tIgDes	1
14	tIgMeg	0,1	14	tNmTb	0,5882353
14	tIgDes	1	14	tAt	1
14	tNmTb	0,5882353	14	tInLef	0
14	tAt	1	14	tInRig	0
14	tInLef	0	14	tMeMag	1
14	tInRig	0	14	tMeMa	1
14	tMeMag	1	14	tMeIg	1
14	tMeMa	1	14	tMeDe	1
14	tMeIg	1	14	tMeMeg	0
14	tMeDe	1	14	tPoVr	1
14	tMeMeg	0	14	tAndOr	1
14	tPoVr	1	15	tAt	0,5714286
14	tAndOr	1	15	tAndOr	0,6666667
15	tAt	0,8333333	15	iNotBetween	0,6666667
15	tAndOr	0,5	15	tNmTb	1
15	iNotBetween	0,5	15	tMuAd	1
15	tNmTb	1	15	tMuSu	1
16	tAt	0,8333333	15	tMuDi	1
16	tAndOr	0,5	16	tAt	0,8333333
16	tNmTb	1	16	tAndOr	0,5
16	iNotBetween	0,5	16	tNmTb	1
17	tAt	0,8888889	16	iNotBetween	0,5
17	tNmTb	1	17	tAt	0,8888889
17	tIgDes	0	17	tNmTb	1
17	tIgMeg	0	17	tIgDes	0
17	tIgMe	0	17	tIgMeg	0
17	tIgMag	0	17	tIgMe	0
17	tIgMa	0	17	tIgMag	0
17	tOrAnd	0	17	tIgMa	0
18	tAt	0,8	17	tOrAnd	0
18	tNmTb	1	18	tAt	0,8
18	tIgDes	0,5	18	tNmTb	1
18	tIgMa	0,5	18	tIgDes	0,5
18	tIgMag	0,5	18	tIgMa	0,5
18	tIgMe	0,5	18	tIgMag	0,5
18	tIgMeg	0,5	18	tIgMe	0,5
18	tAndOr	0	18	tIgMeg	0,5
18	tOrAnd	0	18	tAndOr	0
19	tIgDes	0,125	18	tOrAnd	0
19	tIgMeg	0,125	19	tIgDes	0
19	tIgMe	0,125	19	tIgMeg	0
19	tIgMag	0	19	tIgMe	0
19	tIgMa	0	19	tIgMag	0
19	tOrAnd	0	19	tIgMa	0
19	tAndOr	0	19	tOrAnd	0

20	tAt	1	19	tAndOr	0
20	tNmTb	1	20	tAt	0,8333333
20	iNotBetween	1	20	tNmTb	1
20	rNot	1	20	iNotBetween	1
20	tOrAnd	1	20	rNot	1
20	tFuAgCMA	1	20	tOrAnd	1
20	tAndOr	1	20	tFuAgCMA	1
			20	tAndOr	1

C.2. Experimento Materiais

A Tabela C-4 representa os resultados obtidos na execução dos testes para as duas bases de dados utilizadas neste experimento. Podemos visualizar a quantidade de mutantes gerados, a quantidade de mortos, vivos e equivalentes e seus respectivos escores de mutação relativos a cada caso de teste executado neste cenário.

Tabela C-4 - Resultados do Experimento Materiais.

Resultados do Experimento Materiais								
	Base 1				**Base 2**			
Qtde de Mutantes	79				85			
Caso de Testes	**Mutantes**				**Mutantes**			
	Mortos	**Vivos**	**Equivalentes**	***Escores***	**Mortos**	**Vivos**	**Equivalentes**	***Escores***
1	2	0		1	2	0		1
2	3	0		1	3	0		1
3	3	0		1	3	0		1
4	15	2		0,882	15	2		0,882
5	15	2		0,882	15	2		0,882
6	27	8		0,771	27	8		0,771
7	49	8		0,86	49	8		0,86
8	2	5		0,286	6	1		0,857
9	3	0		1	3	0		1
10	3	0		1	3	0		1
11	15	2		0,882	15	2		0,882
12	2	0		1	2	0		1
13	12	4		0,75	14	2		0,875
14	43	0		1	23	0		1
15	1	0		1	1	0		1
16	1	0		1	1	0		1
17	5	0		1	5	0		1
18	31	4		0,886	33	2		0,943
19	6	0		1	6	0		1
20	2	0		1	2	0		1
Total	240	35	0		228	27	0	
Total de Mutantes	275				255			

A Tabela C-5 representa a quantidade de defeitos encontrados com a execução dos casos de testes projetados para a aplicação que acessa o banco de dados. A coluna "Não Revelados" indica apenas que a base de dados em questão não revelou a mesma quantidade de defeitos revelado pela outra base no mesmo caso de testes e, em nenhum momento, podemos dizer quantos defeitos ainda existem não revelados na aplicação em teste.

Os defeitos reais apresentados na tabela C-5 são defeitos encontrados na aplicação por testes feitos na aplicação antes da execução deste experimento, desta forma, podemos visualizar se as instâncias das bases de dados foram capazes de revelar os defeitos que já existiam no aplicativo.

Tabela C-5 - Quantidade de Defeitos Encontrados no Experimento Materiais.

Defeitos Reais Experimento Materiais				
Caso de Testes	**Base 1**		**Base 2**	
	Revelados	**Não Revelados**	**Revelados**	**Não Revelados**
1	1		1	
2	2		2	
3	1		1	
4	1	1	2	
5				
6				
7				
8		1	1	
9	1		1	
10	1		1	
11				
12				
13				
14				
15	1		1	
16	1		1	
17				
18	1		1	
19				
20	1		1	
Total	11	2	13	0

A Tabela C-6 demonstra os operadores e seus escores de mutação para cada caso de teste. Para os operadores que possuem suboperadores ou que necessitam de alguma informação

sobre o tipo de troca, a representação no campo Operador é feita pelo suboperador ou pelo suboperador, no intuito de dar exatidão e riqueza de informação.

A análise dos operadores de mutação deve ser feita considerando cada caso de teste como um cenário único e a comparação com outros casos de testes torna-se irrelevante, pois a persistência de dados em cada caso de testes pode ser diferente, o que alterará completamente o resultado em relação à eficiência de cada operador de mutação.

A coluna "Não Revelados" indica apenas que a base de dados em questão não revelou a mesma quantidade de defeitos revelados pela outra base no mesmo caso de testes e, em nenhum momento podemos dizer quantos defeitos ainda existem não revelados.

Tabela C-6 - Operadores e seus respectivos Escores de Mutação no Experimento Materiais.

Operadores Experimento Materiais					
Base 1			Base 2		
Caso de Teste	Operador	*Escores* de Mutação	Caso de Teste	Operador	*Escores* de Mutação
1	tPoAt	1	1	tPoAt	1
2	tPoAt	1	2	tPoAt	1
2	tPoVr	1	2	tPoVr	1
3	tNmTb	1	3	tNmTb	1
4	tAt	1	4	tAt	1
4	tIgDes	1	4	tIgDes	1
4	tIgMa	1	4	tIgMa	1
4	tIgMag	1	4	tIgMag	1
4	tIgMe	0,5	4	tIgMe	0,5
4	tIgMeg	0,5	4	tIgMeg	0,5
4	tNmTb	1	4	tNmTb	1
4	tPoVr	1	4	tPoVr	1
5	tAt	1	5	tAt	1
5	tIgMa	1	5	tIgMa	1
5	tIgDes	1	5	tIgDes	1
5	tIgMeg	0,5	5	tIgMeg	0,5
5	tIgMe	0,5	5	tIgMe	0,5
5	tIgMag	1	5	tIgMag	1
5	tNmTb	1	5	tNmTb	1
6	tOrAnd	1	6	tOrAnd	1
6	tNmTb	1	6	tNmTb	1
6	tIgMag	0,3333333	6	tIgMag	0,3333333
6	tIgMe	1	6	tIgMe	1
6	tIgMeg	0,3333333	6	tIgMeg	0,3333333
6	tIgDes	1	6	tIgDes	1
6	tIgMa	1	6	tIgMa	1

6	tJoinInRig	0	6	tJoinInRig	0
6	tJoinInLef	0	6	tJoinInLef	0
6	tAt	1	6	tAt	1
6	tFuAgCMA	1	6	tFuAgCMA	1
7	tNmTb	1	7	tNmTb	1
7	tOrAnd	1	7	tOrAnd	1
7	tIgMa	1	7	tIgMa	1
7	tIgDes	1	7	tIgDes	1
7	tIgMeg	0,6666667	7	tIgMeg	0,6666667
7	tIgMe	1	7	tIgMe	1
7	tIgMag	0,6666667	7	tIgMag	0,6666667
7	tJoinInRig	0	7	tJoinInRig	0
7	tAt	1	7	tAt	1
7	tJoinInLef	0	7	tJoinInLef	0
7	tVr	1	7	tVr	1
7	iNotIsnull	1	7	iNotIsnull	1
8	tNmTb	1	8	tNmTb	1
8	tMeDe	0	8	tMeDe	0
8	tMeIg	0	8	tMeIg	1
8	tMeMa	0	8	tMeMa	1
8	tMeMag	0	8	tMeMag	1
8	tMeMeg	0	8	tMeMeg	1
9	tNmTb	1	9	tNmTb	1
9	rNot	1	9	rNot	1
10	tNmTb	1	10	tNmTb	1
10	rNot	1	10	rNot	1
11	tIgDes	1	11	tIgDes	1
11	tIgMa	1	11	tIgMa	1
11	tIgMeg	0,3333333	11	tIgMeg	0,3333333
11	tIgMag	1	11	tIgMag	1
11	tIgMe	1	11	tIgMe	1
11	tNmTb	1	11	tNmTb	1
12	tFuAgCMA	1	12	tFuAgCMA	1
13	tAt	1	13	tAt	1
13	tIgMa	0,5	13	tIgMa	1
13	tIgDes	0,5	13	tIgDes	1
13	tIgMe	1	13	tIgMe	1
13	tIgMag	1	13	tIgMag	1
13	tIgMeg	0	13	tIgMeg	0
13	iNotIn	1	13	iNotIn	1
13	tFuAgCMA	1	13	tFuAgCMA	1
13	tNmTb	1	13	tNmTb	1
14	tIgDes	1	14	tIgDes	1
14	tIgMa	1	14	tIgMa	1
14	tIgMeg	1	14	tIgMeg	1
14	tIgMag	1	14	tIgMag	1
14	tIgMe	1	14	tIgMe	1
14	tOrAnd	1	14	tAt	1

14	tAt	1	14	tVr	1
14	tFuAgCMA	1	15	tOrAnd	1
14	tMaDe	1	16	tOrAnd	1
14	tVr	1	17	tAt	1
14	tMaIg	1	17	tInSecEI	1
14	tMaMag	1	17	tInSecEU	1
14	tMaMeg	1	17	tNmTb	1
15	tOrAnd	1	18	tAt	1
16	tOrAnd	1	18	tNmTb	1
17	tAt	1	18	iNotIn	1
17	tInSecEI	1	18	tIgMeg	1
17	tInSecEU	1	18	tIgMe	1
17	tNmTb	1	18	tIgMag	1
18	tAt	1	18	rNot	1
18	tNmTb	1	18	tIgMa	1
18	iNotIn	1	18	tIgDes	1
18	tIgMeg	1	18	tFuAgCMA	1
18	tIgMe	1	18	tMaDe	0
18	tIgMag	1	18	tMaIg	1
18	rNot	1	18	tMaMag	0
18	tIgMa	1	18	tMaMeg	1
18	tIgDes	1	18	tVr	1
18	tFuAgCMA	1	19	tNmTb	1
18	tMaDe	0	19	tFuAgCMA	1
18	tMaMeg	0	19	tAt	1
18	tMaMag	0	20	tOrAnd	1
18	tMaIg	0			
18	tVr	1			
19	tNmTb	1			
19	tFuAgCMA	1			
19	tAt	1			
20	tOrAnd	1			

C.3. Experimento Controle

A tabela abaixo representa os resultados obtidos na execução dos testes para as duas bases de dados utilizadas neste experimento. Podemos visualizar a quantidade de mutantes gerados, a quantidade de mortos, vivos e seu respectivo escore de mutação relativo a cada caso de teste executado neste cenário. Este experimento não apresentou nenhum mutante equivalente.

Tabela C-7 - Resultados do Experimento Controle.

Resultados do Experimento Controle						
	Base 1			Base 2		
Comando SQL	Mutantes			Mutantes		
	Mortos	Vivos	*Escores*	Mortos	Vivos	*Escores*
1	12	2	0,8571429	12	2	0,8571429
2	7	0	1	2	5	0,2857143
3	28	2	0,9333333	6	24	0,2
4	3	0	1	3	0	1
5	6	2	0,75	6	2	0,75
6	9	1	0,9	9	1	0,9
7	10	2	0,8333333	10	2	0,8333333
8	6	0	1	5	1	0,8333333
9	0	0	0	0	0	0
10	0	0	0	0	0	0
11	10	0	1	10	0	1
12	3	0	1	3	0	1
13	8	0	1	8	0	1
14	2	1	0,6666667	2	1	0,6666667
15	4	0	1	4	0	1
16	7	1	0,875	2	6	0,25
17	1	0	1	1	0	1
18	1	0	1	0	1	0
19	2	0	1	1	1	0,5
20	9	0	1	5	4	0,5555556
21	2	0	1	2	0	1
22	1	0	1	1	0	1
23	4	0	1	2	2	0,5
24	5	3	0,625	5	3	0,625
25	5	3	0,625	5	3	0,625
26	5	0	1	2	3	0,4
27	8	0	1	2	6	0,25
28	4	0	1	1	3	0,25
29	4	0	1	1	3	0,25
30	7	0	1	1	6	0,1428571
31	8	0	1	3	5	0,375
32	7	0	1	4	3	0,5714286
33	4	0	1	2	2	0,5
34	5	0	1	5	0	1
35	1	0	1	1	0	1
36	2	0	1	2	0	1
37	2	0	1	2	0	1
38	2	0	1	2	0	1
39	0	0	0	0	0	0
40	0	0	0	0	0	0
41	2	0	1	2	0	1
42	3	0	1	3	0	1
Total de Mutantes	226			226		

A Tabela C-8 demonstra os operadores e seus escores de mutação para cada caso de teste. Para os operadores que possuem suboperadores ou que necessitam de alguma informação sobre o tipo de troca, a representação no campo Operador é feita pelo suboperador ou pelo suboperador, no intuito de dar exatidão e riqueza de informação.

Tabela C-8 - Operadores e seus respectivos Escores de Mutação no Experimento Controle.

Operadores Experimento Controle			
Base 1		**Base 2**	
Operador	***Escores* de Mutação**	**Operador**	***Escores* de Mutação**
tIgDes	0,9807692	tIgDes	0,76923077
tAt	0,9130435	tAt	0,19047619
tNmTb	1	tNmTb	0,95833333
tMeIg	1	tMeIg	0,46666667
tMeMa	1	tMeMa	0,46666667
tMaIg	0,9166667	tMaIg	0,66666667
tMaMe	1	tMaMe	0,75
tAdSu	1	tPoAt	0,6
tPoAt	1	tAdSu	0,8
tInLef	0,7142857	tInLef	0,57142857
tInRig	0,5714286	tnRig	0,14285714
tVr	1	tVr	1
tFuAgAC	1	tFuAgAC	1
tFuAgAMA	0,5	tFuAgAMA	0
tFuAgAMI	0,5	tFuAgAMI	0
tFuAgAS	0,5	tFuAgAS	0
tFuAgMIA	0,6666667	tFuAgMIA	0
tFuAgMIC	1	tFuAgMIC	0,33333333
tFuAgMIMA	1	tFuAgMIMA	0
tFuAgMIS	1	tFuAgMIS	0
iNotLike	1	iNotLike	1
tPoVr	1	tPoVr	1
rNot	1	rNot	1
iNotBetween	1	iNotBetween	1
iNotIsnull	0	iNotIsnull	0