Universidade Federal de Pernambuco
Centro de Informática - CIn

Graduação em Engenharia da Computação

Um Catálogo de Regras de Boa Formação para Consultas SQL

João Paulo Siqueira Lins

Trabalho de Graduação

Recife
12 de Julho de 2017

Universidade Federal de Pernambuco
Centro de Informática - CIn

João Paulo Siqueira Lins

Um Catálogo de Regras de Boa Formação para Consultas SQL

Trabalho apresentado ao Programa de Graduação em Engenharia da Computação do Centro de Informática - CIn da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: *Robson Fidalgo*

Recife
12 de Julho de 2017

Agradecimentos

Agradeço primeiramente ao meu pai, minha mãe, e minha irmã, pelo apoio que sempre tive durante toda a minha vida. Com quem eu sempre pude contar, independente da situação.

Agradeço ao professor Robson Fidalgo e também à Edson Alves pelo apoio e orientação durante a escrita deste trabalho.

Agradeço à todos os professores e funcionários do CIn, onde passei os últimos 6 anos. O centro e todos que estão nele são responsáveis pela formação que tive e sou muito grato por isso.

Agradeço também às pessoas com quem passei muito tempo na faculdade, gente que entrou no curso junto comigo. Tive o prazer de compartilhar esta jornada junto com vocês. Agradeço à Artur, Leonardo, Marina, Moisés, Pedro, Rafael, Rebeca, Rodrigo, Thaís, Thiago e Walter. Obrigado TT!

Agradeço também aos meus amigos mais próximos, os quais conheço a mais de 10 anos. É bom saber que posso contar com eles a qualquer hora. Sempre estiveram comigo nas vitórias e também nos momentos mais difíceis. Meus agradecimentos à Gabriel, Gustavo, João Guilherme, João Rafael, Paulo, Pedro, Raphael e Thiago. Obrigado por tudo F9!

Resumo

A interface utilizada para interagir com bancos de dados é o SQL. Assim como qualquer outra linguagem usada para enviar instruções ao computador, códigos em SQL podem conter erros em sua produção. Podemos separar os erros em três tipos : sintático, semântico, e violação de regras de boa formação de consultas. Apesar dos Sistemas de Gerenciamento de Banco de Dados (SGBD) serem muito bons para lidar com erros sintáticos, o mesmo não pode se dizer para os erros semânticos, muito menos para análise de boas práticas. Este trabalho propõe um catálogo de regras que contribuem para garantir a boa formação de consultas SQL. Além disso, avaliar como os SGBD se comportam com essas regras violadas também é um objetivo. Uma pesquisa foi realizada para coletar quais tipos de análises já haviam sido feitas na literatura, além de programas comerciais de análise estática de código, que contém regras que ajudam a manter a qualidade de código. Após a pesquisa, foi feita a filtragem e a classificação das regras. E por último a avaliação nos SGBD com a utilização de exemplos que representam as regras. O catálogo possui 8 casos de erros semânticos e 24 regras de boas práticas. No resultado da avaliação, foi confirmado que os SGBD não emitem nenhum tipo de aviso para a grande maioria dos casos onde uma orientação deveria ter sido dada ao usuário.

Palavras-chave: SQL, Erros Semânticos, *Feedback*, Sistema de Gerenciamento de Bancos de Dados

Abstract

SQL is a language used as an interface to interact with databases. As any other language used to send instructions to a computer, SQL statements can have errors in your design. It is possible to categorize these errors in three classes : Syntactic, Semantic, and violation of good practice rules. Although DBMS - Database Management Systems - are good for dealing with syntax erros, the same is not true for semantic errors or analysis of good practice rules. This essay proposes a rule catalog that contributes to assure the creation of well-formed SQL statements. Also an analysis is made about DBMS and their behavior against statements that does not follow the rules from the catalog. Articles about semantic errors were studied in search of rules already stated that could help the catalog's creation. Commercial software that does static code analysis are researched as well. Then, the rules found are filtered and sorted by classes. SQL statements are created from the rules, and are executed in a DBMS. The catalog consists of 8 rules about semantic errors, and 24 good practice rules. The DBMS analyzed does not emit any warning or error for almost all the rules where an feedback should be given to the user.

Keywords: SQL, Semantic Errors, Feedback, Database Management Systems

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Listagens

CAPÍTULO 1

Introdução

Neste capítulo, são apresentados o contexto no qual o trabalho está inserido, o problema e a motivação da pesquisa, os objetivos e perguntas da pesquisa, qual foi a delimitação do escopo, o método empregado e a estrutura da dos outros capítulos desta dissertação.

1.1 Contexto e motivação

No processo de desenvolvimento de *software* os desenvolvedores precisam lidar com problemas que surgem durante a escrita de código fonte. Estes problemas podem ser sintáticos (e.g., impedem a execução de código), semânticos (e.g., um bloco de código que foi escrito para realizar determinada tarefa, mas não a realiza corretamente) ou descumprimento de regras de boa formação (e.g., presença de termos desnecessários na consulta). Para evitar estes problemas, os desenvolvedores contam com ferramentas que os auxiliam a detectar estes erros. No contexto de consultas SQL em banco de dados, esta atividade é de responsabilidade dos Sistema de Gerenciamento de Banco de Dados (SGBD).

Os SGBD funcionam muito bem para informar ao usuário quando um erro de sintaxe é cometido. Porém, não são tão efetivos quando o sentido de um comando precisa ser interpretado. Comandos que contêm falhas na sua semântica são mais difíceis de serem corrigidos, pois a percepção do erro pode demorar para acontecer ou o erro só é evidente em alguns casos.

Um exemplo de erro sintático capturado pelos SGBD é apresentado na listagem 1.1. Neste exemplo, a ordem das cláusulas está invertida. Portanto a consulta não é executada e o SGBD emite uma mensagem de erro.

Listagem 1.1 Exemplo de erro sintático.

```
FROM Tabela
SELECT *;
```

Erros semânticos passam despercebidos pelos SGBD, como o exemplo apresentado na listagem 1.2. A comparação de um valor nulo deve ser feita com a palavra reservada IS (IS

NULL), e não usando o operador de igualdade (= NULL). Isso é necessário pois o resultado da comparação utilizando o operador de igualdade não condiz com o resultado esperado.

Listagem 1.2 Exemplo de erro semântico.

```
SELECT *
FROM Tabela
WHERE atributo = NULL;
```

Violações de regras de boa formação não alteram necessariamente o comportamento de uma consulta SQL. Porém seguir regras de boa formação pode ajudar a manter o código simples e legível, além de manter sua coerência. A listagem 1.3 possui uma consulta que não segue uma boa prática: Uma tabela está sendo referenciada na cláusula FROM mas seus atributos não estão sendo referenciados em nenhuma outra parte da consulta.

Listagem 1.3 Exemplo de violação de regra de boa prática.

```
SELECT TabelaA.atributo
FROM TabelaA, TabelaB
WHERE TabelaA.atributo > 10;
```

1.2 Objetivo

É notado que existe uma necessidade em abordar os erros semânticos na área de comandos SQL de consulta. O objetivo deste trabalho consiste em propor um catálogo de regras que possam auxiliar o desenvolvimento de consultas SQL de forma a evitar erros durante o processo. Além disso, é também um intuito avaliar como os SGBD se comportam quando expostos à comandos que infringem estas regras.

1.3 Delimitação do escopo

É possível detectar falhas de consultas SQL de várias formas diferentes. Porém, este trabalho irá ter o foco somente em um conjunto específico de regras. É possível detectar erros semânticos em função da presença de dados do banco. Por exemplo, uma consulta que acarretará em erro de execução, mas somente em um estado específico do banco. Por causa da abrangência

dessa categoria, estas regras não farão parte do catálogo.

Outro tipo de erro semântico descartado do foco da pesquisa é aquele que depende de um contexto de uma tarefa específica para fazer sentido. Por exemplo, no estudo de [APBL16], um erro semântico é aquele em que a consulta submetida pelo estudante tem um resultado diferente da consulta resposta de uma dada questão. Neste estudo, se uma questão pede três atributos diferentes e só são retornados dois, a consulta é considerada errada, neste contexto. Por isso, esse tipo de erro está fora do escopo deste trabalho.

Violações de regras relacionadas à performance também estão fora do escopo deste trabalho. A preocupação maior é com o sentido das consultas, que precisam ser coerentes. Um comando que foi construído com uma tarefa em mente deve realizar esta tarefa em todos os casos. Um erro semântico está presente em um comando quando o mesmo não realiza o que deveria realizar, pois não foi escrito da forma correta.

1.4 Metodologia

As etapas da pesquisa são as seguintes:

1. **Definição do Problema:** Erros Semânticos precisam ser exemplificados e categorizados.

2. **Revisão da Literatura:** Uma pesquisa foi realizada em busca de trabalhos anteriores que tenham tido algum avanço nesta área. Ferramentas comerciais também servem de base para a construção do catálogo.

3. **Filtro das Regras:** Regras dos trabalhos relacionados que não entram no catálogo final, por causa da restrição de escopo.

4. **Categorização das Regras:** Uma categorização é proposta, a fim de dividir e organizar as regras.

5. **Criação de exemplos:** Exemplos são criados para melhor ilustrar as regras, e também para serem avaliadas na próxima etapa.

6. **Avaliação dos SGBD:** Os exemplos construídos na etapa anterior são executados nos SGBD, para avaliação de seu comportamento.

1.5 Estrutura do documento

O documento está estruturado da seguinte maneira: No capítulo 2, é fornecida uma fundamentação e explicação mais detalhada dos assuntos abordados nos capítulos seguintes. No capítulo 3, um estudo dos trabalhos relacionados da área é apresentado. No capítulo 4, detalhes sobre o catálogo são expostos, como o método para obtenção das regras e a classificação das mesmas, além da avaliação nos SGBD. No capítulo 5, as conclusões deste trabalho; No apêndice A é apresentado os códigos de criação e povoamento do banco de dados utilizados para a avaliação proposta.

CAPÍTULO 2

Fundamentação teórica

Este capítulo apresenta os principais pontos para o entendimento dos próximos capítulos presentes neste trabalho de graduação. Tópicos como Consulta SQL e *Feedback* no contexto de SQL são explanados nas próximas seções.

2.1 Consulta SQL

SQL (Structured Query Language) é uma linguagem de computador criada para manipulação de dados em bancos de dados relacionais [Hal17]. Instruções SQL são executadas por meio de um SGBD (Sistema de Gerenciamento de Bancos de Dados). Os SGBD são responsáveis por servir de interface entre as aplicações e os bancos de dados. São exemplos de SGBD: Oracle; SQL Server; MySQL; PostgreSQL.

Visto que a maior parte deste trabalho se refere à instruções que permitem a consulta dos dados, somente as instruções SELECT serão explanadas. Com estas instruções, é possível obter dados do banco de dados, adicionando filtros, agrupamento, ordenação, junção de tabelas diferentes, entre outros.

Cláusulas SELECT e FROM

A instrução SELECT pode ser dividida em cláusulas, que são obrigatórias ou não. Em sua forma mais simples, uma consulta possui as cláusulas SELECT e FROM. A listagem 2.1) exemplifica este caso.

Listagem 2.1 Exemplo de consulta SQL simples.

```
SELECT * FROM Cliente;
```

A cláusula SELECT indica quais colunas serão retornadas pela consulta. Um asterisco na cláusula SELECT significa que todas as colunas da tabela serão retornadas. A cláusula FROM especifica quais tabelas farão parte da consulta. No exemplo, estamos obtendo todas as colunas

de todas as linhas da tabela Cliente. Todas as linhas serão retornadas pois não foi estabelecida nenhuma condição em relação às linhas.

Existem duas palavras reservadas que são usadas na cláusula SELECT: DISTINCT e TOP. O DISTINCT elimina do resultado da consulta linhas com valores de atributos iguais, enquanto que o TOP é usado para limitar o resultado da consulta usando um número de linhas máximo arbritrário, ou uma porcentagem dos resultados. Na listagem 2.2 a primeira consulta retorna nomes únicos dos clientes, enquanto que a segunda consulta retorna as 5 primeiras linhas da tabela Cliente.

Listagem 2.2 Exemplo de consultas SQL com DISTINCT e TOP.

```
SELECT DISTINCT(Nome) FROM Cliente;
SELECT TOP 5 * FROM Cliente;
```

Cláusula WHERE

A cláusula responsável por filtrar as linhas utilizando condições é a WHERE. A cláusula WHERE possui critérios onde somente as linhas que correspondem àqueles critérios são selecionadas. Estes critérios podem se utilizar de operadores de comparação, como maior que (>), menor que (<), igual (=) e diferente (<>), por exemplo. Operadores lógicos também podem ser usados nestes critérios, para relacionar duas ou mais condições. Os operadores disponíveis são o AND e o OR. A listagem 2.3 apresenta uma consulta com o uso do operador maior que (>), para obter os clientes com idade acima de 20, e outra com o operador AND, para se obter os clientes com idade inferior à 30, e com o nome 'Pedro'.

Listagem 2.3 Exemplo de consultas SQL com a cláusula WHERE.

```
SELECT *
FROM Cliente
WHERE idade > 20;

SELECT *
FROM Cliente
WHERE idade < 30 AND nome = 'Pedro';
```

Ainda na cláusula WHERE, os seguintes operadores podem ser utilizados: BETWEEN, LIKE e IN. O BETWEEN realiza uma comparação inclusiva com os valores entre dois pontos.

O IN serve para comparação entre um conjunto de valores. O LIKE é usado para busca de padrões em strings, o símbolo % significa que zero ou mais caracteres podem ser substituídos pelo mesmo, enquanto que o símbolo _ significa que ele só pode ser substituído por somente um caractere. Na listagem 2.4, três consultas são apresentadas com o uso destes operadores. Na primeira, é retornado os clientes com idade entre 20 e 30. Na segunda, clientes com nome 'Pedro','Artur', ou 'Caio'. E por último, na terceira, clientes cujo nome se iniciam com a letra 'P'.

Listagem 2.4 Exemplo de consultas SQL com os operadores BETWEEN, IN e LIKE.

```
SELECT *
FROM Cliente
WHERE idade BETWEEN 20 AND 30;

SELECT *
FROM Cliente
WHERE nome IN ('Pedro','Artur,'Caio');

SELECT *
FROM Cliente
WHERE nome LIKE ('P%');
```

Cláusula ORDER BY

A cláusula ORDER BY serve para ordenação dos resultados. Pode-se informar as colunas que servirão para a ordenação, bem como o seu sentido: Ascendente ou Descendente. A listagem 2.5 mostra dois exemplos de uso do ORDER BY e das palavras reservadas ASC e DESC, que especifica o sentido da ordenação. O primeiro exemplo lista os clientes pelo nome em ordem alfabética. O segundo exemplo lista os 50 clientes mais velhos.

Listagem 2.5 Exemplo de consultas SQL com a cláusula ORDER BY.

```
SELECT nome
FROM Cliente
ORDER BY nome ASC;

SELECT TOP 50 *
FROM Cliente
```

```
ORDER BY idade DESC;
```

Cláusula GROUP BY

A cláusula GROUP BY é utilizada para agrupamento das linhas. É frequentemente utilizada com *funções de agregação*. Funções de agregação calculam um valor determinado de acordo com um conjunto de linhas, e pode-se definir estes conjuntos com a utilização do GROUP BY. Alguns exemplos de funções de agregação são: MAX, MIN, AVG, COUNT e SUM. Na listagem 2.6, é apresentado um exemplo de uso da função média (AVG), calculando a média de idade dos clientes, e um exemplo com o uso da função de contagem (COUNT) com a cláusula GROUP BY, contando o número de clientes por cidade.

Listagem 2.6 Exemplo de consultas SQL com ORDER BY e funções de agregação.

```
SELECT AVG(idade)
FROM Cliente;

SELECT cidade, COUNT(*)
FROM Cliente
GROUP BY cidade;
```

Cláusula HAVING

A cláusula HAVING para conter condições que se apliquem após a etapa de agrupamento. Como não se pode aplicar condições em funções de agregação no WHERE, estas devem estar na cláusula HAVING. A listagem 2.7 contém um exemplo de uma consulta com HAVING. A consulta retorna a quantidade de clientes por cidade, mas somente as que contenham mais de 10 clientes.

Listagem 2.7 Exemplo de consulta SQL com a cláusula HAVING.

```
SELECT cidade, COUNT(*)
FROM Cliente
GROUP BY cidade;
HAVING COUNT(*) > 10;
```

Cláusula JOIN

Na cláusula JOIN, definimos como será feita a junção com outras tabelas do banco de dados,

baseados em relacionamentos entre estas tabelas, expressados por colunas. Em SQL, pode-se usar os seguintes tipos de JOIN:

- **INNER JOIN**: Retorna linhas que tem referências nas duas tabelas.
- **LEFT JOIN**: Retorna todas as linhas da tabela da esquerda, mesmo que elas não tenham referência na tabela da direita.
- **RIGHT JOIN**: Retorna todas as linhas da tabela da direita, mesmo que elas não tenham referência na tabela da esquerda.
- **OUTER JOIN**: Retorna as linhas que possuem referência em qualquer uma das duas tabelas.
- **CROSS JOIN**: Realiza um produto cartesiano entre as tabelas.
- **NATURAL JOIN**: Realiza uma junção entre as tabelas onde a condição de junção é a igualdade das colunas de mesmo nome.

Para os primeiros quatro tipos de JOIN, é necessário fornecer a *condição de junção*, ou seja, definir quais colunas são usadas para relacionar as tabelas. Para isto, é usada a palavra reservada ON. A listagem 2.8 contém dois exemplos com o uso de JOIN. No primeiro exemplo, é retornada a data dos pedidos realizados pelos clientes de nome 'Paulo'. No segundo, a quantidade de pedidos por cliente, mesmo que o cliente não tenha realizado nenhum pedido.

Listagem 2.8 Exemplo de consultas SQL com o uso de JOIN.

```
SELECT Pedido.data
FROM Pedido
INNER JOIN Cliente
ON Pedido.cliente_id = Cliente.id
WHERE Cliente.nome = 'Paulo';

SELECT Cliente.nome, COUNT(*)
FROM Cliente
LEFT JOIN Pedido
ON Pedido.cliente_id = Cliente.id
GROUP BY Cliente.nome;
```

Subconsultas

Uma subconsulta é simplesmente uma consulta que faz parte da construção de uma consulta principal. Uma subconsulta pode estar presente de várias formas diferentes na consulta principal. Uma forma comum de uso é com a palavra reservada IN, onde o atributo está sendo comparado com uma lista retornada pela subconsulta, por exemplo. Outra forma é com a palavra reservada EXISTS, onde a condição é verdadeira se a subconsulta retorna algo ou não. Uma subconsulta correlacionada é aquela que possui referências à tabelas da consulta principal na sua construção. A listagem 2.9 apresenta exemplos de consultas que possuem subconsulta. No primeiro exemplo, é obtida a lista de clientes que realizaram algum pedido a partir de 01 de janeiro de 2017. O segundo exemplo retorna o nome dos clientes que realizaram algum pedido.

Listagem 2.9 Exemplo de consultas SQL que possuem subconsulta

```
SELECT Cliente.nome
FROM Cliente
WHERE Cliente.id IN (SELECT cliente_id
                FROM Pedido
                WHERE Pedido.data > '2017-01-01');

SELECT Cliente.nome
FROM Cliente
WHERE EXISTS (SELECT *
           FROM Pedido
           WHERE Pedido.cliente_id = Cliente.id);
```

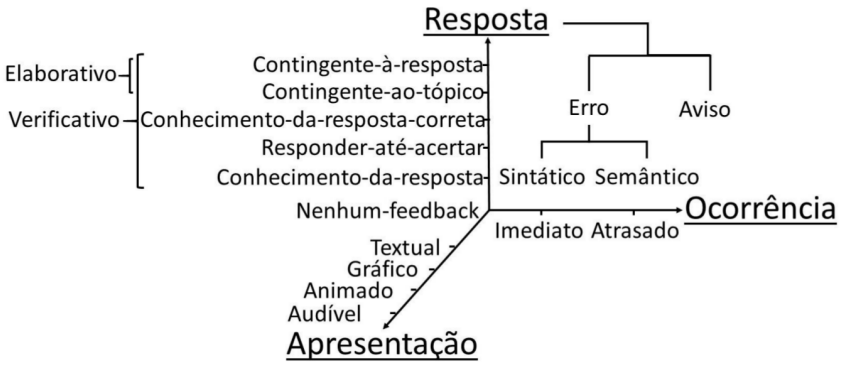
2.2 *Feedback*

Pode-se definir *feedback* de uma maneira geral como o que é gerado a partir de uma determinada ação, e que influencia de alguma forma as próximas ações. No contexto de sistemas de computador, pode-se dizer que *feedback* é uma forma de o usuário perceber como suas ações estão alterando o sistema, ou se ele está fazendo algo de errado, e como ele pode corrigir isto. Um exemplo seria um usuário que, ao tentar realizar o acesso em um sistema, uma mensagem de aviso aparecer para informá-lo que a senha está errada. Um *feedback* ruim seria aquele em que nenhuma mensagem é retornada para o usuário, mesmo com a falha no acesso.

O *feedback* tem uma parte vital no processo de desenvolvimento de sistemas. Um programador teria muitas dificuldades de encontrar o que está errado no seu código se não existisse um processo de *feedback* nas ferramentas de compilação e execução de código. O mesmo se aplica para banco de dados, e a elaboração de instruções SQL para os mesmos.

Pode-se classificar os tipos de *feedback* de várias formas. O trabalho de Robson, Silva e Franco [FSF15] apresenta uma análise e classificação deste processo. Um *feedback* pode ser classificado em questão de Resposta, Apresentação e Ocorrência. Um diagrama desta classificação pode ser visto na Figura 2.2.

Figura 2.1 Classificação de *Feedback*



Um *feedback* de erro semântico deve ser de ocorrência imediata, e pode ser apresentado de qualquer forma. Um *feedback* de erro semântico também pode ser apresentado de qualquer forma, mas pode ser de ocorrência imediata ou atrasada. Ambos os erros são restritivos, ou seja, impedem a boa formação do modelo. Um *feedback* de aviso se trata de violação de regras de boas práticas, mas é sugestivo. Ou seja, não é requesito para a boa formação do modelo.

2.2.1 Erro sintático

Um erro sintático, no contexto de SQL, é aquele que viola as regras da gramática da linguagem. Por exemplo, um erro de digitação em uma palavra reservada - e.g. SELETC, GROPU BY - é considerado um Erro Sintático, pois essas palavras novas não existem no catálogo de palavras reservadas da linguagem SQL. Estes erros impedem o código de ser executado. Os SGBD alertam os usuários de erros de sintaxe justamente pela impossibilidade de execução do código.

2.2.2 Erro semântico

Um erro semântico pode ser definido como aquele que viola regras de boa formação do contexto. O sentido de uma consulta com erro semântico é diferente da intenção que o programador tinha quando elaborou a consulta. Um erro semântico não impede um código de executar. Apesar disso, a sua correção é necessária, pois o comportamento do código não condiz com seu propósito. Uma consulta com um erro semântico, após aplicada a sua correção, terá seu comportamento modificado, agora agindo da maneira correta.

2.2.3 Boas práticas

Uma falha de Boas Práticas é definida como uma violação de padrões de design. Não é necessariamente um erro, mas um padrão geralmente tem um bom motivo para existir. Portanto, sua violação é desencorajada. Um código que não segue uma Boa Prática, quando modificado para incorporar o padrão de Boa Prática, não muda seu comportamento. Mesmo assim, é indicado a sua modificação. Uma consulta pode se tornar mais simples, mais legível ou fazer mais sentido para a tarefa que realiza.

CAPÍTULO 3

Trabalhos relacionados

Este capítulo apresenta de forma resumida quatro trabalhos relacionados.

O primeiro [BG06] se propõe a listar e categorizar características de consultas SQL que podem indicar algum tipo de falha semântica, além de fazer recomendações a fim de aumentar a qualidade do código SQL. A lista contém 43 indícios de erros semânticos e 11 recomendações. O trabalho também propõe uma ferramenta capaz de auxiliar na criação e elaboração de consultas SQL. A ferramenta iria verificar se existe algum indicativo de falha nas consultas inseridas pelo usuário.O segundo [APBL16] analisa um acervo de 160 mil consultas SQL coletadas ao longo de 9 anos, feitas por estudantes de um curso introdutório à bancos de dados. A análise é feita para identificar os principais erros semânticos cometidos por estes estudantes quando estes estão no processo de aprendizado de SQL, além de buscar corrigir a didática dos instrutores para minimizar este problema.O terceiro [Son17] é uma ferramenta comercial que possui um conjunto de regras para análise de qualidade de código. O SonarPLSQL é desenvolvido pela empresa SonarSource contém 150 regras, divididas entre: 30 Bugs, 1 Vulnerabilidade, e 119 *Code Smells*, que seriam possíveis indícios de problemas. Dentre essas regras, também se destaca algumas regras que atendem à padrões de código em geral. Os padrões cobertos pelo SonarPLSQL são: CWE, com 7 regras; MISRA, com 13 regras; CERT, com 6 regras. Por último, o quarto [Ubi17] também é uma ferramenta comercial que analisa código SQL para manter padrões de qualidade, como o SonarPLSQL. O SQL Enlight é desenvolvido pela empresa Ubitsoft. Ele contém 190 regras para análise, divididas em 5 categorias: Regras de Performance; Regras de Manutenção; Regras Explícitas; Regras de Nomeação; Regras de Design. Oferece integração com SQL Server e Visual Studio, funções de Integração Contínua, além de revisão e refatoração de código.

3.1 Brass e Goldberg, 2006

Brass e Goldberg [BG06] investigam erros em consultas SQL, mais especificamente erros semânticos. O trabalho classifica um erro semântico como uma consulta SQL válida, mas que não produz, ou nem sempre produz os resultados planejados, e por isso, é dita como incorreta

para uma dada tarefa. O artigo tem como foco erros semânticos que independem da tarefa, ou seja, quando uma consulta SQL possui indícios de que algo pode estar errado.

O trabalho indica uma lista de condições e características das consultas que as tornavam potencialmente problemáticas. Os problemas são diversos: Simplificação, Desempenho, Inconsistência dos resultados, entre outros. A base desta lista veio de correção de exames que continham consultas SQL, feitos por estudantes ao longo de vários anos. Os casos apresentados foram vistos em prática por estudantes, que depois se tornarão desenvolvedores reais, o que melhora a relevância e importância do artigo.

Foi visto que os SGBD não alertavam aos usuários tais falhas nas consultas, o que chamou a atenção dos autores. Por isso, é proposta uma ferramenta chamada *sqllint* que faria a análise de consultas SQL e mostraria ao usuário se o que foi colocado na entrada tem algum indício de erro semântico. Tal programa não receberia dados, somente a consulta e o esquema como entrada. Um protótipo em PROLOG foi desenvolvido mas não recebe atualizações desde 2005.

Embora o artigo denomine os elementos da lista como Erros Semânticos, pode-se dizer que não são exatamente erros, do ponto de vista gramatical. Alguns contatos, dos autores, da área de desenvolvimento de compiladores, avisaram aos mesmos que estes defeitos não são necessariamente *erros*, pois as consultas ainda são executáveis, e sim *alertas*, e em alguns casos somente *destaca* uma má prática. Mas de qualquer forma, a ferramenta seria muito útil para muitos usuários, pois iria elevar a qualidade de consultas SQL em softwares que acessam bancos de dados.

O artigo é dividido em categorias que representam os motivos pelos quais consultas SQL podem ser suspeitas de terem problemas. A lista completa contém 43 erros e 11 sugestões de estilo, que não são *exatamente* erros mas são indicados para evitar problemas. A Tabela 3.1 ilustra as categorias e a quantidade de regras de cada uma.

Tabela 3.1 Número de erros por categoria

Categoria	Número de elementos
Complicações Desnecessárias	24
Formulações Ineficientes	2
Violações de Padrões Canônicos	10
Muitas Linhas Repetidas	2
Possibilidades de Erros em Tempo de Execução	5
Checagem de Estilo Sugeridas	11

A listagem 3.1 apresenta o esquema utilizado na criação dos exemplos desta seção.

Listagem 3.1 Esquema utilizado nos exemplos da seção 3.1.

```
EMP(EMPNO (PK), ENAME, JOB, SAL, COMM, MGR->EMP, DEPTNO->DEPT)
DEPT(DEPTNO (PK), DNAME, LOC)
```

Complicações Desnecessárias

Um indicativo de que uma consulta provavelmente não está fazendo o que deveria como deveria, é quando ela está em um estado desnecessariamente complicado. Por exemplo, uma consulta Q é escrita e existe uma consulta Q' que realiza o mesmo trabalho de Q, mas é derivada dessa última se certas partes dela forem removidas. Todas as consultas da classe Complicações Desnecessárias podem ser simplificadas sem perda de sentido, e por isso deve-se gerar uma notificação para o usuário.

A listagem 3.2 ilustra o caso Condição Inconsistente. Não importando o estado do banco de dados, o retorno da consulta será vazio. A *constraint* garante que só existirão valores 'M' e 'F' para o atributo 'sex' da tabela 'person'.

Listagem 3.2 Exemplo de consulta SQL com condição inconsistente.

```
ALTER TABLE person
ADD CONSTRAINT sex_constraint CHECK (sex = 'M' OR sex = 'F');

SELECT *
FROM person P
WHERE P.sex = 'W';
```

Formulações Ineficientes

O programador deve ajudar o sistema escrevendo consultas não muito custosas, pois o otimizador de consultas não é eficiente o suficiente para cobrir todos os casos possíveis. Apesar de alguns casos da classe anterior sejam mais eficientes na sua versão correta, existem cenários onde uma consulta *mais longa* é mais eficiente. Esses casos se encontram na classe Formulações Ineficientes

No exemplo apresentado pela listagem 3.3, como a condição E.deptno = D.deptno utiliza atributos do GROUP BY sem função de agregação, ela pode ser colocada no WHERE ou no HAVING. Por motivos de eficiência, a condição deveria estar na cláusula WHERE.

Listagem 3.3 Exemplo de consulta SQL com HAVING Ineficiente.

```
SELECT D.DEPTNO, D.DNAME, COUNT(*)
```

```
FROM EMP E, DEPT D
GROUP BY D.DEPTNO, D.DNAME, E.DEPTNO
HAVING E.DEPTNO = D.DEPTNO
```

Violações de Padrões Canônicos

Quando uma consulta contém elementos que fogem do padrão usado nas consultas, pode ser um indicativo de erro semântico. Tais elementos compõem a classe Violações de Padrões Canônicos.

Quando, na condição de junção de uma junção à esquerda, é usado um atributo da tabela da esquerda, certamente algo está errado. Quando isso acontece, candidatos da tabela à esquerda serão descartados sem nenhuma condição à esquerda. Vale notar que condições à esquerda fazem sentido se colocadas dentro da condição de junção, e não na cláusula WHERE. A listagem 3.4 apresenta um exemplo ilustrando este caso. No exemplo, a restrição D.loc = 'NEW YORK' faz com que os empregados que pertencem à departamentos fora de Nova York não sejam ligados à eles. Porém os departamentos que não são localizados em Nova York aparecem na consulta, com 0 empregados. Este não era o efeito desejado.

Listagem 3.4 Exemplo de consulta SQL com condição na tabela da esquerda de uma junção à esquerda.

```
SELECT D.DNAME, COUNT(E.EMPNO)
FROM DEPT D LEFT OUTER JOIN EMP E
ON D.LOC = 'NEW YORK'
AND D.DEPTNO = E.DEPTNO
```

Muitas Linhas Repetidas

Apesar de não ser necessariamente errado, consultas com muitas repetições de linha também podem indicar que algo não está funcionando como deveria, pode ser inclusive causado por algum outro erro de outra categoria, como a falta de uma condição de junção. Os erros relacionados a repetição estão na categoria Muitas linhas repetidas. A listagem 3.5 contém uma consulta que retorna muitas repetições.

Listagem 3.5 Exemplo de consulta SQL com repetições desnecessárias.

```
SELECT JOB
FROM EMP
```

Provavelmente a intenção do usuário era outra. Podemos corrigir esta consulta de duas

formas, dependendo se o número de repetições é importante ou não. Exemplos de correções estão na listagem 3.6.

Listagem 3.6 Exemplo de consulta SQL com filtragem de repetições desnecessárias.

```
SELECT DISTINCT JOB
FROM EMP

SELECT JOB, COUNT(*)
FROM EMP
GROUP BY JOB
```

Porém, repetições nem sempre são um mau sinal, como por exemplo na consulta presente na listagem 3.7.

Listagem 3.7 Exemplo de consulta SQL com repetições relevantes.

```
SELECT ENAME
FROM EMP
WHERE DEPTNO = 20
```

Embora seja possível existir dois empregados com o mesmo nome, é esperado que isso não aconteça com frequência. De qualquer forma, não se pode descartar as linhas duplicadas nesse caso. Isso acontece pois o atributo ENAME é usado para identificar objetos, mas não é uma chave restritamente única. Os autores definem este tipo de caso como *Soft Keys*. Avisos podem ser gerados para consultas que geram muitas linhas repetidas e não contém Soft Keys na sua composição.

Possibilidades de Erros em Tempo de Execução

Quando uma consulta tem a possibilidade de falhar enquanto executa, ela representa um risco, mesmo que o programador julgue a chance muito baixa. Consultas devem entrar nessa categoria se existe pelo menos um caso ou estado onde um erro de execução ocorra. Tais casos estão contidos na classe Possibilidades de Erros em Tempo de Execução.

No exemplo de Embedded SQL da listagem 3.8, caso existam dois ou mais empregados com o mesmo nome, a consulta falha. Para a consulta se tornar segura, ENAME deveria ser declarada como chave de EMP, ou com uma restrição Unique.

Listagem 3.8 Exemplo do caso SELECT INTO pode retornar mais de uma tupla.

```
SELECT JOB, SAL
INTO :X, :Y
FROM EMP
WHERE ENAME = :N
```

Checagem de Estilo Sugeridas

Por último, são listados recomendações de estilo no momento de se escrever consultas SQL, em uma classe denominada Checagem de Estilo Sugeridas. O conteúdo desta classe pode ajudar a legibilidade, estabelecer padrões que aumentam a qualidade de consultas, e evitar possíveis problemas de interpretação em relação ao ambiente de execução das consultas.

Por exemplo, uma subconsulta do tipo IN que é correlacionada provavelmente deveria ser substituída por uma subconsulta do tipo EXISTS, por questões semânticas.

3.2 Ahadi, Prior, Behbood e Raymond, 2015

Ahadi, Prior, Behbood e Raymond [APBL16] focam seu trabalho em analisar erros semânticos cometidos por estudantes quando estes são submetidos à questões de exames que envolvem consultas SQL do tipo SELECT. O artigo contém 9 anos de trabalho, coletando 160 mil consultas SQL de 2300 estudantes. Cada consulta representa uma tentativa de um estudante em uma dada questão. O objetivo do trabalho é mapear estas falhas em categorias e explicar o porquê os estudantes cometerem esses erros.

Os dados utilizados no artigo foram coletados de uma ferramenta de avaliação online chamada *AsseSQL*, desenvolvida para avaliação de estudantes em uma disciplina introdutória de banco de dados. As consultas foram capturadas durante testes supervisionados de 50 minutos, que continham sete questões de SQL. Cada questão testa o estudante a elaborar uma consulta que cobre um conceito específico da linguagem SQL. A Tabela 3.2 mostra os conceitos avaliados e suas respectivas quantidades de consultas.

Uma vez coletadas as consultas SQL, elas são re-executadas no PostgreSQL e a saída de cada consulta é obtida. A classificação da resposta é dividida em: *Correta*, se o conjunto resposta é exatamente igual à resposta da consulta solução associada à questão; *Sintaticamente errada* se alguma mensagem de erro foi retornada pelo SGBD; Ou *Semanticamente errada* se o conjunto resposta for vazio ou diferente do conjunto gerado pela resposta da consulta solução.

Tabela 3.2 Conceitos SQL e o número de consultas associado

Conceito	Número de Consultas
GROUP BY com HAVING	32k (20%)
Auto-junção	27k (17%)
GROUP BY	25k (15%)
Natural Join	24k (15%)
Subconsulta Simples	19k (12%)
Simples com uma tabela	18k (11%)
Subconsulta Correlacionada	16k (10%)

Análise dos dados obtidos

Em média, o número de erros sintáticos é maior que o número de erros semânticos, porém isto não significa que os erros semânticos têm menos importância que os sintáticos. O estudo sugere que erros sintáticos são cometidos por falta de prática e cuidado dos estudantes. Aproximadamente 69% dos erros sintáticos são causados por erros de digitação em nomes de atributos, tabelas, ou sintaxe do comando SELECT. Se esse conjunto for ignorado, os erros semânticos ocorrem mais que os erros sintáticos. Outro fator importante para a importância elevada dos erros semânticos é que uma consulta com erro sintático pode já conter um erro semântico. Ou seja, mesmo com a sintaxe corrigida ela ainda não está correta.

A categorização dos erros é realizada por conceitos, mas também pode se mapear por localização do erro na consulta, ou seja, pela cláusula. A Tabela 3.2 mostra a divisão por cláusula, os erros que ocorrem nelas, e quais conceitos estão envolvidos.

Ogden [OKS86] categoriza o conhecimento de elaborar consultas em: Conhecimento dos dados; Conhecimento da estrutura do banco de dados; Conhecimento da linguagem de consulta. A falta de conhecimento das duas primeiras categorias geralmente se leva aos erros sintáticos, enquanto a ultima gera erros semânticos. O artigo de Ahadi [APBL16] sugere que os estudantes falham semanticamente pois não conseguem escolher a melhor direção para elaborar uma consulta para resolver o problema. Geralmente os estudantes falham em escolher a técnica mais apropriada - junção, subconsulta, auto-junção - e por isso acabam não conseguindo escrever uma consulta corretamente.

Tabela 3.3 Categorização dos erros semânticos por cláusula

Cláusula	Erro(s)	Conceitos
WHERE (46%)	Condição omitida ou incorreta	Simples, Auto-junção, Subconsulta correlacionada, Junção
FROM (26%)	Auto-junção não realizada	Auto-junção
HAVING (13%)	GROUP BY ou HAVING omitidos, uso de atributo incorreto	HAVING
ORDER BY (5%)	Order By omitido, uso de atributo incorreto ou omitida	Simples, GROUP BY
SELECT (5%)	Atributo omitido ou além do necessário	Simples, GROUP BY
GROUP BY (5%)	GROUP BY omitido, uso da atributo incorreto	GROUP BY e GROUP BY com HAVING

3.3 SonarPLSQL, 2017

O SonarPLSQL [Son17] é um analisador de código estático para PL/SQL, desenvolvido pela empresa SonarSource. A ferramenta conta com 150 regras que indicam vulnerabilidades, erros ou *code smells*. O SonarPLSQL tem integração com o Ambiente de Desenvolvimento Integrado Eclipse, além de ser oferecido seu acesso online, pelo serviço SonarCloud e integração com fluxos de análise automática de código, com o SonarQube.

Ele também fornece a funcionalidade de criação customizada de regras. As 150 regras estão divididas entre: Bugs, com 22 regras; Vulnerabilidade, com 1 regra; e *Code Smells*, com 119 regras. Exemplos de cada uma das divisões se encontram nas listagens 3.9, 3.10 e 3.11.

Bugs

Declarações Delete e Update sem a cláusula WHERE representam um descuido do programador, e muito provavelmente não foi intencional. A listagem 3.9 contém um código que não segue a regra e outro que segue a regra.

Listagem 3.9 Exemplos da regra: Declarações "DELETE"e "UPDATE"devem conter a cláusula "WHERE".

```
--Noncompliant Code Example

DECLARE
```

```
  maxAge PLS_INTEGER := 60;
BEGIN
  UPDATE employee SET status = 'retired';
  -- Noncompliant - the WHERE was forgotten
END;
/

--Compliant Solution

DECLARE
  maxAge PLS_INTEGER := 60;
BEGIN
  UPDATE employee SET status = 'retired' WHERE age > maxAge;
END;
/
```

Vulnerabilidades

Alguns pacotes do Oracle contém funções muito poderosas que pertencem a "SYS", e elas podem ser usadas para atividades mal-intencionadas. A maioria dos programas não precisam dessas funções, por isso alertas são gerados sempre que elas são usadas. A listagem 3.10 demonstra como esta regra pode ser violada.

Listagem 3.10 Exemplo da regra: Funções delicadas de "SYS"não devem ser usadas.

```
DECLARE
  c INTEGER;
  sqltext VARCHAR2(100) := 'ALTER USER system IDENTIFIED BY
     hacker';
BEGIN
  c := SYS.DBMS_SYS_SQL.OPEN_CURSOR();

   -- Will change 'system' user's password to 'hacker'
  SYS.DBMS_SYS_SQL.PARSE_AS_USER(c, sqltext, DBMS_SQL.NATIVE,
     UID);

  SYS.DBMS_SYS_SQL.CLOSE_CURSOR(c);
```

```
END;
/
```

Code Smells

Quando múltiplas tabelas estão envolvidas em uma consulta, as tabelas devem ter alias por questão de legibilidade. A listagem 3.11 apresenta dois códigos. Um que viola esta regra, e outro que a satisfaz.

Listagem 3.11 Exemplos da regra: Tabelas devem conter alias.

```
--Noncompliant Code Example
BEGIN
  SELECT
    name,
    firstname,
    location
  INTO employeesArray
  FROM employee -- Noncompliant - should be aliased
  INNER JOIN department -- Noncompliant - should be aliased
  ON employee.DepartmentID = department.ID;
END;
/

-- Compliant Solution

BEGIN
  SELECT
    empl.name,
    empl.firstname,
    dpt.location
  INTO employeesArray
  FROM employee empl
  INNER JOIN department dpt
  ON empl.DepartmentID = dpt.ID;
END;
/
```

Regras de padrões de código

Algumas regras do SonarPLSQL também buscam atender a requesitos de padrões de código, como CWE, MISRA e CERT. A cobertura se dá pela seguinte forma: CWE [MIT17], com 7 regras; MISRA [MIS17], com 13 regras; CERT [SEI17], com 6 regras. Essas regras entram na contagem das 150 regras do programa e também fazem parte da classificação anterior.

O CWE - Common Weakness Enumeration - é uma lista formal de tipos de fraquezas de software para: Servir como linguagem comum para descrever pontos fracos nos softwares, em relação à arquitetura, design ou código; Servir como um medidor para ferramentas de segurança que focam em evitar estes pontos fracos; Prover uma referência padrão para identificação, tratamento e prevenção dessas fraquezas.

O padrão MISRA C, desenvolvido pela MISRA - Motor Industry Software Reliability Association - é um conjunto de diretrizes para desenvolvimento de software. Seu objetivo é facilitar a codificação em sistemas embarcados, no que diz respeito à: Segurança; Proteção, Portabilidade e Confiabilidade. O SonarPLSQL adaptou algumas dessas regras para seu produto, já que o MISRA tem como foco sistemas desenvolvidos em C.

O CERT se refere ao CERT/CC - Computer Emergency Response Team - do centro de pesquisa e desenvolvimento SEI - Software Engineering Institute. Eles elaboraram um conjunto de regras e diretrizes para as linguagens C, C++, Java e Perl. Este padrão é desenvolvido em comunidade, por meio de uma Wiki, e cerca de 1700 contribuidores e revisores participaram do projeto.

3.4 SQL Enlight, 2017

O SQL Enlight [Ubi17] é uma extensão do Visual Studio e SQL Server Management Studio, usada para análise de código. A sua função é identificar possíveis falhas e sugerir otimizações em trechos de código Transact-SQL e bancos de dados SQL Server. A ferramenta ainda oferece revisão, refatoração e formatação de código e suporte à Integração Contínua.

Esta ferramenta contém 190 regras, divididas nas seguintes categorias : Regras de Design; Regras Explícitas; Regras de Nomeação; Regras de Performance; Regras de Manutenção. Uma regra pode estar classificada em duas categorias ao mesmo tempo. A Tabela 3.4 mostra a quantidade de regras separada em categorias.

Regras de Design

Esta seção contém regras de análise relacionadas a design.
Exemplo: Quando operações de inserção forem executadas, elas devem conter a lista de atri-

Tabela 3.4 Número de regras por categoria

Categoria	Número de elementos
Regras de Design	103
Regras Explícitas	12
Regras de Nomeação	31
Regras de Performance	41
Regras de Manutenção	18

butos, para facilitar a legibilidade e a manutenabilidade do código. A listagem 3.12 apresenta exemplos de violação e cumprimento da regra.

Listagem 3.12 Exemplos da regra: Operações de Insert devem conter lista de atributos.

```
CREATE PROCEDURE HumanResources.uspGetEmployees
   @LastName nvarchar(50),
   @FirstName nvarchar(50),
   @JobTitle nvarchar(50),
   @Department nvarchar(50)
AS

-- Target columns are not provided
INSERT INTO HumanResources.vEmployeeDepartment
VALUES (@FirstName,@LastName,@JobTitle,@Department)

-- Target columns are explicitly provided
INSERT INTO HumanResources.vEmployeeDepartment(FirstName,
   LastName, JobTitle, Department)
VALUES (@FirstName,@LastName,@JobTitle,@Department)
```

Regras Explícitas

Esta seção contém regras de análise que devem ser executadas explicitamente.
Exemplo: Um padrão de desenvolvimento de código é a boa documentação dos procedimentos. Esta regra de exemplo checa se, antes de criação de objetos do banco de dados, existe uma documentação que segue o molde fornecido. A listagem 3.13 apresenta um exemplo no qual a regra está sendo cumprida.

Listagem 3.13 Exemplo da regra: Adicionar documentação na criação de novos objetos.

```
--============================================
-- Author: Author's name
-- Create Date: 2010-05-01
-- Description: Example Stored Procedure
-- Update Date: 2010-05-19
-- ============================================
CREATE PROCEDURE MyProcedureName
      @p1 int = 0,
      @p2 int = 0
AS
[...]
```

Regras de Nomeação

Esta seção contém regras de análise relacionadas com nomeação de objetos.
Exemplo: Na criação de funções, evitar usar o prefixo 'fn_'. Apesar de permitido, é preferível que não sejam usados para não entrar em conflito com objetos da Microsoft. A listagem 3.14 apresenta um exemplo que viola esta regra.

Listagem 3.14 Exemplo de violação da regra: Evitar prefixo 'fn_'.

```
CREATE FUNCTION dbo.fn_myfunction
(
    @value AS int
)
RETURNS int
WITH EXECUTE AS CALLER
AS
BEGIN
   SET @value=@value + 1
END;
```

Regras de Performance

Esta seção contém regras de análise relacionadas a performance.
Exemplo: Atributos utilizados pelo predicado IN devem ser indexados, a fim de evitar um *Table Scan*. A listagem 3.15 apresenta exemplos de variáveis que devem ser indexadas para o

cumprimento desta regra.

Listagem 3.15 Exemplos da regra: Utilizar atributos indexados em predicados IN.

```
--Category column have to be indexed in order to avoid a table
    scan during query processing.
SELECT [Comment]
FROM [Sales].[SpecialOffer] a
WHERE [Category] IN (Description)

-- Category column have to be indexed in order to avoid a
   table scan during query processing.
SELECT [Comment]
FROM [Sales].[SpecialOffer]
WHERE [SpecialOfferID] IN (1, 2, 3)
AND [Category] IN ('Category1', 'Category2','Category3' )
```

Regras de Manutenção

Esta seção contém regras de análise relacionadas à manutenção e administração do SQL Server. Exemplo: Cópias de segurança dos bancos devem ser atualizadas com frequência. Uma regra do SQL Enlight checa se existe cópias de segurança recentes disponíveis para uso, se não, um alerta é gerado.

CAPÍTULO 4

Catálogo de regras de boa formação

Neste capítulo, é apresentado o método usado para elaboração do catálogo de erros semânticos e boas práticas, bem como a avaliação dos SGBD quando submetidos à consultas que contém alguma característica apresentada no catálogo.

4.1 Método de obtenção de erros semânticos e boas práticas

Inicialmente, uma pesquisa foi realizada na literatura em busca de artigos que abordassem o tema Erros Semânticos em SQL. Ferramentas comerciais e sites da internet com tutoriais e guias também fizeram parte da pesquisa a fim de enriquecer as opções que se teria para a elaboração do catálogo. Após as fontes serem encontradas, foi realizado um trabalho de filtragem dos erros, para que se houvesse um escopo melhor definido. Alguns critérios foram utilizados na exclusão das regras encontradas nas fontes. Foram excluídas do catálogo regras que:

- Dependem do estado do banco de dados.
- Dependem de informação externa - Em Goldberg[BG06] existe a definição de *Soft Keys*, que são atributos usados para identificação de objetos, mas não são necessariamente únicos.
- Só são consideradas errôneas dentro de um contexto específico, como por exemplo uma tarefa arbitrária.
- São executadas dentro de um programa hospedeiro - *Embedded SQL*.
- Podem ser interrompidas em tempo de execução.
- Carecem de legibilidade mas não afetam a semântica da consulta.
- Visam melhorar a eficiência da consulta.

Após a filtragem, as regras restantes são classificadas em Erros Semânticos e Boas Práticas. Apesar dos trabalhos alegarem trabalhar com *Erros Semânticos*, sentiu-se a necessidade de reclassificar as regras, para as mesmas se encaixarem no nosso escopo. Para esta reclassificação, foram utilizadas as definições de Erro Sintático, Erro Semântico e Boas Práticas, apresentadas no Capítulo 2.

Além desta primeira classificação, os elementos do catálogo também são classificados em: Consulta ou Esquema. Um erro pertence à classe Consulta se somente a consulta é necessária para determinar se existe erro ou não. Já um erro da classe Esquema é necessário, além da consulta, ter acesso ao esquema no qual a consulta está sendo aplicada. A Tabela 4.1 ilustra essa classificação de forma quantitativa, e a Tabela 4.2 qualitativamente.

Tabela 4.1 Quantidade de regras e suas classificações

Erros Semânticos		**Boas Práticas**	
Consulta	**Esquema**	**Consulta**	**Esquema**
6 regras	2 regras	20 regras	4 regras

4.2 Apresentação do catálogo

Esta seção apresenta o catálogo em sua versão final, com 8 erros semânticos e 24 recomendações de boas práticas, bem como o exemplo que ilustra cada um destes elementos. É também apresentado o esquema do qual os exemplos fazem parte.

4.2.1 Esquemas utilizados nos exemplos

Para a elaboração de exemplos dos elementos do catálogo, é necessário definir um esquema no qual os exemplos atuarão. É dado o esquema de 4 tabelas. 2 tabelas serão utilizadas na maioria dos exemplos, as tabelas **Empregado** e **Departamento**. As tabelas **Pessoa** e **Endereco** são utilizadas somente no exemplo 8. Um diagrama com os esquemas é apresentado na Figura 4.1.

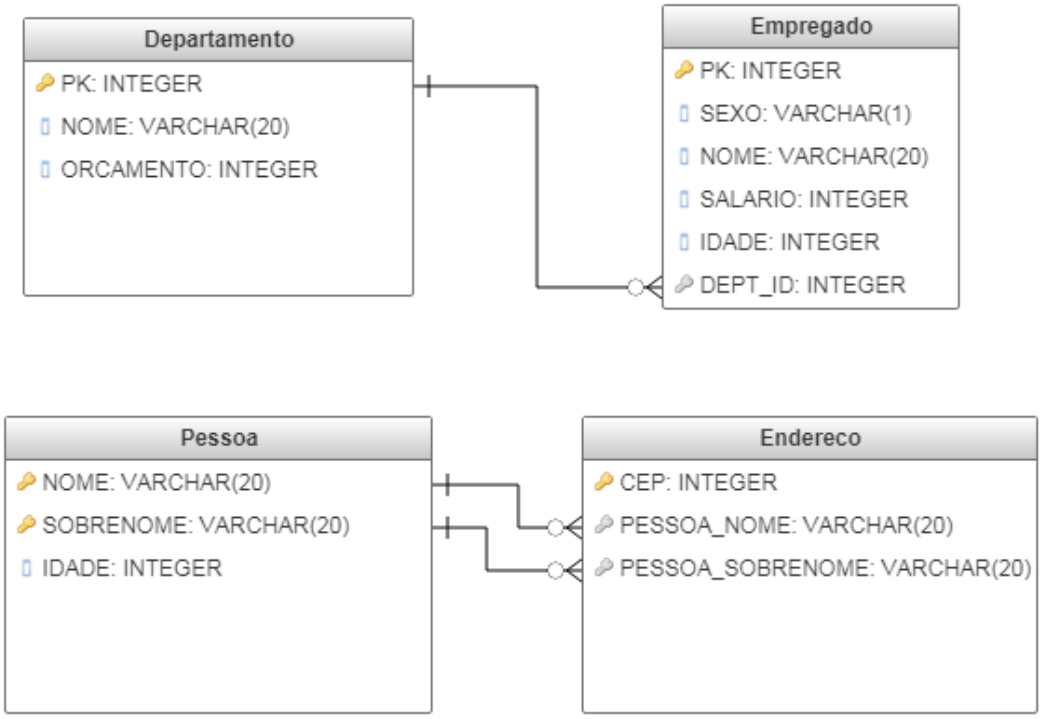
4.2.2 Erros Semânticos - Consulta

Esta subseção apresenta os seis erros semânticos do catálogo que podem ser identificados quando se utiliza somente a consulta.

Tabela 4.2 Listagem das regras

Erros Semânticos - Consulta
1. Condição Inconsistente
2. Subconsulta EXISTS não corelacionada
3. SELECT de subconsulta não utiliza variáveis da subconsulta
4. Condição envolve tabela da esquerda em uma junção à esquerda
5. HAVING sem GROUP BY
6. Comparação com NULL utilizando igualdade
Erros Semânticos - Esquema
7. Condição Inconsistente
8. DISTINCT desnecessário nas funções MIN e MAX
Boas Práticas - Consulta
1. Atributo constante é retornada na consulta
2. Atributos duplicados no SELECT
3. Tabelas não utilizadas no FROM
4. Junção desnecessária
5. LIKE sem caracteres especiais
6. Subconsulta EXISTS deve retornar algo simples
7. Subconsulta IN/EXISTS substituída por comparação simples
8. DISTINCT desnecessário nas funções MIN e MAX
9. Argumento desnecessário no COUNT
10. GROUP BY desnecessário em subconsultas EXISTS
11. GROUP BY pode ser substituído por um DISTINCT
12. UNION pode ser substituído por OR
13. Condição de subconsulta pode ser movida para consulta Principal
14. Uso de DISTINCT em AVG e SUM
15. Uso de caracteres especiais sem uso do LIKE
16. OUTER JOIN pode ser substituído por INNER JOIN
17. Sempre utilizar aliases
18. Consultas IN correlacionadas devem se tornar EXISTS
19. SELECT DISTINCT desnecessário em subconsultas IN
20. Evitar consultas utilizando NATURAL JOIN
Boas Práticas - Esquema
21. Cláusula DISTINCT desnecessária
22. Atributo em GROUP BY desnecessário
23. Termo desnecessário do ORDER BY
24. Comparação de domínios diferentes

Figura 4.1 Diagrama das tabelas utilizadas nos exemplos.



1. Condição Inconsistente:

Uma consulta contém uma condição inconsistente quando o resultado dela é vazio para qualquer estado do banco. Isto significa que algo está errado com a cláusula de condição. Esta regra está dividida em duas, pois podemos ou não utilizar o esquema. A Listagem 4.1 possui um exemplo de uma condição inconsistente, pois o resultado da consulta é sempre vazio.

Listagem 4.1 Exemplo de consulta SQL que viola a regra 1 de erro semântico.

```
SELECT *
FROM Empregado E
WHERE E.sexo = 'M' AND E.sexo = 'F';
```

2. Subconsulta EXISTS não corelacionada:

Se uma consulta EXISTS não está relacionada com a consulta principal, há algo de errado, pois ela será ou verdadeira ou falsa para todos os casos. A Listagem 4.2 apresenta um exemplo onde a subconsulta EXISTS não é correlacionada.

Listagem 4.2 Exemplo de consulta SQL que viola a regra 2 de erro semântico.

```
SELECT D.id
FROM Departamento D
WHERE EXISTS (SELECT *
   FROM Empregado E
   WHERE E.idade < 30);
```

3. SELECT de subconsulta não utiliza variáveis da subconsulta:
Uma subconsulta que seleciona atributos da consulta principal não faz sentido. O retorno da subconsulta deve envolver as tabelas da cláusula FROM da própria subconsulta. A Listagem 4.3 apresenta um exemplo onde a subconsulta retorna um atributo da consulta principal.

Listagem 4.3 Exemplo de consulta SQL que viola a regra 3 de erro semântico.

```
SELECT E.nome
FROM Empregado E
WHERE E.id IN (SELECT E.idade
   FROM Departamento D
   WHERE D.nome = 'Contabilidade');
```

4. Condição envolve tabela da esquerda em uma junção à esquerda:
Uma condição de junção que utiliza a tabela da esquerda, quando é uma junção à esquerda, indica um erro grave. Candidatos a junção da tabela da direita serão descartados por uma condição não imposta à eles, e sim à outra tabela. A condição em questão deveria ser movida para a cláusula WHERE. A Listagem 4.4 apresenta um exemplo onde existe uma condição para a tabela à esquerda da junção (E.idade > 30), o que descartará elementos da tabela à direita (Departamento). Uma possível solução também é apresentada, onde a condição é movida para a cláusula WHERE.

Listagem 4.4 Exemplo de consulta SQL que viola a regra 4 de erro semântico.

```
SELECT E.id
FROM Empregado E
LEFT JOIN Departamento D
ON E.dep_id = D.id
AND E.idade > 30;
```

```
-- Consulta Correta:
SELECT E.id
FROM Empregado E
LEFT JOIN Departamento D
ON E.dep_id = D.id
WHERE E.idade > 30;
```

5. HAVING sem GROUP BY:

Apesar de ser estranhamente válido, pode existir um HAVING sem GROUP BY. Tal caso é considerado um erro semântico pois a consulta só pode ter um ou nenhum resultado, já que a aplicação da condição é sobre o conjunto após o agrupamento. Como não existe cláusula GROUP BY, todo o conjunto é agrupado para uma linha, que não faz sentido. A Listagem 4.5 apresenta um exemplo onde um HAVING é utilizado sem a presença do GROUP BY.

Listagem 4.5 Exemplo de consulta SQL que viola a regra 5 de erro semântico.

```
SELECT MIN(E.idade)
FROM Empregado E
HAVING MIN(E.idade) > 20;
```

6. Comparação com NULL utilizando igualdade:

Comparações com o valor NULL devem ser feitas usando as palavras reservadas IS (NULL) ou IS NOT (NULL). A condição não se comporta como o esperado se o operador de igualdade for utilizado. A Listagem 4.6 apresenta um exemplo onde um atributo está sendo comparado com o valor NULL utilizando o operador de igualdade (E.dep_id = NULL).

Listagem 4.6 Exemplo de consulta SQL que viola a regra 6 de erro semântico.

```
SELECT *
FROM Empregado E
WHERE E.dep_id = NULL;

-- Consulta Correta:
SELECT *
FROM Empregado E
```

```
WHERE E.dep_id IS NULL;
```

4.2.3 Erros Semânticos - Esquema

Esta subseção apresenta os dois erros semânticos do catálogo que podem ser identificados quando se utiliza o esquema no qual a consulta está aplicada.

7. Condição Inconsistente:

Uma consulta com condição inconsistente sempre retorna um resultado inválido, independente do estado do banco. Esta regra é análoga a regra 1, porém agora se tem acesso ao esquema. No exemplo, o atributo CONST possui um CONSTRAINT de valores, e a busca está sendo realizada por valores que, por causa do CONSTRAINT, são impossíveis de existir. A Listagem 4.7 apresenta um exemplo onde a consulta sempre irá retornar vazio, e isto pode ser garantido por causa da CONSTRAINT na tabela acessada.

Listagem 4.7 Exemplo de consulta SQL que viola a regra 7 de erro semântico.

```
-- CHECK CONSTRAINT(sexo = 'M' or sexo = 'F');

SELECT *
FROM Empregado E
WHERE E.sexo = 'W';
```

8. Ausência da condição de junção:

Quando deseja-se realizar uma junção entre duas tabelas relacionadas, a condição de junção deve ser empregada corretamente. Mesmo o uso de um produto cartesiano entre duas tabelas relacionadas pode indicar este erro, pois a condição de junção deve ser aplicada corretamente sobre as chaves de ambas as tabelas. No exemplo presente na Listagem 4.8, a tabela C tem duas chaves primarias e a tabela D tem duas chaves estrangeiras para C, porém a condição de junção só envolve uma das chaves.

Listagem 4.8 Exemplo de consulta SQL que viola a regra 8 de erro semântico.

```
SELECT *
FROM Pessoa P
LEFT JOIN Endereco E
```

```
ON P.nome = E.pessoa_nome;

-- Consulta Correta:
SELECT *
FROM Pessoa P
LEFT JOIN Endereco E
ON P.nome = E.pessoa_nome
AND P.sobrenome = E.pessoa_sobrenome;
```

4.2.4 Boas Práticas - Consulta

Esta subseção apresenta as vinte boas práticas do catálogo que podem ser indicadas quando se utiliza somente a consulta.

1. Atributo constante é retornada na consulta:

Um atributo que é aplicado à uma condição constante não é necessária, pois o seu valor será conhecido antes mesmo da consulta. A Listagem 4.9 apresenta um exemplo onde a consulta retorna um atributo constante, já definido pela condição presente no WHERE.

Listagem 4.9 Exemplo de consulta SQL que viola a regra 1 de boas práticas.

```
SELECT E.nome, E.idade
FROM Empregado E
WHERE E.idade = 25;

-- Consulta Correta:
SELECT E.nome
FROM Empregado E
WHERE E.idade = 25;
```

2. Atributos duplicados no SELECT:

Não faz sentido existir atributos duplicados no SELECT de uma consulta: Os valores serão repetidos. A Listagem 4.10 apresenta um exemplo onde a consulta retorna atributos repetidos.

Listagem 4.10 Exemplo de consulta SQL que viola a regra 2 de boas práticas.

```
SELECT E.nome, E.nome
FROM Empregado E
WHERE E.idade >= 30;

-- Consulta Correta:
SELECT E.nome
FROM Empregado E
WHERE E.idade >= 30;
```

3. Tabelas não utilizadas no FROM:

Uma tabela declarada na cláusula FROM e não é usada em nenhum outro lugar da consulta é desnecessário e deve ser removido. Este exemplo se assemelha com o Erro Semântico 8, mas é causada por um motivo semântico diferente. É um caso de *esquecimento* do programador. Portanto é considerado como boa prática existir na cláusula FROM somente tabelas utilizadas na consulta. A Listagem 4.11 apresenta um exemplo onde a consulta contém uma tabela não referenciada na cláusula FROM.

Listagem 4.11 Exemplo de consulta SQL que viola a regra 3 de boas práticas.

```
SELECT E.nome
FROM Empregado E, Departamento D
WHERE E.salario >= 5000;

-- Consulta Correta:
SELECT E.nome
FROM Empregado E
WHERE E.salario >= 5000;
```

4. Junção desnecessária:

Uma consulta que possui uma junção entre duas tabelas mas só acessa atributos de uma tabela não é recomendada. Neste caso, o JOIN não está sendo utilizado, portanto deve ser removido. A Listagem 4.12 apresenta um exemplo onde a consulta realiza um JOIN mas não acessa nenhum atributo da tabela utilizada no JOIN.

Listagem 4.12 Exemplo de consulta SQL que viola a regra 4 de boas práticas.

```
SELECT E.nome
FROM Empregado E
LEFT JOIN Departamento D ON E.dep_id = D.id
WHERE E.idade > 30;

-- Consulta Correta:
SELECT E.nome
FROM Empregado E
WHERE E.idade > 30;
```

5. LIKE sem caracteres especiais:

O LIKE se difere do operador de igualdade pois ele pode ser utilizado com caracteres especiais, como '_' e '%'. Portanto, não é indicado utilizá-lo para comparar strings sem estes caracteres. A Listagem 4.13 apresenta um exemplo onde a consulta realiza uma comparação usando LIKE mas sem o uso de caracteres especiais (E.nome LIKE 'Pedro').

Listagem 4.13 Exemplo de consulta SQL que viola a regra 5 de boas práticas.

```
SELECT E.nome
FROM Empregado E
WHERE E.nome LIKE 'Pedro';

-- Consulta Correta:
SELECT E.nome
FROM Empregado E
WHERE E.nome = 'Pedro';
```

6. Subconsulta EXISTS deve retornar algo simples:

Uma subconsulta do tipo EXISTS só é usada para saber se é retornado algo ou não, não importando o que esse algo seja. Por isso, é recomendado se retornar uma constante, ou um * na subconsulta. A Listagem 4.14 apresenta um exemplo onde a subconsulta que utiliza EXISTS está retornando atributos desnecessários (D.nome, D.id, D.orcamento).

Listagem 4.14 Exemplo de consulta SQL que viola a regra 6 de boas práticas.

```
SELECT E.nome
FROM Empregado E
WHERE EXISTS (SELECT D.nome, D.id, D.orcamento
   FROM Departamento D
   WHERE D.id = E.dep_id);

-- Consulta Correta:
SELECT E.nome
FROM Empregado E
WHERE EXISTS (SELECT 1
   FROM Departamento D
   WHERE D.id = E.dep_id);
```

7. Subconsulta IN/EXISTS substituída por comparação simples:
Se todas as tabelas de uma subconsulta estão presentes na consulta principal, e todas as referências desta subconsulta estão somente em suas tabelas internas, toda a subconsulta pode ser substituída por uma comparação simples na consulta principal. A Listagem 4.15 demonstra um caso que viola esta regra.

Listagem 4.15 Exemplo de consulta SQL que viola a regra 7 de boas práticas.

```
SELECT E.nome
FROM Empregado E
WHERE E.id IN (SELECT E2.id
   FROM Empregado E2
   WHERE E2.nome = 'Lucas');

-- Consulta Correta:
SELECT E.nome
FROM Empregado E
WHERE E.nome = 'Lucas');
```

8. DISTINCT desnecessário nas funções MIN e MAX:
Nas funções de agregação MIN e MAX, o DISTINCT é desnecessário. A Listagem 4.16 apre-

senta um exemplo onde uma consulta que utiliza a função de agregação MAX junto com o DISTINCT.

Listagem 4.16 Exemplo de consulta SQL que viola a regra 8 de boas práticas.

```
SELECT MAX(DISTINCT E.salario)
FROM Empregado E;

-- Consulta Correta:
SELECT MAX(E.salario)
FROM Empregado E;
```

9. Argumento desnecessário no COUNT:
Quando não existe DISTINCT e o atributo argumento do COUNT não pode ser nulo, é preferível a versão sem argumento. A Listagem 4.17 apresenta um exemplo onde uma consulta que utiliza o COUNT em um atributo, quando o COUNT(*) serviria para o mesmo propósito.

Listagem 4.17 Exemplo de consulta SQL que viola a regra 9 de boas práticas.

```
SELECT COUNT(E.id)
FROM Empregado E
WHERE E.salario < 5000;

-- Consulta Correta:
SELECT COUNT(*)
FROM Empregado E
WHERE E.salario < 5000;
```

10. GROUP BY desnecessário em subconsultas EXISTS:
A presença de GROUP BY em subconsultas EXISTS não altera o resultado de forma significativa, portanto deve ser removido por ser desnecessário. A Listagem 4.18 apresenta um exemplo onde uma subconsulta EXISTS possui uma cláusula GROUP BY, que não afeta o resultado da avaliação.

Listagem 4.18 Exemplo de consulta SQL que viola a regra 10 de boas práticas.

```
SELECT E.nome
```

```
FROM Empregado E
WHERE EXISTS (SELECT *
   FROM Departamento D
   WHERE D.id = E.dep_id
   GROUP BY D.nome);

-- Consulta Correta:
SELECT E.nome
FROM Empregado E
WHERE EXISTS (SELECT *
   FROM Departamento D
   WHERE D.id = E.dep_id);
```

11. GROUP BY pode ser substituído por um DISTINCT:

Se os atributos do SELECT forem exatamente os mesmos do GROUP BY, e funções de agregação não são utilizadas, o GROUP BY pode ser substituído pelo DISTINCT, pois é mais direto e tem o mesmo efeito. A Listagem 4.19 apresenta um exemplo onde uma subconsulta possui três atributos na cláusula SELECT e os três atributos, na mesma ordem, na cláusula GROUP BY.

Listagem 4.19 Exemplo de consulta SQL que viola a regra 11 de boas práticas.

```
SELECT E.sexo, E.idade
FROM Empregado E
GROUP BY E.sexo, E.idade;

-- Consulta Correta:
SELECT DISTINCT E.sexo, E.idade
FROM Empregado E;
```

12. UNION pode ser substituído por OR:

Duas consultas unidas por UNION ALL cujas condições são exclusivas, e as cláusulas SELECT e FROM são iguais, podem ser simplificadas. Neste caso, o UNION ALL pode ser reduzido à união OR das condições de ambas as consultas. UNION também pode ser usado, dado que a restrição de valores nulos seja conferida previamente. A Listagem 4.20 apresenta um exemplo

onde duas consultas estão ligadas pela palavra UNION, porém as suas condições não possuem intercessão. Portanto, infringe a regra.

Listagem 4.20 Exemplo de consulta SQL que viola a regra 12 de boas práticas.

```
SELECT E.nome
FROM Empregado E
WHERE E.idade > 30
UNION ALL
SELECT E.id
FROM Empregado E
WHERE E.salario < 5000;

-- Consulta Correta:
SELECT E.nome
FROM Empregado E
WHERE E.idade > 30 OR E.salario < 5000;
```

13. Condição de subconsulta pode ser movida para consulta Principal:
Uma condição de subconsulta que só acessa valores da consulta principal não faz sentido. Esta condição pode ser movida para a consulta principal. A Listagem 4.21 apresenta um exemplo onde a condição da subconsulta acessa somente a tabela da consulta principal (E.idade > 30). Portanto, esta condição deve ser movida para a consulta principal.

Listagem 4.21 Exemplo de consulta SQL que viola a regra 13 de boas práticas.

```
SELECT E.nome
FROM Empregado E
WHERE E.dep_id IN (SELECT D.id
   FROM Departamento D
   WHERE D.id = E.dep_id
   AND E.idade > 30);

-- Consulta Correta:
SELECT E.nome
FROM Empregado E
WHERE E.idade > 30 AND
```

```
E.dep_id IN (SELECT D.id
   FROM Departamento D
   WHERE D.id = E.dep_id);
```

14. Uso de DISTINCT em AVG e SUM:

O uso de DISTINCT em funções de agregação AVG e SUM significa perda de informação não intencionada. Por isso, deve-se evitar usar DISTINCT nestes casos. A Listagem 4.22 apresenta um exemplo onde a função de agregação média (AVG) está sendo utilizada com um DISTINCT, ocasionando perda de informação.

Listagem 4.22 Exemplo de consulta SQL que viola a regra 14 de boas práticas.

```
SELECT AVG(DISTINCT E.salario) as AVERAGE
FROM tabela_A A;

-- Consulta Correta:
SELECT AVG(E.salario) as AVERAGE
FROM tabela_A A;
```

15. Uso de caracteres especiais sem uso do LIKE:

Embora existam casos possíveis onde strings podem ser armazenadas no banco com '_' ou '%', existe também o caso em que o programador usou o operador de igualdade por engano. Em uma comparação com uma string que contenha estes caracteres e o LIKE não é usado, um aviso seria interessante. A Listagem 4.23 apresenta um exemplo onde o atributo E.nome está sendo comparado com a string 'Pedro%' por meio de uma igualdade, quando o LIKE deveria ser usado.

Listagem 4.23 Exemplo de consulta SQL que viola a regra 15 de boas práticas.

```
SELECT E.salario
FROM Empregado E
WHERE E.nome = 'Pedro%';

-- Consulta Correta:
SELECT E.salario
FROM Empregado E
```

```
WHERE E.nome LIKE 'Pedro%';
```

16. OUTER JOIN pode ser substituído por INNER JOIN:
Se, em um LEFT JOIN, existe uma condição no WHERE que acessa a tabela à direita, o LEFT JOIN age como um INNER JOIN. Pois nas linhas de A que não tem relação com B, o resultado da condição será *unknown*. Nesse caso, a linha não será retornada. Como a consulta não retorna algo inconsistente, este caso é classificado como boas práticas. A Listagem 4.24 apresenta um exemplo onde a condição do WHERE está se referenciando à tabela da direita (WHERE D.orcamento > 10000). Por causa desta condição, o JOIN que faz mais sentido é o INNER JOIN.

Listagem 4.24 Exemplo de consulta SQL que viola a regra 16 de boas práticas.

```
SELECT *
FROM Empregado E
LEFT JOIN Departamento D
ON D.id = E.dep_id
WHERE D.orcamento > 10000;

-- Consulta Correta:
SELECT *
FROM Empregado E
INNER JOIN Departamento D
ON D.id = E.dep_id
WHERE D.orcamento > 10000;
```

17. Sempre utilizar aliases:
Sempre utilizar aliases nas tabelas, e sempre acessar os atributos por meio de aliases. Estas indicações facilitam a legibilidade e evitam ambiguidades nas consultas. A Listagem 4.25 apresenta um exemplo onde a consulta não se utiliza de aliases.

Listagem 4.25 Exemplo de consulta SQL que viola a regra 17 de boas práticas.

```
SELECT nome
FROM Empregado
WHERE dep_id IN (SELECT id
```

```
   FROM Departamento
   WHERE id = dep_id);

-- Consulta Correta:
SELECT E.nome
FROM Empregado E
WHERE E.dep_id IN (SELECT D.id
   FROM Departamento D
   WHERE D.id = E.dep_id);
```

18. Consultas IN correlacionadas devem se tornar EXISTS:

Se uma consulta IN é correlacionada, ela pode ser substituída por uma consulta EXISTS, onde a semântica faz mais sentido. A Listagem 4.26 apresenta uma consulta que possui uma subconsulta IN, que está correlacionada, por causa da condição *WHERE D.id = E.dep_id*.

Listagem 4.26 Exemplo de consulta SQL que viola a regra 18 de boas práticas.

```
SELECT E.id
FROM Empregado E
WHERE E.dep_id IN (SELECT D.id
   FROM Departamento D
   WHERE D.id = E.dep_id);

-- Consulta Correta:
SELECT E.id
FROM Empregado E
WHERE EXISTS (SELECT D.id
   FROM Departamento D
   WHERE D.id = E.dep_id);
```

19. SELECT DISTINCT desnecessário em subconsultas IN:

O DISTINCT não afeta o resultado nem o funcionamento de uma subconsulta IN, portanto, não deve ser utilizado. A Listagem 4.27 apresenta uma subconsulta IN que contém a palavra DISTINCT na sua lista de retorno.

Listagem 4.27 Exemplo de consulta SQL que viola a regra 19 de boas práticas.

```
SELECT E.id
FROM Empregado E
WHERE E.idade IN (SELECT DISTINCT E2.idade
   FROM Empregado E2
   WHERE salario > 5000);

-- Consulta Correta:
SELECT E.id
FROM Empregado E
WHERE E.idade IN (SELECT E2.idade
   FROM Empregado E2
   WHERE salario > 5000);
```

20. Evitar consultas utilizando NATURAL JOIN:

NATURAL JOIN é uma junção onde a condição de junção se dá pelos atributos de nomes iguais entre as duas tabelas. Isso é perigoso pois a consulta perderá a semântica em caso de alteração do esquema de uma das tabelas. A Listagem 4.28 apresenta uma consulta que se utiliza de NATURAL JOIN, e por isso, viola a regra.

Listagem 4.28 Exemplo de consulta SQL que viola a regra 20 de boas práticas.

```
SELECT *
FROM Empregado E
NATURAL JOIN Departamento D;
```

4.2.5 Boas Práticas - Esquema

Esta subseção apresenta as quatro boas práticas do catálogo que podem ser indicadas quando se utiliza o esquema no qual a consulta está aplicada.

21. Cláusula DISTINCT desnecessária:

A cláusula DISTINCT é desnecessária em consultas que retornem uma chave primária ou um campo UNIQUE. A Listagem 4.29 apresenta uma consulta que contém um DISTINCT, mesmo retornando uma chave primária (E.id).

Listagem 4.29 Exemplo de consulta SQL que viola a regra 21 de boas práticas.

```
SELECT DISTINCT E.id
FROM Empregado E
WHERE E.salario > 5000;

-- Consulta Correta:
SELECT E.id
FROM Empregado E
WHERE E.salario > 5000;
```

22. Atributo em GROUP BY desnecessário:

Se um atributo for funcionalmente dependente dos seus anteriores em uma cláusula GROUP BY, e não aparece em SELECT ou HAVING, ele pode ser retirado desta cláusula. Isso acontece quando atributos são antecedidos por chaves primárias ou campos UNIQUE, por exemplo. A Listagem 4.30 apresenta uma consulta que contém um atributo (E.idade) na cláusula GROUP BY que é determinado pelo atributo anterior à ele (E.id).

Listagem 4.30 Exemplo de consulta SQL que viola a regra 22 de boas práticas.

```
SELECT E.id
FROM Empregado E
GROUP BY E.id, E.idade;

-- Consulta Correta:
SELECT E.id
FROM Empregado E
GROUP BY E.id;
```

23. Termo desnecessário do ORDER BY:

Se um atributo for funcionalmente dependente dos seus anteriores em uma cláusula ORDER BY, ele pode ser retirado desta cláusula. Isso acontece quando atributos são antecedidos por chaves primárias ou campos UNIQUE, por exemplo. A Listagem 4.31 apresenta uma consulta que contém um atributo (E.idade) na cláusula ORDER BY que é determinado pelo atributo anterior à ele (E.id).

Listagem 4.31 Exemplo de consulta SQL que viola a regra 23 de boas práticas.

```
SELECT E.id
FROM Empregado E
ORDER BY E.id, E.idade;

-- Consulta Correta:
SELECT E.id
FROM Empregado E
ORDER BY E.id;
```

24. Comparação de domínios diferentes:
Se dois atributos estão sendo comparados e eles são de tipos diferentes no esquema, pode ser um indicativo de que algo está errado. É uma verificação a mais para validar a semântica da consulta. A Listagem 4.32 apresenta uma consulta que compara dois atributos de tamanhos diferentes. *E.sexo* representa uma string de um caractere, enquanto que *E.nome* representa uma string de 20 caracteres.

Listagem 4.32 Exemplo de consulta SQL que viola a regra 24 de boas práticas.

```
SELECT E.id
FROM Empregado E
WHERE E.nome = E.sexo;
```

4.3 Avaliação

Os exemplos apresentados anteriormente foram executados nos SGBD Oracle e PostgreSQL. O objetivo é avaliar se estes ambientes têm algum tipo de indicação para o usuário, em consultas cujas características se assemelham aos casos do catálogo.

Para o Oracle, foi usada a versão Oracle Database 11g Express Edition. O programa cliente usado foi o SQL Developer, da própria Oracle. Além da simples execução das consultas, foi também utilizada a função Tuning Advisor, que serve para analisar e dar sugestões de como melhorar consultas SQL. No caso do PostgreSQL, foi utilizada a versão PostgreSQL 9.6, e o programa cliente utilizado foi o pgAdmin 4, versão 1.4.

O código de criação do banco de dados está apresentado no Apêndice A. O *script* de criação

do Oracle é apresentado na Listagem A.1. Já o *script* de criação do PostgreSQL é apresentado na Listagem A.2.

A Tabela 4.3 contém uma contagem dos avisos emitidos com a execução dos exemplos do catálogo.

Tabela 4.3 Número de avisos emitidos

	Erros Semânticos		**Boas Práticas**	
	Consulta	**Esquema**	**Consulta**	**Esquema**
Oracle DB	1 aviso de 6	0 avisos de 2	0 avisos de 20	0 avisos de 4
PostgreSQL	0 avisos de 6	0 avisos de 2	0 avisos de 20	0 avisos de 4

Nenhum SGBD deu aviso na grande maioria dos exemplos do catálogo. Apesar da parte de Boas Práticas não ser considerada estritamente errada, até mesmo os Erros Semânticos passaram despercebidos.

A regra do catálogo no qual o Oracle emitiu um alerta foi o de Boas Práticas Número 3 - Tabelas não utilizadas no FROM. O aviso foi emitido pelo Tuning Advisor pois o Oracle detectou uma operação muito custosa - um produto cartesiano - e sugere a retirada de uma das tabelas do produto cartesiano, ou a adição de uma condição de junção. Vale ressaltar que o alerta emitido pelo Tuning Advisor se refere ao produto cartesiano, e não à ausência de uso de uma das tabelas da consulta: o aviso é emitido mesmo se atributos das duas tabelas forem referenciadas. A descrição do alerta está presente na Listagem 4.33.

Listagem 4.33 Alerta emitido pelo Oracle para a regra de Boas Práticas número 3.

```
1- Restructure SQL finding (see plan 1 in explain plans
   section)
-----------------------------------------------------------

  An expensive cartesian product operation was found at line ID
      1 of the execution plan.

  Recommendation
  --------------
  - Consider removing the disconnected table or view from this
     statement or add a join condition which refers to it.
```

CAPÍTULO 5

Conclusão

Este capítulo descreve as conclusões deste estudo, apresentando os resultados obtidos e as propostas de trabalhos futuros.

Neste trabalho foi especificado um Catálogo de regras de boa formação para avaliação de consultas SQL. Neste catálogo foram definidas 32 regras. Destas regras 8 são para avaliação de erros semânticos e 24 são para avaliação de boas práticas. A avaliação deste Catálogo consistiu na análise do suporte dos SGBD Oracle e PostgreSQL às regras de boa formação especificadas no Catálogo. Os resultados destas análises mostram que as regras de boa formação apresentadas no Catálogo não são cobertas por estes SGBD. Vale ressaltar que somente uma regra de boa formação foi capturada pelo Oracle e nenhuma regra foi capturada pelo PostgreSQL.

Os resultados desta pesquisa mostram que erros semânticos são pouco abordados pelos sistemas de gerenciamento de bancos de dados. Por isso, estes erros são difíceis de serem notados. O estudo de técnicas e ferramentas que auxiliem os desenvolvedores a errar menos semanticamente é necessário, a fim de melhorar a qualidade de código SQL produzido.

Para trabalhos futuros, o Catálogo pode ser evoluído para também cobrir regras que analisem os dados no banco de dados. Testes com SGBD diferentes também podem ser incluídos na avaliação, para analisar se outros desenvolvedores implementaram algum tipo de *feedback* para as regras catalogadas. Pode-se utilizar o catálogo como base para desenvolvimento de uma ferramenta completa que analise o código SQL e verifique se as regras do Catálogo estão sendo cumpridas.

Referências Bibliográficas

[APBL16] Alireza Ahadi, Julia Prior, Vahid Behbood, and Raymond Lister. Students' Semantic Mistakes in Writing Seven Different Types of SQL Queries. In *ITiCSE '16*, pages 272–277, 2016.

[BG06] Stefan Brass and Christian Goldberg. Semantic errors in SQL queries: A quite complete list. *Journal of Systems and Software*, 79(5):630–644, 2006.

[FSF15] Robson Fidalgo, Edson Silva, and Natália Franco. Classificando e implementando feedbacks para aprendizado ativo em ferramentas CASE: o caso EERCASE. *Anais do XXVI Simpósio Brasileiro de Informática na Educação (SBIE 2015)*, page 308, 2015.

[Hal17] Hans-Petter Halvorsen. *Structured Query Language*, Acessado em 8 de Julho de 2017. `http://home.hit.no/~hansha/documents/database/documents/StructuredQueryLanguage.pdf`.

[MIS17] MISRA. *MISRA C*, Acessado em 1 de Julho de 2017. `http://www.programmingresearch.com/coding-standards/misra/`.

[MIT17] MITRA. *CWE - Common Weakness Enumeration*, Acessado em 1 de Julho de 2017. `https://cwe.mitre.org/about/`.

[OKS86] WD Ogden, R Korenstein, and JB Smelcer. An intelligent front-end for sql. *IBM, San Jose, CA*, 1986.

[SEI17] SEI. *CERT*, Acessado em 1 de Julho de 2017. `http://www.cert.org/secure-coding/research/secure-coding-standards.cfm`.

[Son17] SonarSource. *SonarAnalyzer for PL/SQL*, Acessado em 30 de Junho de 2017. `http://dist.sonarsource.com/reports/coverage/rules_in_plsql.html`.

[Ubi17] Ubitsoft. *SQL Enlight*, Acessado em 30 de Junho de 2017. `https://www.sonarsource.com/products/codeanalyzers/sonarplsql.html`.

APÊNDICE A

Scripts de Criação

Este apêndice contém os códigos de criação das tabelas envolvidas na avaliação do catálogo. Os códigos estão divididos em: Código para plataforma Oracle (cf. Listagem A.1); Código para plataforma PostgreSQL (cf. Listagem A.2).

Listagem A.1 Código de criação das tabelas no Oracle.

```
CREATE TABLE DEPARTAMENTO
(
  ID NUMBER NOT NULL
, NOME VARCHAR2(20 BYTE) NOT NULL
, ORCAMENTO NUMBER
, CONSTRAINT DEPARTAMENTO_PK PRIMARY KEY
  (
    ID
  )
  ENABLE
);

CREATE TABLE EMPREGADO
(
  ID NUMBER NOT NULL
, SEXO VARCHAR2(1)
, NOME VARCHAR2(20)
, SALARIO NUMBER
, IDADE NUMBER
, DEP_ID NUMBER
, CONSTRAINT EMPREGADO_PK PRIMARY KEY
  (
    ID
```

```
  )
  ENABLE
);

ALTER TABLE EMPREGADO
ADD CONSTRAINT EMPREGADO_FK1 FOREIGN KEY
(
  DEP_ID
)
REFERENCES DEPARTAMENTO
(
  ID
)
ENABLE;

ALTER TABLE EMPREGADO
ADD CONSTRAINT EMPREGADO_CHK1 CHECK
(sexo = 'M' or sexo = 'F')
ENABLE;

------

CREATE TABLE PESSOA
(
  NOME VARCHAR2(20) NOT NULL
, SOBRENOME VARCHAR2(20) NOT NULL
, IDADE NUMBER
, CONSTRAINT PESSOA_PK PRIMARY KEY
  (
    NOME
  , SOBRENOME
  )
  ENABLE
);
```

```
CREATE TABLE ENDERECO
(
  CEP NUMBER NOT NULL
, PESSOA_NOME VARCHAR2(20)
, PESSOA_SOBRENOME VARCHAR2(20)
, CONSTRAINT ENDERECO_PK PRIMARY KEY
  (
    CEP
  )
  ENABLE
);

ALTER TABLE ENDERECO
ADD CONSTRAINT ENDERECO_FK1 FOREIGN KEY
(
  PESSOA_NOME
, PESSOA_SOBRENOME
)
REFERENCES PESSOA
(
  NOME
, SOBRENOME
)
ENABLE;
```

Listagem A.2 Código de criação das tabelas no PostgreSQL.

```
CREATE TABLE public.departamento
(
    id integer NOT NULL,
    nome character varying(20) NOT NULL,
    orcamento integer,
    PRIMARY KEY (id)
)
WITH (
```

```
   OIDS = FALSE
);

CREATE TABLE public.empregado
(
   id integer NOT NULL,
   sexo character varying(1),
   nome character varying(20),
   salario integer,
   idade integer,
   dep_id integer,
   PRIMARY KEY (id),
   CONSTRAINT dep_fk FOREIGN KEY (dep_id)
      REFERENCES public.departamento (id) MATCH SIMPLE
      ON UPDATE NO ACTION
      ON DELETE NO ACTION,
   CONSTRAINT sexo_check CHECK (sexo = 'M' or sexo = 'F')
)
WITH (
   OIDS = FALSE
);

CREATE TABLE public.pessoa
(
   nome character varying(20) NOT NULL,
   sobrenome character varying(20) NOT NULL,
   idade integer,
   PRIMARY KEY (nome, sobrenome)
)
WITH (
   OIDS = FALSE
);

CREATE TABLE public.endereco
(
```

```
    cep integer NOT NULL,
    pessoa_nome character varying(20),
    pessoa_sobrenome character varying(20),
    PRIMARY KEY (cep),
    CONSTRAINT pessoa_fk FOREIGN KEY (pessoa_nome,
       pessoa_sobrenome)
       REFERENCES public.pessoa (nome, sobrenome) MATCH SIMPLE
       ON UPDATE NO ACTION
       ON DELETE NO ACTION
)
WITH (
    OIDS = FALSE
);
```