Aplicação de métodos de análise sintática baseadas em precedência de operadores em software comercial

William Breno Rodrigues Cavalcante

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2024

William Breno Rodrigues Cavalcante

Aplicação de métodos de análise sintática baseadas em precedência de operadores em software comercial

Monografia apresentada ao curso Ciência da Computação
do Centro de Informática, da Universidade Federal da Paraíba,
como requisito para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Alisson Vasconcelos de Brito

Maio de 2024

Catalogação na publicação
Seção de Catalogação e Classificação

C376a Cavalcante, William Breno Rodrigues.
Aplicação de métodos de análise sintática baseadas em precedência de operadores em software comercial / William Breno Rodrigues Cavalcante. - João Pessoa, 2024.
43 f. : il.

Orientação: Alisson Vasconcelos de Brito.
TCC (Graduação) - UFPB/CI.

1. Web API. 2. Tradução de linguagens. 3. Consultas SQL. 4. Análise sintática. 5. Evolução de software. I. Brito, Alisson Vasconcelos de. II. Título.

UFPB/CI CDU 004.4

Elaborado por Michelle de Kássia Fonseca Barbosa - CRB-738

DEDICATÓRIA

Dedico este trabalho aos meus pais, Washington e Miriam.

AGRADECIMENTOS

Agradeço aos meus pais, Washington e Miriam, por proporcionarem as condições e o incentivo para estudar.

Agradeço a toda minha Família por darem apoio durante todos esses anos.

Agradeço ao meu orientador, Prof. Dr. Alisson Vasconcelos de Brito, pela dedicação.

Agradeço aos meus colegas da universidade e da Synchro, com destaque ao Cesar por ter me apoiado durante a autoria deste trabalho.

Agradeço a todas as pessoas que deixaram sua marca na minha vida.

RESUMO

Soluções utilizadas em sistemas de mercado devem ser flexíveis e simples para diminuir custo de desenvolvimento e manutenção. Para que isso seja alcançado, as etapas de desenvolvimento e implementação devem ser acompanhadas de uma investigação estudando as soluções que atendam aos requisitos esperados. A Synchro Soluções Fiscais atende diversos clientes de grande porte, os quais manipulam abundância de dados. Uma nova demanda de negócio surgiu com a necessidade de realizar consultas complexas, associando diversas tabelas no contexto das bases de dados desses clientes, as quais contém as massas de dados a serem retornadas. Este trabalho tem como foco o estudo da aplicação de métodos de análise sintática para a implementação de uma interface flexível que atende grande parte dos requisitos impostos pelos clientes Synchro, que foi projetada para ser facilmente extensível, segundo as demandas do negócio para a evolução do produto. Como solução foi implementado um método de análise sintática por precedência de operadores para atender a necessidade de ter uma linguagem simples, porem robusta e expansível. O resultado do trabalho foi um pacote Java que implementa filtros de dados para a consulta, disponíveis para os usuários em uma API RESTful.

Palavras-chave: <Web API>, <Tradução de Linguagens>, <Consultas SQL>, <Análise Sintática>, <Evolução de Software>.

ABSTRACT

Solutions used in commercial systems must be flexible and simple enough to be able to reduce development and maintenance costs. For this to be achieved, the development and implementation stages must be accompanied by an investigation studying solutions that must meet the expected requirements. Synchro Soluções Fiscais serves several large clients, who handle large amounts of data. A new business demand arose with the need to carry out complex queries, associating several tables in the context of these clients' databases, which contain the masses of data to be returned. This work focuses on studying the application of parsing methods to implement a flexible interface that meets most of the requirements imposed by Synchro customers, and which was designed to be easily extensible, according to business demands for product evolution. As a solution, a syntactic analysis method using operator precedence was implemented to meet the need for a simple, yet robust and expandable language. The result of the work was a Java package that implements data filters for querying, available to users in a RESTful API.

Key-words: <Web API>, <Language Translation>, <SQL Queries>, <Parsing>, <Software Evolution>

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

SCS - Synchro Cloud Services

SQL - Structured Query Language

BP – Binding Power

RBP – Right Binding Power

LBP – Left Binding Power

LHS - Left Hand Side

RHS - Right Hand Side

AST - Abstract Syntax Tree

Sumário

1 INTRODUÇÃO

A coleta de impostos é um dos métodos mais comuns empregados pelas instituições governamentais para arrecadação de renda, e o Governo do Brasil, sendo um deles, é um dos países que tem uma das situações tributárias mais complexas. No Brasil existe uma abundância de impostos e tributos que possuem suas próprias regras que podem variar para os diferentes estados da federação, como o ICMS, e que constantemente alteram de acordo com novas legislações [1].

Para solucionar o problema da tributação nacional, diversas empresas foram fundadas com o princípio de ofertar serviços e produtos para proporcionar uma melhor vigência fiscal. A Synchro Soluções Fiscais [2] foi uma delas, fundada em 1991 e desde então atende diversas empresas nacionais e multinacionais que atuam no mercado brasileiro e que, portanto, precisam estar em conformidade com suas obrigações fiscais.

As soluções da Synchro envolvem a ingestão de uma abundância de dados tributários e em seguida uma série de processos de cálculos para os diferentes tributos. O consumo desses dados, que é a porta de entrada da solução Synchro, é chamado de integração. Da mesma forma, a Synchro também fornece soluções para recuperar os resultados das operações via interfaces de usuário e consultas por API.

1.1 Problema

A Synchro inicialmente construiu um software monólito para atender as necessidades de ter uma API de ingestão de dados e de consulta após o processamento, porém com o incentivo de atender mais clientes e clientes maiores, juntamente com a necessidade de fácil evolução do software, um novo conjunto de produtos SaaS (Software as a Service ou Software como Serviço em português) [3] chamado de Synchro Cloud Services (SCS) foi criado.

O SCS em comparação ao monólito anterior possui uma arquitetura de microsserviços onde nele foram divididas as operações existentes em diferentes serviços, cada um podendo ser desenvolvido por times independentes. Ele possui serviços centrais que fazem parte da infraestrutura como o API Gateway que serve como um reverse proxy e redireciona as requisições para os serviços [4], além disso, o gateway possui outras responsabilidades como rate limiting, que significa controlar a taxa de requisições por segundo do usuário, e de autorização de credenciais levando em consideração quais serviços podem ser utilizados pelo usuário e a validação das credenciais fornecidas pelo mesmo.

Exemplos de componentes do SCS utilizados por todos os serviços são os repositórios e servidores de configuração, que armazenam e disponibilizam as propriedades de configuração para cada serviço. Outros componentes importantes são os de métricas,

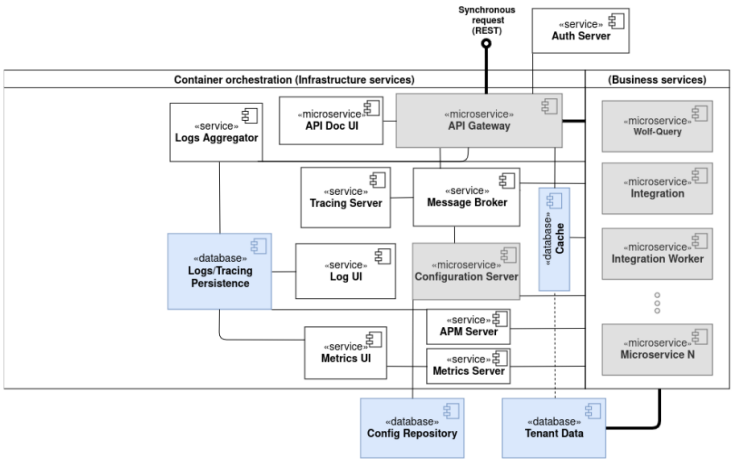


Figura 1: Ilustração dos componentes do SCS

tracing e principalmente os agregadores de logs, importantes para a observabilidade dos microsserviços, proporcionando uma melhor compreensão do comportamento dos serviços.

Por último, o aspecto central das soluções é o acesso e manipulação dos dados do cliente que estão armazenados em banco de dados Oracle, e por isto o SCS e os serviços foram arquitetados para seguirem uma modalidade de multi-tenant, na qual os componentes e serviços sejam capazes de escolher livremente o banco de dados que será acessado segundo o contexto do cliente específico.

Para atender o consumo de dados dos clientes foram criados dois microsserviços, Phoenix-Integration API, responsável pela API REST [5] para criar e gerenciar o processo de inserção e o Phoenix-Integration Worker, onde atual trabalho de inserção dos dados no banco de dados é realizado. Outro microsserviço, chamado de Wolf-Query, foi criado para a consulta dos dados gerados ao longo do processo, que expõe uma API flexível para consultas de dados de diferentes produtos e serviços. Este requisito impõe uma grande importância que a solução escolhida seja o mais robusta possível, pois ela pode gerar altos custos de utilização do banco de dados com consultas de longa duração, porém a solução também precisa ser flexível na implementação de novas features afetando o mínimo possível a semântica das operações existentes.

1.2 Objetivo Geral

Este relatório tem como um dos seus objetivos o estudo do processo de desenvolvimento dos produtos do SCS e as tecnologias adotadas, acompanhando o desenvolvimento de um novo serviço e suas peculiaridades. Também como objetivo é como diferentes tecnologias são empregadas e principalmente modificadas para solucionar problemas técnicos e de usabilidade. O serviço em foco será o Wolf-Query, um novo serviço criado para solucionar as necessidades vindas de usuários para consultar os dados processados de forma

flexível.

1.3 Objetivos Específicos

1 Criação de uma API flexível e intuitiva para consulta de dados genéricos

2 Métodos para implementação de interface para consultas

3 Estudo de algoritmos de parsing por precedência de operadores

1.4 Estrutura do Relatório Técnico

Este capítulo aborda uma breve introdução da problemática e do produto desenvolvido como solução. O segundo capítulo tem como foco o estudo dos frameworks e tecnologias utilizadas na implementação da solução e suas peculiaridades. O capítulo 3 detalha o processo de desenvolvimento e as várias metodologias utilizadas para a escolha das soluções adequadas. O capítulo 4 relata a Prova de Conceito construído a partir do desenvolvimento. Por fim, o capítulo 5 conclui o relatório técnico com as lições aprendidas e propostas de futuras melhorias.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem como proposito uma revisão da literatura e da base teórica que será utilizado na explicação do desenvolvimento e as decisões tomadas para solucionar o problema apresentado neste trabalho.

2.1 Arquitetura de Microsserviços

Quando planejado, inicialmente um sistema possui um conjunto de funcionalidades que deve atender, bem como um número de requisitos mínimos. Com o passar do tempo, seja durante o desenvolvimento ou já em produção, o sistema tende a crescer em quantidade de recursos oferecidos e na capacidade para atender mais usuários. No entanto, esse crescimento tende a ser desigual em relação a quais partes precisa ser "escalado". No desenvolvimento, podem acontecer cenários nos quais vários times trabalhando em recursos idealmente isolados se tornem dependentes uns dos outros, e no pior dos casos entrem em conflito.

A arquitetura com microsserviços propõe dividir um sistema em diferentes componentes de software menores, chamados de serviços (ou como nome indica, microsserviços), que podem ser desenvolvidos separadamente, possivelmente por times diferentes, com um nível menor de acoplamento e que podem ser "escalados"conforme o grau de utilização adequado para cada serviço [6].

Normalmente a arquitetura envolve um conjunto de tecnologias com diferentes propósitos, como, por exemplo:

Gateway Ponto de entrada único para cada um dos serviços, pode concentrar serviços como autenticação e autorização [4].

Conteinerização Como a ideia da arquitetura de microsserviços tem em mente um software reduzido que atende um pequeno número de requisitos, ter ele disponível em uma unidade contida com todas as dependências e configurações contribui na facilidade da implantação, e containers são um conjunto de tecnologias que torna isso possível [7].

Orquestração de containers Idealmente todos os serviços serão implantados em containers trazendo as facilidades que o mesmo providencia, porém com o crescente número desses containers surge a necessidade de gerenciá-los eficientemente. Um sistema de orquestração fornece uma forma de gerenciar as versões e réplicas de cada serviço, juntamente com regras de resiliência como, por exemplo, reiniciar container e fazer "rollout" de novas versões sem downtime [8].

Comunicação entre containers Por mais que os serviços sejam desenvolvidos separadamente e que sejam implantados independente, é comum que serviços comuniquem-se entre si. Um exemplo disso é quando um serviço depende de outro, como, por exemplo, em um sistema que um serviço é responsável pela autenticação de credenciais e os outros serviços utilizam o primeiro para realizar a autenticação de seus usuários. As tecnologias disponíveis para isto pode ser uma simples comunicação por HTTP ou em casos mais complexos envolvem message brokers que são tecnologias especializadas para atender esta necessidade.

2.2 APIs Web com HTTP

Comunicação entre cliente e servidor depende de um protocolo. O TCP/IP, o protocolo base da internet atual [9], fornece uma forma de enviar e receber dados entre dois computadores, porém o protocolo não define a estrutura das mensagens passadas entre os participantes. Usando a camada TCP/IP subjacente, o protocolo HTTP foi criado para definir o formato de documentos estruturados e as operações que podem ser feitas para cada documento [10]. Um exemplo é a operação GET utilizada para o pedido de um documento, enquanto PUT e POST são utilizados para a criação de um novo documento.

Por mais que o HTTP foi criado para a troca de hipertexto, chamado pelo fato de conter hiperlinks que são referências a outros hipertextos, começou rapidamente a ser utilizado para transferência de dados como imagens ou outros formatos de mensagens estruturadas. Atualmente, um formato estruturado amplamente utilizado é o JSON ou "Javascript Object Notation" [11] que contém dados em formato de lista ou pares de chave-valor. Com a utilização de JSON e outros formatos de dados, HTTP começou a servir de base para expor recursos de APIs Web. O protocolo HTTP define recursos que são uma string para identificar o recurso, chamada de URL, em conjunto com operações definidas no protocolo fazem parte de como o HTTP é utilizado para estruturar a comunicação de APIs Web.

2.3 Banco de Dados Relacionais e SQL

Bancos de dados relacionais são bancos de dados baseados no modelo relacional [12], onde cada relação é uma tupla e que cada elemento dessa tupla tem um tipo de dado. As relações são melhor exemplificadas como uma tabela onde o cabeçalho da tabela define os tipos dos dados de cada coluna e as tuplas são as linhas de uma tabela.

Parte do poder de um banco de dados relacional é a linguagem de consultas e outras operações relacionais, mais comumente sendo a SQL (Structured Query Language)

[13]. Com o SQL é possível declarar novas tabelas, deletar, consultar tabelas entre outras operações.Atualmente o SQL é o padrão do mercado, porém certas empresas expandiram a capacidade da linguagem, podendo criar como, por exemplo, procedures, estruturas mais comumente encontradas em linguagens de programação de propósito geral.

2.4 Linguagem de computador

Como foi discutido, linguagens na computação são utilizadas para diversas tarefas além de programação. Além de SQL para consultar existe também GraphQL (Graph Query Language) [14], uma linguagem mais recente para a consulta de dados em bancos "grafos" e as duas são classificadas de Query Languages, outro exemplo ubíquo é o HTML (HyperText Markup Language) que é normalmente usado junto o protocolo HTTP e como o nome indica é uma Markup Language, linguagem usada para anotar documentos com formatação e outros aspectos da estrutura do texto [15]. Então, linguagens na computação podem ser observadas como forma de expressar uma operação ou configuração. Outra classificação de linguagem seria as DSLs (ou Domain Specific Language) [16] que surgiram como linguagens específicas para exercer apenas uma tarefa, como, por exemplo, para configuração de build de projetos ou definição de testes integrados.

A implementação de uma linguagem de computador envolve diversos passos, os quais podem ser divididos em: (i) análise gramatical, que consiste em ler as palavras do texto e a estrutura que o compõe, e (ii) execução ou interpretação da semântica do texto. O software que realiza essa interpretação de texto é chamado de interpretador ou compilador, dependendo se a saída desejada é o resultado da execução do texto ou um software executável [17].

2.4.1 Parsing de Linguagens

A estrutura de um interpretador ou compilador de uma linguagem é dividida em duas partes:

Frontend Tem o propósito de ler a entrada e gerar uma forma intermediária baseada em uma estrutura de dados que seja adequada para processamento.

Backend Usa a forma intermediária para interpretação ou compilação.

Por sua vez, o Frontend pode ser dividido em mais duas fases:

Tokenização ou Scanning O texto de entrada é lido em tokens, que são símbolos ou palavras da linguagem. Um token pode conter o texto de qual foi derivado e outras informações como o seu tipo ou categoria sintática.

Análise Sintática ou Parsing O papel desta fase é verificar se a sequência de tokens lidas pelo Scanner pertence à linguagem. Como produto deste processo, uma forma intermediária da linguagem pode ser produzida.

2.4.2 Análise Sintática

A gramática de uma linguagem contem um conjunto de regras são chamadas de produções ou derivações, e são elas que definem todas as possíveis frases ou strings que pertencem à linguagem. Uma produção é formada por símbolos terminais e não-terminais, e são anotadas da seguinte forma: A $\rightarrow$ Ba, onde A ou lado esquerdo da produção é um não-terminal e Ba ou lado direito é um conjunto de não-terminas e terminais. Durante a análise sintática é construído uma parse tree, derivation tree ou arvore sintática, uma árvore em que os nós são símbolos terminais ou não-terminais, sendo que terminais sempre são as folhas da árvore e não-terminais os nós internos com um ou mais filhos.

Algoritmos de análise Sintática são classificados da forma que o parsing é feito:

Top-Down O parser inicia pela produção do nó raiz e usa as derivações para assumir as estruturas inferiores, tentando correspondê-las com uma alternativa que satisfaz a produção.

Bottom-Up O parser tenta reconstruir a árvore sintática a partir das folhas e construindo as estruturas intermediárias até o nó raiz.

Exemplos de algoritmos Bottom-Up são os parsers LR (Left-to-Right, rightmost derivation) como o LR(1), SRL(1) e LALR(1) [17], implementações desses algoritmos estão disponíveis em pacotes de software como a suíte GNU Bison/Flex, que é um gerador de parser. Já exemplos de algoritmos Top-Down é LL (Left-to-Right, leftmost derivation), sendo o ANTLR, um gerador de parsers LL(*). Outro algoritmo é o Recursive Descent [17] que é um conjunto de funções mutuamente recursivas onde cada uma implementa uma produção da gramática, este método é notável por normalmente ser escrito à mão diferente de outros métodos utilizado por geradores de parsers.

Para um produto desenvolvido comercialmente que será utilizado por empresas, o custo de desenvolvimento para um conjunto de software para uma linguagem simples pode ser alto, então alternativas como o ANTLR [18] e GNU Flex/Bison [19] são soluções para facilitar o desenvolvimento. Uma desvantagem de utilizar ferramentas existentes são a de as mesmas podem ser grandes dependências com uma complexidade alta, que existem para atender até mesmo os casos mais complexos, então o ideal seria uma forma de implementar uma simples linguagem com um algoritmo simples ou com uma dependência menor.

Expressões em linguagens variam de complexidade, mas algo comum a todas é a existência de operadores e operandos como em expressões matemáticas, e a gramática que envolve esses componentes pode ser reduzida em operações da análise mais simples que pode ser facilmente generalizada. O algoritmo Shunting Yard criado por Edgar W. Dijkstra [20] exemplifica este conceito por ser um algoritmo simples que não utiliza estruturas de dados complexas ou operações complexas.

2.4.3 Shunting Yard

O algoritmo possui três estruturas, (i) a fila de entrada, (ii) a fila de saída e (iii) a pilha de operadores, sendo esta última de grande importância. Para uma melhor compreensão, entende-se por combinação uma operação que consome dois operandos e resulta em um valor, símbolo ou qualquer outra estrutura validada de saída.

Algorithm 1 Shunting Yard

```
1:  procedure parseExpresssion
2:      S ← ∅
3:      while Q_e ≠ ∅ do
4:          token ← Dequeue(Q_e)
5:          if token é um numero then
6:              Enqueue(Q_e, currToken)
7:          else if token é um operador then
8:              if token.precedence < Head(S).precedence then
9:                  operand_1 ← Dequeue(Q_e)
10:                 operand_2 ← Dequeue(Q_e)
11:                 exp ← Combine(token, operand_1, operand_2)
12:                 Enqueue(Q_e, exp)
13:             else
14:                 Push(S, token)
15:             end if
16:         end if
17:     end while
18:     while S ≠ ∅ do
19:         operand_1 ← Dequeue(Q_e)
20:         operand_2 ← Dequeue(Q_e)
21:         exp ← Combine(currToken, operand_1, operand_2)
22:         Enqueue(Q_e, exp)
23:     end while
24: end procedure
```

O algoritmo funciona da seguinte forma, assumindo que operadores sempre associam à esquerda:

1. Enquanto tiver símbolos restantes na entrada, remover um da entrada

(a) Caso o símbolo lido seja um operando, o enfileire na fila de saída

(b) Caso o símbolo lido é um operador, dois casos são possíveis:

* O operador no topo da pilha de operadores tem menor precedência ou a pilha está vazia: o operador lido é empilhado na pilha de operadores.
* O operador no topo da pilha de operadores tem maior precedência ou possui a mesma precedência: dois operandos são removidos da fila de saída, o operador no topo é desempilhado e é combinado com os operandos, então o resultado é inserido na fila de saída.

2. Se estivar operadores sobrando, desempilhá-los e combiná-los com valores da fila de saída.

3. Caso a fila de entrada e a pilha de operadores estiverem vazios, o algoritmo finalizou com sucesso, caso não todos os operandos e operadores não conseguiram ser combinados e restaram símbolos.

A ordem de associatividade é apenas considerada quando o operador lido e o operador no topo da pilha tem a mesma precedência e, portanto, a associatividade é usada para escolher um dos casos de operador apresentados acima. Caso os operadores associem a esquerda, o operador no topo da pilha é usado, já no caso dos operadores associarem a direita, o operador lido da entrada é utilizado.

2.4.4 Pilha de chamada de funções

Programação estruturada tem como um dos seus principais componentes as funções, que são blocos de código com um número de parâmetros. Especificamente, a definição de uma função possui parâmetros formais que definem variáveis que serão substituídos por argumentos na chamada da função. As expressões que, após avaliação serão os argumentos, precisam de um espaço em memória alocando uma referência a esses valores e, ao mesmo tempo, no final da função, os argumentos precisam ser liberados para diminuir o consumo de memória. Para solucionar esse problema, funções ou procedures como são chamadas quando não possuem valor de retorno, utilizam de pilhas.

As pilhas são estruturas de dados LIFO do inglês Last In First Out (Último a entrar é o primeiro a sair) e elas são utilizadas nas chamadas de funções para implementar a alocação de espaço dos argumentos. No início da chamada da função, antes de executar a sequência de instruções referente ao bloco da função, os argumentos são empilhados em uma ordem específica conforme a plataforma de programação ou arquitetura do processador. Após os argumentos serem empilhados, é executado o bloco da função que, por sua vez, desempilha os argumentos inseridos no final da execução, retornando a pilha para o

estado antes da chamada da função. A pilha dedicada para esse papel é chamada de call stack.

2.4.5 Utilizando a Call Stack como Pilha

O algoritmo Shunting Yard utiliza uma pilha para ordenar o processamento dos operadores, sendo o topo da pilha o operador de maior precedência e o último operador de menor precedência. Uma alteração do algoritmo seria utilizar a pilha de chamadas de funções, a call stack como a pilha de operadores. O algoritmo então possui uma função central recursiva:

Algorithm 2 Precedence Climbing

```
1: function parseExpression(minPrecedence)
2:     lhs ← Q_e
3:     loop
4:         token ← read(Q_e)
5:         if token = EOF ou token.precedence > minPrecedence then
6:             exit loop
7:         end if
8:         rhs ← parseExpression(token.precedence)
9:         lhs ← Combine(token, lhs, rhs)
10:    end loop
11:    return lhs
12: end function
```

A função funciona da seguinte forma:

1. Inicialmente a função é chamada com argumento de precedência mínima, ou seja, sempre menor que a precedência de todos os outros operadores ou igual ao operador de menor precedência.
2. Um token operando é lido da entrada e terá o nome dado de lhs
3. Laço
   - 3.1 Ler um símbolo, caso ele seja EOF, sair do laço, caso seja um operador op é dado o nome de lhs.
   - 3.2 Se o operador lido tiver maior precedência que como argumento, sair do laço, caso contrário lhs terá um novo valor da forma: $lhsopparseExpression(op.precedencia)$
   - 3.3 Voltar para 2.1.
4. Retornar o valor lhs como resultado

Este método de parsing é conhecido como Precedence Climbing e foi descrito por Richards e Whitby-Strevens [21]. Precedence Climbing é um exemplo de um algoritmo de Parsing por Precedência de Operadores como o Shunting Yard, e é comumente classificado como um algoritmo Top-Down, diferente do algoritmo Shunting Yard. Precedence Climbing se assemelha com Recursive Descent, porem com as funções mutuamente recursivas "compactadas" em apenas uma que consegue tratar de todos os operadores.

2.4.6 Pratt Parsing

Antes de Richards e Whitby-Strevens [21] terem descrito Precedence Climbing, Vaughnan Pratt sugeriu no seu artigo de nome Top-Down Precedence Parsing o algoritmo que ficou conhecido como Pratt Parsing [22] e é atualmente considerado uma generalização de Precedence Climbing.

Em Pratt Parsing tres terminologias são definidas:

bp ou binding power precedência de um operador, métrica que dita qual operador tem prioridade de vinculação com os operandos.

nud ou null denotation código relacionado a um token que não possui uma expressão a sua esquerda como argumento.

led ou left denotation código relacionado a um token que possui duas expressões como argumento, a expressão à esquerda do token e outra à direita.

Ambos *nud* e *led* são códigos relacionados a um operador que tratam como o parsing será feito para aquele operador. Como um exemplo simples é o operador '−', que poderia ter como led um código que espera que estejam presentes uma expressão aritmética a esquerda e à direita, sendo eles os dois argumentos do código led, já para *nud* seria apenas uma única expressão aritmética à direita o qual o resultado seria a negação dessa expressão.

Utilizando o *nud* e *led* o algoritmo tem a seguinte função principal, responsável por realizar a análise sintática de uma sub expressão:

O laço funciona da seguinte forma:

1. Um token é lido da entrada
2. O *nud* associado a esse token é executado e o resultado é guardado em *lhs* ou left hand side
3. Na condição do laço, o *lbp*, ou left binding power, do próximo token é comparado com o *rbp*, ou right binding power da sub expressão atual.

Algorithm 3 Pratt Parser

```
1: function parseExpression(rbp)
2:     token ← read(Q_e)
3:     lhs ← token.nud()
4:     while peek(Q_e).lbp > rbp do
5:         token ← read(Q_e)
6:         lhs ← token.led(lhs)
7:     end while
8: end function
```

(a) Caso $lbp > rbp$, isso significa que o operador anterior não possui preferencia para se associar com o token a sua imediatamente à sua direita, portanto led do token lido é chamado com a sub expressão a sua esquerda e o resultado se tornando o novo lhs, após isso o laço retorna para o início.

(b) Já no caso contrário, isso sendo $lbp \leq rbp$, significa que o próximo operador tem uma precedência menor para se associar com o token imediatamente à sua esquerda, à direita da sub expressão atual, portando o laço finaliza e a função retorna lhs.

Em Pratt Parsing, o laço principal não contém a recursão do método de parsing e esta chamada recursiva é feita conforme o *nud* ou *led* do operador, resgatando o exemplo do operador '−', o seu led irá chamar o método para preencher o seu argumento a direita, com o argumento *rbp* sendo o seu próprio.

Algorithm 4 Infix Negation *led*

```
function parseNegation(token, lhs)
    rhs ← parseExpression(token.rbp)
    return negate(lhs, rhs)
end function
```

Para tratar parênteses basta com que o token '(' tenha um *nud* que chama o método de parsing com *rbp* mínimo e espera que o token ')' seja o próximo token de entrada.

Algorithm 5 Prefix Negation *nud*

```
function parseNegation(token)
    rhs ← parseExpression(token.rbp)
    return negate(rhs)
end function
```

Com esses exemplos fica claro como Pratt Parsing pode ser usado para diversos casos, sendo possível até mesmo implementar parser para expressões de $if - then - else$, como foi demonstrado por Pratt [22]. Outra observação é a similaridade entre os algoritmos de Precedence Climbing e Pratt Parsing, discutido por Andy Chu no blog do seu projeto OilShell [23].

3 DESENVOLVIMENTO

No progresso de desenvolvimento varias tecnologias principais tecnologias foram estudadas, sendo elas padrões de mercado ou padrões do ecossistema adotado. Neste capitulo será abordado como essas tecnologias foram empregadas e como elas tiveram impacto nas decisões tomadas. Também é abordado como métodos e algoritmos foram adotados para solucionar principal problema de disponibilizar uma API de consultas flexível.

3.1 Base tecnológica

A arquitetura do SCS como descrito no trabalho de Cesar et al. [24] é baseada na arquitetura de *microsserviços multi-tenant*, para alcançar os requisitos da empresa de servir aplicações com múltiplos clientes (tenants) e obter benefícios na escalabilidade, gerenciamento e integração de serviços de terceiros.

Os serviços contidos em um cluster devem seguir uma padronização no que diz respeito às formas de comunicação entre os serviços e como cada um é exposto para a internet. A comunicação interna dos serviços SCS é feita por meio de três protocolos/-serviços, sendo eles: API HTTP, Filas AMQP e tabelas de bancos compartilhadas. HTTP é utilizado como a porta de entrada do serviço que será consumido pelo cliente, portanto ela define o contrato entre o serviço e o cliente. Filas são importantes para a coordenação, que inclui a distribuição de trabalho entre os serviços. Já utilizar bancos é quando é necessário uma maior robustez na preservação dos dados ou os dados são sigilosos e devem ser contidos em um ambiente exclusivo. A mais importante para esse trabalho é a API REST, por ser o protocolo de entrada do serviço desenvolvido e a sintaxe dos parâmetros serem o que será ultimamente traduzido para requisições SQL.

O serviço Wolf-Query, que é o objeto do presente trabalho, foi criado a partir do framework Java Spring Boot. O Spring Boot é um framework modular idealizado para construção de aplicações genéricas que podem ser utilizadas no contexto de microsserviços. A modularidade da framework é realizado por dependências Spring que são facilmente integradas e expandem a funcionalidade do framework, essa facilidade de expansão são resultados de dois aspectos da framework: (i) o Spring Boot é um framework baseado na ideia de dependency injection, na qual existem pontos no código do framework que podem ser substituídos ou referenciam uma implementação, e (ii) a autoconfiguração de injeção das dependências na aplicação. Um exemplo das capacidades do Spring Boot é na utilização de bancos de dados, no qual o Spring instancia automaticamente os objetos necessários e os configura injetando-os nas referências existentes no código, tornando o código cliente do framework independente do RDBMS utilizado.

Para comunicação externa, os serviços do SCS utilizam um componente com a

função de implementar o padrão reverse proxy, materializado pelo gateway. Esse gateway tem dois papeis fundamentais: (i) expor uma única porta de entrada aos demais serviços e (ii) a autorização dos serviços seguindo o protocolo OAuth2 com OpenID [25][26]. O projeto do Gateway também é baseado no framework Spring Boot, mais especificamente o Spring Cloud Gateway que aproveita tecnologias de programação reativa que oferece um melhor desempenho para um fluxo non-blocking.

3.2 Consultas de Dados

O Wolf-Query foi criado para substituir uma funcionalidade que estava disponível em um produto monólito legado, com o objetivo de disponibilizar uma forma de o cliente recuperar dados dos bancos Synchro. Essa tecnologia legada baseava-se em padrões arquiteturas antigos, cuja arquitetura dificultava a extensão de funcionalidades e manutenibilidade do código. Além disso, o desempenho percebido pelos usuários não era satisfatório, portanto carecia de uma modernização tecnológica.

Pode-se considerar os bancos de dados da Synchro como os componentes centrais para realização do processamento das transações, pois neles tanto são armazenados os dados dos clientes, como são executados diversos processos de cálculo tributário. O Wolf-Query tem como propósito disponibilizar uma forma de consultar os resultados parciais e intermediários das execuções realizadas no banco.

Para exemplificar uma utilização do Wolf-Query, primeiro é necessário discutir como os dados entram no banco de dados. Clientes que desejam realizar cálculos tributários precisam primeiro realizar uma integração e esta integração contém diversos dados como documentos fiscais, dados de mercadorias e de pessoas. O serviço do SCS que disponibiliza esse processo de ingestão de dados é o Phoenix-Integration, que como descrito por Cesar et al. [29] é um sistema que utiliza de filas AMQP para distribuição de trabalho, e no fim da execução da integração o mesmo dispara os processos de banco que realizam os cálculos tributários. Depois da ingestão e do disparo do processo de banco, é necessário existir uma forma do cliente buscar o resultado da execução, para ser possível ter conhecimento se o processo executou com sucesso e em caso falha quais são e como corrigi-las. Porém, essa consulta, devido a características de como o processo foi implementado e da natureza dos dados, pode envolver tabelas com inúmeras colunas e inúmeras linhas, não sendo incomum existar tabelas com mais de 200 colunas e mais de 1 milhão de registros. Então o Wolf-Query foi criado para fornecer a API de consulta que consegue filtrar esses conforme a necessidade do cliente e podendo ser configurado com os limites com a intenção de diminuir custos com processamento de banco e banda de rede.

As consultas do Wolf-Query são disponibilizadas a partir de uma API REST, da mesma forma que o monólito anterior, sendo essas consultas com a possibilidade do cliente

utilizar de operações de filtragem para diminuir custos de processamento e transmissão de dados.

3.3 OData

A API foi inicialmente projetada para suportar parcialmente o protocolo Open Data Protocol (OData) [27][28], sendo esse um protocolo criado para prover uma API HTTP utilizada para realizar consultas e outras operações em banco de dados.

O OData, como demonstrado anteriormente, é um protocolo complexo que disponibiliza vários tipos de saída e diversas operações. Portanto, para o serviço desenvolvido, apenas a operação de consulta era relevante, em especial o conjunto de filtros do operador filtro, que é o escopo central deste trabalho.

3.4 Implementação Reduzida do protocolo OData

Para que o OData seja adotado primeiramente foi investigado as dependências disponíveis no mercado que conseguem disponibilizar uma implementação do OData a partir de especificações necessárias. O requisito da dependência de maior importância é a capacidade da implementação ser flexível e dinâmica o suficiente para se adequar a datasets, como é chamado a tabela ou view do banco de dados, que podem ser adicionadas após a autoria do software e serem configuradas de forma online. Algo que também foi considerado é o custo da manutenção de qualquer dependência seja adotada, seja pela implantação inicial que pode necessitar de um planejamento e codificação para que software, seja compatível, quanto a manutenção associada a atualização da dependência.

Após investigações foi concluído que as soluções disponíveis no mercado não eram adequadas, pois traziam um nível de complexidade ao projeto e a maioria das dependências era incompatível com a flexibilidade necessária para o projeto. Como solução, uma implementação parcial do OData seria mais adequada ao cenário.

Foi escolhido para implementação duas operações do OData: (i) `$metadata` que recupera os metadados de uma tabela, sendo eles as colunas e os respectivos tipos, e (ii) `$select` o operador de consulta. O operador de consulta possui algumas opções esperadas como selecionar quais colunas serão projetadas ou selecionadas, a opção de `$skip` utilizado para pular um número de colunas e `$top` para recuperar apenas um número limitado de linhas. As opções `$skip` e `$top` em conjunto são utilizados para paginação, uma forma de disponibilizar ao usuário um tamanho controlado de dados transferidos por requisição, podendo também usá-los para recuperar o próximo conjunto de dados.

Neste escopo, a opção de consulta do OData que foi implementado foi o `$filter`. O `$filter` é a opção que possibilita o usuário formular sua consulta mais precisamente.

Fazendo um paralelo com o SQL, o `$filter` tem o papel da cláusula `WHERE` que funciona como uma expressão booleana em que o resultado da avaliação da expressão é usada para a decisão de filtragem dos dados, sendo descartada caso o resultado seja falso. Como definido no OData, o filtro tem diversos operadores e a maioria deles possui uma tradução direta para SQL.

Tabela 1: Operadores de filtro OData e seus equivalentes em SQL

`$filter`	**SQL**
`eq`	`=`
`ne`	`!=`
`lt`	`<`
`gt`	`>`
`le`	`<=`
`ge`	`>=`
`and`	`&`
`or`	`|`
`not`	`!`
`in`	`in`

Devido à similaridade, uma opção de implementação seria a utilização de expressões regulares e simples substituições de símbolos na construção do SQL final. Uma grave falha nessa implementação é a falta de cuidado com quais operadores e operandos estão sendo utilizados, podendo causar uma vulnerabilidade de SQL Injection.

SQL Injection é uma falha de segurança em projetos que utilizam SQL. Ela surge em operações nas quais um usuário pode enviar dados arbitrários de entrada que serão utilizados na criação do comando SQL que finalmente será usado na consulta ao banco de dados. Essa criação quando feita de forma ingênua envolve geralmente a concatenação de strings. Por exemplo, concatenar o início do SQL que tem os campos selecionados com a cláusula `WHERE` que tenha sido criada concatenando operandos sendo estes enviados pelo usuário. Um exemplo dessa situação seria em um campo de busca, onde o usuário mal intencionado pode injetar outros trechos de SQL que, quando concatenados, geram uma query maliciosa. Em casos mais extremos, possibilita o acesso a dados sigilosos, alteração ou criação de dados pertencentes a outro usuário, cenários esses catastróficos para o negócio.

Para evitar a vulnerabilidade de SQL Injection, o código que interage com o banco deve sempre validar os dados enviados, identificando os tipos dos dados corretos e se tem o formato esperado. Dessa forma, é possível detectar que um simples dado enviado para um campo "nome" por exemplo, não pode conter símbolos como `<=`, `DELETE` e `;`, que são strings válidas na sintaxe SQL.

```
(1) username = "user"
(2) "SELECT * FROM USERS WHERE username = " + username + ";"
                ⇓
(3) "SELECT * FROM USERS WHERE username = user;"
```

```
(1) username = "user;DROP TABLE USERS;"
(2) "SELECT * FROM USERS WHERE username = " + username + ";"
                ⇓
(3) "SELECT * FROM USERS WHERE username = user;DROP TABLE USERS;"
```

Figura 2: Exemplo de concatenação de string que pode criar vulnerabilidades

3.5 Parsing dos filtros

À primeira vista, a opção de filtros pode parecer algo simples, porém a partir de uma melhor observação das capacidades conclui-se o oposto. Nela, é definida uma sintaxe com uma gramática livre de contexto, como apenas as simples expressões comuns às linguagens de programação de propósito de geral, de forma mais genérica, uma expansão das expressões em calculadoras digitais. A sintaxe tem operadores que tem como parâmetros tipos específicos e valores resultando seus próprios tipos. Desta forma, eles podem ser combinados para expressar o filtro que o usuário deseja. Além dos operadores comuns como maior ou igual ($\geq$), diferente de ($\neq$), a gramática também define funções utilizadas como, por exemplo, funções específicas para manipulação de strings.

Uma implementação que cubra todas as possibilidades vindas dos filtros OData pode ser algo custoso, então bibliotecas precisam ser utilizadas para aliviar o custo da implementação. Entretanto, isto traz de volta o problema anteriormente exemplificado: adicionar novas dependências ao código tem por si seus custos de manutenção, e essa dependência pode ter um peso tanto em complexidade quanto na implementação que a torna inviável.

3.6 Implementando Parsing por Precedência de Operadores

Para solucionar o problema apresentado na seção anterior, foi escolhido que um analisador sintático simples deve ser escrito atendendo os requisitos mínimos.

3.6.1 Scanner ou Lexer

Para atender os requisitos foi implementado um parser caseiro, esse parser foi feito baseado em Precedence Climbing. Antes disso, um Scanner, ou Tokenizador simples foi escrito que funciona da seguinte forma:

1) Um caractere é lido da entrada

2) Caso seja um caractere branco (space, newline, carriage return) volta para o passo anterior

3) O caractere então se enquadra em algum dos casos

   Dígito neste caso dígitos consecutivos são consumidos e um token de número é gerado

   Letra após o primeiro caractere, enquanto próximos caracteres forem alfanuméricos eles são consumidos e neste caso o token resultante pode ser um identificador caso seja desconhecido ou um operador formado por letras como 'eq'.

   Símbolo o caractere pode fazer parte de um símbolo como "(" ou "!", gerando um token do tipo adequado.

   Aspas simples os caracteres são consumidos até encontrar outra aspa simples e o texto interno é considerado um token do tipo string.

Alguns detalhes foram omitidos na descrição acima, como números reais que possuem ponto e como é decidido se uma sequência de caracteres alfanuméricos é um identificador ou um operador, porém a implementação é trivial.

3.6.2 Arvores Sintática Abstrata

O parser foi implementado em uma classe que mantém o contexto atual e o centro deste parser é o método `parseExpression`, capaz de ler um texto e traduzir em nó de uma árvore sintática abstrata. Esse método tem como retorno um nó do tipo da árvore abstrata e como argumento *bp* ou *bindingpower*, nome dado ao valor da precedência do operador.

A árvore abstrata, ou AST, é um conjunto de classes Java que deriva de uma única classe pai que representa todos os nós. Três classes da AST são as mais importantes: (i) uma que representa uma operação infixa e possui dois parâmetros, (ii) outra que representa operações prefixas e (iii) a última que atômica que representa valores. Nós de operadores são simples, possuindo apenas os operandos e o tipo da operação. A classe que define operadores infixas inclui tipos de operadores como "eq" (igual), "gt" (maior ou igual), "and" (predicado e) e como único operador prefixo é o "not" (inversão do predicado). Já o nó de valores é mais complexo no sentido do que pode representar, nos casos mais simples, os valores literais como números e strings e, para casos mais complexos, identificadores que são referências às colunas de uma tabela. Por último, pode-se considerar que uma expressão em si é um valor, e isso é útil ao lidar com marcações de parênteses.

3.6.3 Parser de Expressões

O método é inicialmente chamado com o valor *bp* (binding power ou a precedência do operador) mínimo e isso faz com que quaisquer operadores lidos tenham prioridade. A ideia é que a prioridade dos operadores esteja implícita na pilha de chamada de funções, então começar com o valor mínimo é uma forma de indicar que a pilha está vazia.

O primeiro passo é executar um estado similar ao *nud* descrito em Pratt Parsing. Isso envolve alguns casos:

- O caso inicial é que o próximo token lido é um valor literal como String ou número e é considerado resultado de um nó da AST criado com o token.
- Outro caso é de ter sido lido um operador que deve ser um operador prefixo, pois neste estado ainda não foi lido o operando da esquerda de um operador infixo.
- O último caso de *nud* é os parênteses esquerdo, que funciona como um operador prefixo e ele causa que o método seja chamado recursivamente. Após isso é esperado que o parênteses direito esteja na entrada.

Algorithm 6 Parse Atomic

```
function parseAtomic(minBp)
    if peek() = LParen then
        consumeToken()
        expr ← parseExpression(0)
        token ← consumeToken()
        if token ≠ RParen then
            throwError
        end if
        return expr
    end if
    if peek() é um operador prefixo then
        return parsePrefixExpression(minBp)
    end if
    return parseValue()
end function
```

O resultado do primeiro passo é nomeado de *lhs* ou *left hand side*, pois ele servirá como operando do lado esquerdo de um operador infixo. O método como descrito em Precedence Climbing e Pratt Parsing começa com um laço e este tem os seguintes passos:

1. Um token da entrada é lido, porém não consumido. Este token deve ser um operador. Caso contrário, o laço é encerrado.

2. Após a leitura do operador, o binding power dele é comparado com o binding power que está como argumento do método. Caso seja menor, o laço é finalizado. Caso contrário, o operador é consumido, o método é chamado recursivamente e o resultado da chamada é chamado de *rhs* ou *right hand side*.

3. Um novo nó de operador infixo é criado, com o tipo de operação a partir do operador lido. Sendo *lhs* e *rhs* os operandos, o novo nó é o novo *lhs* e o laço volta para o início. O laço só finaliza quando encontra um parênteses ou *EOF* ao tentar ler o operador. Para casos além desses, é considerado um erro de análise sintática.

4. Em caso de sucesso o método irá retornar um nó da AST, o qual é a raiz. Caso falhe, uma exceção de erro será lançada.

3.7 Avaliação da AST

A análise sintática da expressão nos dá como resultado uma representação da expressão, para que termos o resultado, que é um código SQL valido, é preciso avaliar essa representação, a AST. Como avaliar a AST é um assunto importante, pois nele será introduzido as verificações que validam a expressão, garantindo que o SQL final esteja livre de vulnerabilidades.

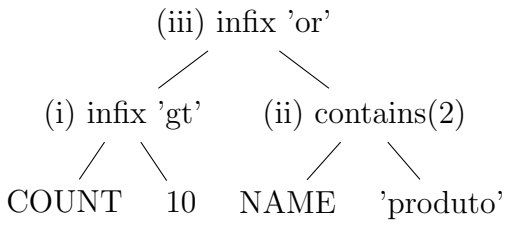
3.7.1 Tradução para SQL

Até o momento, o parsing das expressões teve apenas como resultado prático validar se a frase pertence ou não na gramática. Entretanto, o produto dele (a AST) será usado na avaliação que tem como resultado a geração de um código SQL. Mais especificamente, será analisada a cláusula "WHERE" equivalente. O texto do SQL é feito de forma simples, concatenando os operandos com o operador de forma que seja um SQL válido. Para isso, os operandos precisam ser avaliados e o resultado ser utilizado na formatação do texto. Dessa forma, a execução da avaliação é similar ao algoritmo Depth First Search (DFS), no qual os nós da árvore são avaliados descendo recursivamente para os nós mais profundos antes de avaliar os nós irmãos, conforme exemplificado na Figura 3.

Parênteses são adicionados no texto para sobrescrever a ordem de precedência padrão do SQL. Isso é feito de forma conservadora mesmo que a ordem não seja diferente no SQL. Assim, é criada a tradução da expressão da linguagem dos filtros para o SQL equivalente.

3.7.2 Verificação dos Tipos

Outro passo importante é a verificação dos tipos ao construir o SQL. Isso garante que o texto resultante é um SQL válido. Além de confiabilidade, a verificação dos tipos



i infix 'gt' → `"COUNT > 10"`

ii contains(2) → `"NAME like '%produto\%'`

iii infix 'or' → `"(COUNT > 10) || (NAME like '\%produto\%')"`

Figura 3: Exemplo de avaliação da AST, o processamento da árvore se assemelha ao Depth First Search, com a ordenação de visita Post-Order, visitando o nó filho da esquerda (i), depois o da direita (ii) e por último o nó parente (iii)

é necessária para que colunas referenciadas existam e que possuam o tipo de dado compatível com o operador. Este é um passo essencial para garantir que não exista nenhuma forma do código ser vulnerável a SQL Injection ou outras vulnerabilidades, pois todos os aspectos semânticos são levados em consideração ao gerar a tradução.

É importante ressaltar que os metadados do banco são recuperados com o intuito verificar os tipos das colunas. Cada RDBMS tem sua forma de disponibilizar essa informação. No caso de bancos Oracle, basta realizar uma consulta na família de tabelas `tab_columns`. Essa informação de metadados faz parte do contexto de avaliação.

3.7.3 Funções

Uma expansão do parser foi o suporte de funções. A adição no parser para funções é novo passo no tratamento de *nuds*, como valores atômicos (literais ou identificadores de colunas) e expressões entre parenteses. A sintaxe de uma função expresso por um token de identificador seguido por um parêntese esquerdo (ex.: "contains"seguido de "("), que deve ser fechado com um parêntese direito similar como expressões entre parênteses são analisadas. Diferente de uma expressão entre parênteses, uma chamada função tem entre os parênteses uma lista de expressões delimitada por vírgulas e esta lista de expressões será os argumentos para os parâmetros da função.

O corpo da função é avaliado por substituição, ou seja, a avaliação da função gera uma expressão SQL com pontos dessa expressão substituídos pelos argumentos, pontos chamados dos parâmetros informais da função. Uma implementação da função `contains(<string>, <string>)` é demonstrado na Figura 4.

i `contains(2) = #1 like $'\%' <> #2 <> '\%'`

ii `contains(NAME, 'MARIA')| $\to$ \verb|NAME like '%' || 'MARIA' || '%'`

Figura 4: Exemplo de uma função (i), onde #1 e #2 é o ponto de substituição do primeiro e segundo argumento, respectivamente. Em (ii) um exemplo de avaliação da função

3.8 Testes Unitários

Os componentes que foram desenvolvidos possuem uma ampla possibilidade de entradas, sendo que durante o desenvolvimento apenas algumas são utilizadas como exemplo, e esses exemplos sempre são usados para testar o código como todo. Testes Unitários são uma forma de escrever testes para apenas partes isoladas do código. O Scanner por mais que faca parte da solução por completo, seus métodos podem ser testados isoladamente, verificando que consegue tratar as situações especificas que podem ocorrer durante a tokenização, sem envolver outras partes do código. Indo além com o conceito de testes unitários, Beck (2002) no seu trabalho em Extreme Programming descreveu o que seria conhecido como Test Driven Development e no seu artigo mais recente (Beck, 2023) ele descreve o TDD em três passos:

i Listar todos os novos comportamentos esperados do sistema

ii Escrever os testes a partir da lista criada que validam esses comportamentos

iii Alterar o sistemar até que os testes passem

Aplicando essa disciplina garante que a implementação atenda os requisitos mínimos, pois os mesmos estarão definidos como testes unitário se os passos (i) e (ii) foram realizados. No projeto para cada divisão da solução (Scanner, Parser, AST) foram criados testes para garantir que o resultado esperado seja alcançado, alterando o código até que o mesmo funcione da forma esperada.

4 PROVA DE CONCEITO

Neste capitulo será apresentado uma Prova de Conceito das contribuições cujas implementações foram discutidas no Capítulo 3, com o intuito de demonstrar o uso da abordagem desenvolvida e sua relevância perante os serviços disponibilizados pela Synchro.

A complexidade final da implementação ficou centralizada em três componentes, o Scanner, o Parser e as classes da AST. O Scanner é um código simples que pode ser desenvolvido rapidamente que atende os casos encontrados dos tipos básicos de dados como números e strings, operadores aritméticos e de comparação e os delimitadores como os parênteses e as vírgulas. O Parser tem como centro o método de parse de expressões, sendo implementação de um algoritmo Top-Down semelhante ao Precedence Climbing, que é capaz de lidar com diversos cenários, apenas expandindo a quantidade de operadores e os métodos que os tratam. O código final pode ser compacto, de fácil entendimento, sendo composto do método recursivo e um laço.

4.1 Scanner

A tokenização é o primeiro passo da análise, sendo responsável em converter o texto de entrada em uma sequência de tokens, um token sendo uma string (sequencia de caracteres) que possui um significado próprio. Os tokens foram classificados em (i) operadores, (ii) valores ou identificadores e (iii) separadores como parenteses e virgula.

Tabela 2: Classificação dos tokens léxicos

Token	Classe
`not`	Operador "Prefixo"
`or`, `and`, `gt`, `ge`, `lt`, `le`, `eq`, `ne`, `in`	Operador "Infixo"
Number, String, `null`	Valores Literais
"("ou LPARAN, ")"ou RPAREN, ","ou COMMA	Separadores
Identifier	Identificador de coluna ou função

O Scanner irá ler caractere por caractere e dividir a entrada em sequências de tokens. Por exemplo, a expressão "`contains(NOME, 'CELULAR') or REVISAO gt 10`"será traduzido para a seguinte sequência:

```
<INDENTIFIER, contains>; <LPAREN>; <INDENTIFIER, NOME>; <COMMA>; <
    STRING, 'CELULAR'>; <OPERATOR, OR>; <INDENTIFIER, REVISAO>; <
    OPERATOR, GT>; <NUMBER, 10>
```

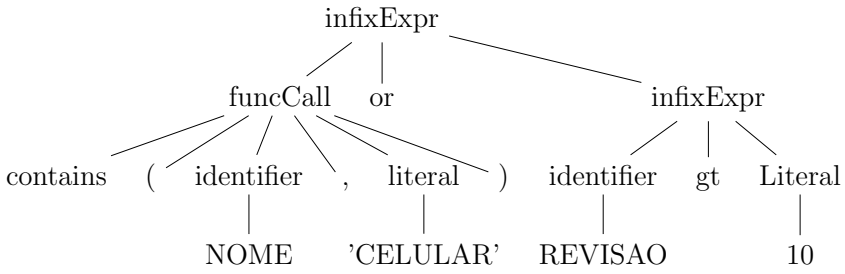


Figura 5: Parse tree da expressão: "contains(NOME, 'CELULAR') or REVISAO gt 10"

4.2 Parser ou Analisador sintático

O parser é responsável por ler a sequência de tokens e análisa-los para gerar uma Árvore Sintática Abstrata (AST). Como foi discutido, o parser é baseado em Precedence Climbing, um método de análise Sintática Top-Down que utiliza a precedência dos operadores como guia para a análise.

O parser foi implementado em uma classe Java que contém o contexto necessário para o parsing, que seria a sequência de tokens e o token atual. Um método chamado de `parseExpression` é responsável pela análise de expressões. Utilizando exemplo anterior, o parser terá a árvore sintática exemplificada na Figura 5. Porém, a saída é um nó da Árvore Sintática Abstrata, que é um conjunto de classes Java para cada tipo de sub-expressao.

Durante a construção da AST, o parser verifica se a sintaxe esta correta e caso de erros uma exceção é lançada. A exceção varia conforme o erro encontrado. Alguns exemplos de erro são o de falta de operador esperado, falta de operando para operadores infixos, falta de parênteses fechando uma expressão ou fechando uma chamada de função.

4.3 Árvore Sintática Abstrata

A Árvore Sintática Abstrata ou AST é um conjunto de classes Java, elas são:

PrefixExpression nó de expressões prefixas, contem o operador e um operando

InfixExpression nó de expressões infixas, contem o operador e dois operandos

ValueExpression nó de valores literais

FieldExpression nó de identificadores de colunas

FunctionCallExpression nó de chamada de função

Todos os nós implementam a interface `FilterExpression` que possui um método principal `eval`. O método `eval` é o método de avaliação que tem como argumento a classe

Java `FilterEvalContext`, o contexto de avaliação e seu papel é o manter as informações intermediarias que serão utilizadas no processo de avaliação. O contexto possui dois membros importantes: (i) dicionário das colunas da tabela, que por si contém os tipos das colunas, (ii) dicionário dos valores literais a serem passados para a biblioteca Spring JDBC. O resultado da avaliação é uma tupla que possui o SQL traduzido e o tipo daquela expressão.

Algo possível com esse esquema é a implementação de regras através da AST. Um simples exemplo é de obrigar que uma certa coluna esteja presente na AST podendo ser alcançado com uma simples verificação no contexto de avaliação. Outra é limitar quais argumentos estão sendo usado para as funções ou operadores, servindo para garantir que índices da tabela sejam utilizados.

Como as classes AST implementam os métodos de verificação de tipos e a avaliação para expressões prefixas e infixas, adicionar um novo tratamento para um novo operador basta a alteração ou adição de novo nó na AST. Para expandir a linguagem com novos operadores o desenvolvedor inclui o novo token no Scanner, caso seja um operador prefixo adiciona um novo tipo no enumerador dos tipos de operador prefixos, caso seja infixo o mesmo se aplica. Adicionar uma nova função envolve a implementação da interface de funções, que indica o tipo de retorno, a quantidade e os tipos dos argumentos e por último o corpo dela, onde os argumentos serão substituídos na hora da avaliação.

O resultado da avaliação é uma tupla representada no *record* `FilterEvalResult`, composto pelo SQL gerado e o tipo da expressão. Os tipos são verificados na avaliação de operador ou função, caso o tipo não seja compatível uma exceção é lançada. Como um exemplo da verificação de tipo, os operadores de comparação validam se os operandos são comparáveis, para isso as expressões dos operandos são avaliados e após isso é verificado se a condição é verdadeira, neste caso sendo que as expressões devem ter o mesmo tipo. O dicionário é usado na verificação dos tipos das expressões que são valores de colunas.

A sequência de substituições da AST na Figura 5 segue:

i identifier('NOME') → `NOME`

ii literal('CELULAR') → `:value_1`

iii funcCall(contains) → `NOME like ('\%' || :value_1 || '\%')`

iv identifier('REVISAO')→ `REVISAO`

v infixExpr('gt') → `REVISAO gt :value_2`

vi infixExpr('or') → `NOME like ('\%' || :value_1 || '\%') or REVISAO gt :value_2`

4.4 API e a Consulta ao Banco

Foi criado apenas um endpoint de consulta que tanto serve para a busca dos metadados de um *dataset* e a própria consulta dos dados. O endpoint de consulta foi baseado no OData, utilizando parâmetros na requisição HTTP que são as opções da consulta.

A consulta de metadados utiliza de uma opção, a `$metadata` que quando é usada possui prioridade, anulando as outras opções. O endpoint de consulta dos dados possui algumas outras opções como:

i `$inlinecount` disponibilizar a quantidade de registros retornados

ii `$skip` e `$top` utilizados para a paginação

iii `$orderBy` para ordenação do resultado

Continuando o exemplo, a SQL resultante será utilizada na requisição dos dados, e para isso outros componentes da consulta são inseridos. A Query SQL é composta por três relevantes, a projeção ou as colunas selecionadas, a tabela e a clausula `WHERE`. A projeção é criada a partir da opção `$select` e por padrão ela possui o valor "*"que significa todas as colunas da tabela, para fins didáticos será assumido que o cliente quer a apenas as colunas `NOME` e `DESCRICAO`, com isto e o nome da tabela temos a primeira parte da requisição:

```
SELECT
    NOME, DESCRICAO
FROM DS_MERCADORIA;
```

Antes disso, na construção da consulta é verificado se a colunas selecionadas estão presentes na tabela desejada, feita de forma não dissimilar de conforme feito na avaliação de expressões de valores de colunas e utilizado os mesmos metadados resgatados do banco. A clausula `WHERE` é apenas o último nível de substituição.

```
SELECT
    NOME, DESCRICAO
FROM DS_MERCADORIA
WHERE
    NOME like ('\%' || :value_1 || '\%')
    or REVISAO gt :value_2;
```

Por último, as opções de paginação são inseridas, usando o padrão de paginação para o banco de dado Oracle recomendada.

```
SELECT
    NOME, DESCRICAO
FROM DS_MERCADORIA
```

```
WHERE
    NOME like ('\%' || :value_1 || '\%')
    or REVISAO gt :value_2
OFFSET 0 ROWS
FETCH FIRST 1000 ROWS;
```

Onde $skip = 0$ e $top = 1000$. No exemplo como foi não foi colocado as opções de paginação, os valores padrões são assumidos. O exemplo de resultado da consulta segue:

```
{
    "link": "http://service-host/dataset/DS_MERCADORIA?$select=NOME
       %2C+REVISAO&filter=contains(NOME, 'CELULAR') or REVISAO gt
       10&$top=1000&$skip=1000"
    "value": [
        {
            "REVISAO": 13,
            "NOME": "CELULAR BRAND A1"
        },
        {
            "REVISAO": 11,
            "NOME": "CELULAR BRAND A2"
        },
        {
            "REVISAO": 10,
            "NOME": "CAPA PARA CELUAR A1 e A2"
        },
        ...
    ]
}
```

A saída consiste em um objeto JSON que possui alguns campos: (i) link, gerado a partir dos valores dados de skip e top indicando uma nova pagina a ser consultada, (ii) inlinetop, a quantidade de registros retornados e por último value, a lista de registros consultados.

5 CONCLUSÕES E TRABALHOS FUTUROS

O resultado do desenvolvimento foi uma implementação simples, porem robusta da linguagem de consultas, sendo possível a fácil expansão da mesma e como a complexidade está concentrada em alguns pontos, a manutenção também se torna fácil.

O algoritmo implementado, o Precedence Climbing, por mais que seja flexível a sua generalização, o Pratt Parsing pode ser uma futura evolução. As classes da AST como implementadas são simples e poucas, porem uma adoção mais profunda dos padrões POO pode ser beneficiosa, removendo a necessidade de utilizar argumentos para distinguir os tipos de operações prefixas e infixas.

REFERÊNCIAS

[1] NETO, Celso de Barros Correia. Sistema Tributário Nacional - Texto base da Consultoria Legislativa. 2019. Disponível em: <https://www2.camara.leg.br/atividade-legislativa/estudos-e-notas-tecnicas/fiquePorDentro/temas/sistema-tributario-nacional-jun-2019/texto-base-da-consultoria-legislativa>. Acesso em: 09 mai. 2024.

[2] Synchro Soluções Fiscais. Disponível em: <https://www.synchro.com.br/empresa/>. Acesso em: 09 mai. 2024.

[3] KILLHAM, Evans. What Does SaaS Mean?. 2020. Disponível em: <https://www.lifewire.com/what-is-saas-5069831>. Acesso em: 17 mai. 2024.

[4] REESE, Will. Nginx: the high-performance web server and reverse proxy. **Linux Journal**, v. 2008, n. 173, p. 2, 2008.

[5] FIELDING, Roy Thomas. **REST**: architectural styles and the design of network-based software architectures. Doctoral dissertation, University of California, 2000

[6] FOWLER, Martin. Microservices. 2014. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 17 mai. 2024.

[7] What is containerization?. 2021. Red Hat. Disponível em: <https://www.redhat.com/en/topics/cloud-native-apps/what-is-containerization>. Acesso em: 20 mai. 2024.

[8] What is container orchestration?. 2022. Red Hat. Disponível em: <https://www.redhat.com/en/topics/containers/what-is-container-orchestration>. Acesso em: 20 mai. 2024.

[9] FOROUZAN, Behrouz A. **TCP/IP protocol suite**. McGraw-Hill Higher Education, 2002.

[10] BERNERS-LEE, Tim; FIELDING, Roy; FRYSTYK, Henrik. Hypertext transfer protocol–HTTP/1.0. 1996.

[11] BRAY, Tim. The javascript object notation (json) data interchange format. 2014.

[12] CODD, Edgar F. A relational model of data for large shared data banks. **Communications of the ACM**, v. 13, n. 6, p. 377-387, 1970.

[13] CHAMBERLIN, Donald D.; BOYCE, Raymond F. SEQUEL: A structured English query language. In: **Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control**. 1974. p. 249-264.

[14] BYRON, Lee. GraphQL: A data query language. Meta. 2015. Disponível em: <https://engineering.fb.com/2015/09/14/core-infra/graphql-a-data-query-language/>. Acesso em: 17 mai. 2024.

[15] BERNERS-LEE, Timothy J. Information management: A proposal. 1989.

[16] FOWLER, Martin. **Domain-specific languages**. Pearson Education, 2010.

[17] COOPER, Keith D.; TORCZON, Linda. **Engineering a compiler**. Morgan Kaufmann, 2022.

[18] ANTLR. Disponível em: <https://www.antlr.org/>. Acesso em: 11 mai. 2024.

[19] GNU Bison. Disponível em: <https://www.gnu.org/software/bison/>. Acesso em: 11 mai. 2024.

[20] DIJKSTRA, Edsger W. **Algol 60 translation: An Algol 60 translator for the x1 and Making a translator for Algol 60**. Stichting Mathematisch Centrum. Rekenafdeling, 1961.

[21] RICHARDS, Martin; WHITBY-STREVENS, Colin. **BCPL: the language and its compiler**. Cambridge University Press, 1981.

[22] PRATT, Vaughan R. Top down operator precedence. In: **Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages**. 1973. p. 41-51.

[23] CHU, Andy. Pratt Parsing and Precedence Climbing Are the Same Algorithm. 2016. Disponível em: <https://www.oilshell.org/blog/2016/11/01.html>. Acesso em: 17 mai. 2024.

[24] BATISTA, Cesar et al. Towards a Multi-Tenant Microservice Architecture: An Industrial Experience. In: **2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)**. IEEE, 2022. p. 553-562.

[25] HARDT, Dick. **The OAuth 2.0 authorization framework**. 2012.

[26] How OpenID Connect Works. OpenID Foundation Disponível em: <https://openid.net/developers/how-connect-works/>. Acesso em: 20 mai. 2024.

[27] OData. Disponível em: <https://www.odata.org/>

[28] OASIS Approves OData 4.0 Standards for an Open, Programmable Web. OASIS, 2014. Disponivel em: <https://www.oasis-open.org/news/pr/oasis-approves-odata-4-0-standards-for-an-open-programmable-web/>. Acesso em: 20 mai. 2024.

[29] Cesar et al. Managing asynchronous workloads in a multi-tenant microservice enterprise environment. **Software: Practice and Experience**, v. 54, n. 2, p. 334-359, 2024.

[30] BECK, Kent. **Extreme Programming**. Addison Wesley, 2002.

[31] BECK, Kent. Canon TDD. 2023. Disponível em: <https://tidyfirst.substack.com/p/canon-tdd>. Acesso em: 20 mai. 2024.