

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

# **Ferramenta de apoio ao ensino de SQL para iniciantes através de linguagem natural**

**Daniel Blando Deluiggi**

**PROJETO FINAL DE GRADUAÇÃO**

**CENTRO TÉCNICO CIENTÍFICO - CTC**

**DEPARTAMENTO DE INFORMÁTICA**

**Curso de Graduação em Ciência da Computação**

Rio de Janeiro, Julho de 2013



**Daniel Blando Deluiggi**

# **Ferramenta de apoio ao ensino de SQL para iniciantes através de linguagem natural**

Relatório de Projeto Final, apresentado ao programa  
**Ciência da Computação** da PUC-Rio como requisito  
parcial para a obtenção do título de Bacharel em Ciência  
da Computação.

Orientador: Rogério Costa

Rio de Janeiro  
Julho de 2013

## Sumário

<b>1. Introdução</b>	<b>5</b>
<b>2. Estado da Arte</b>	<b>6</b>
2.1 Delver [3]	6
2.2 Microsoft English Query [4]	6
2.3 PrimeQue [5]	8
<b>3.Proposta e Objetivos do trabalho</b>	<b>9</b>
<b>4.Plano de Ação</b>	<b>11</b>
4.1 Projeto Final 1	11
4.2 Projeto Final 2	13
<b>5.Estudos</b>	<b>14</b>
5.1 Estudo sobre SQL	14
5.1.1 DML	15
5.1.2 Cláusulas	16
5.1.3 Funções	17
5.1.4 Condições	19
5.2 Estudo sobre compiladores	20
5.2.1 DML	20
5.2.2 Cláusulas	21
5.2.3 Funções	23
5.2.4 Condições	24
5.2.5 Extras	25
5.3 Estudo sobre funcionalidades	26
5.3.1 Auto-complete	26
5.3.2 Keep learning	26
5.3.3 Improve youserlf	27
5.4 Estudo sobre NLP	27
5.4.1 Tokenizers	28

<b>5.4.2 Stemmer</b>	<b>29</b>
<b>5.4.3 Post-tagged</b>	<b>30</b>
<b>5.4.4 String distance</b>	<b>31</b>
<b>6.Aplicação</b>	<b>33</b>
<b>7. Referências bibliográficas</b>	<b>37</b>

# 1. Introdução

O mercado atual para ferramentas de ensino está em alta. Os alunos buscam cada vez mais um aprendizado independente e acelerado como alternativas do ensino tradicional, focado, exclusivamente em salas de aula. Iniciativas como *Coursera*[1] e *Codeschool*[2] têm propostas de ensinar o usuário de maneiras diferentes, mas com o mesmo objetivo. O *Coursera* é um site focado em aulas universitárias e com um tutor para ajudar ao longo do curso. Já o *Codeschool* tem uma iniciativa inovadora que vem fazendo muito sucesso: o site possui inúmeros tutoriais onde o aluno completa etapas que incluem uma explicação teórica e um exercício prático.

Entretanto, essas iniciativas de ensino têm uma grande escassez de ferramentas para aprendizado de banco de dados. O *Coursera*, por exemplo, fornece cursos voltados a banco de dados, mas sem nenhuma ferramenta disponível para um aprendizado prático. Os alunos encontram grande dificuldade ao procurar uma ferramenta que os auxilie de forma mais interativa.

Assim, o objetivo desse trabalho é desenvolver uma ferramenta para ajudar no ensino em banco de dados, de uma forma simples, porém eficaz e interativa. Sua função será auxiliar os alunos no aprendizado de SQL relacionando-o à linguagem natural. Essa ferramenta será instalada no computador com alguns bancos de dados de exemplo, mas permite que o aluno a conecte a seu próprio banco de dados. Terá como funcionalidade receber uma pergunta feita em linguagem natural dentro de uma sintaxe pré-definida. A partir dessa pergunta e das informações extraídas do banco de dados, ela irá gerar o comando SQL, que terá como resultado as informações do próprio banco para tal pergunta. Vale ressaltar que esse fluxo deve ocorrer sem a necessidade de nenhuma informação semântica.

Na ferramenta conhecida como *PrimeQue* foram encontradas as mesmas características desse projeto, onde o sistema consegue guiar o usuário a elaborar perguntas para receber em forma gráfica a resposta no *dashboard*, porém o usuário inicialmente deve informar conceitos semânticos sobre os dados que estão no banco de dados. Outro diferencial é que o *PrimeQue* tem como foco o mercado, o que não se aplica ao intuito da ferramenta proposta.

## 2. Estado da Arte

As propostas existentes que auxiliam o ensino de banco de dados seguem caminhos diversos. Algumas das abordagens encontradas são ferramentas de gerenciamento de banco de dados, cursos on-line, sites com bancos disponíveis para manipulação on-line e tutorias sobre SQL. Todos esses métodos podem ser utilizados para aprendizagem, mas eles não se encaixam na proposta de projeto de ensinar SQL de uma forma mais intuitiva, onde o aluno pode relacionar a linguagem natural com o SQL.

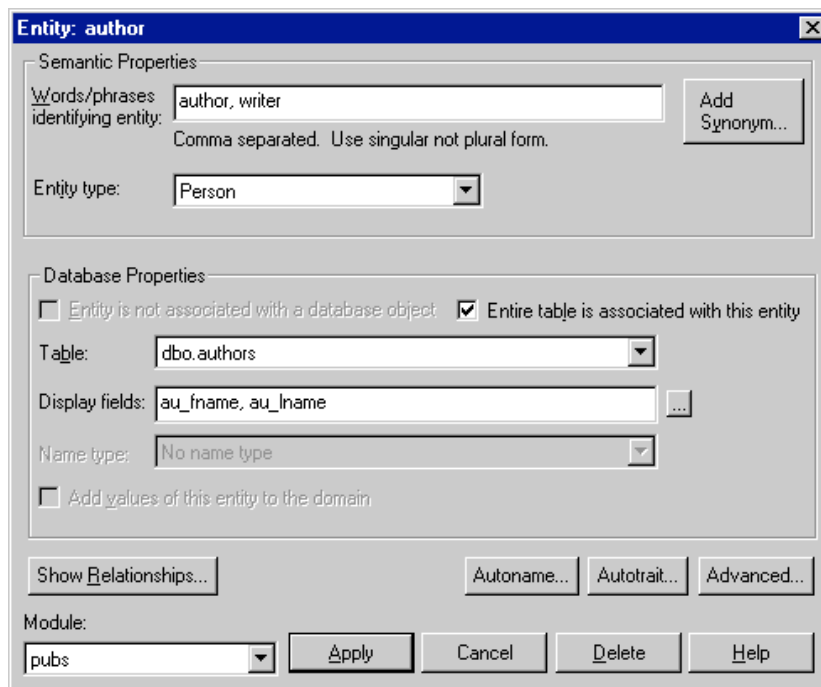
Seguindo o método proposto não foi encontrada nenhuma ferramenta em funcionamento com o intuito de auxiliar o ensino de banco de dados aos alunos. Apresentaremos, abaixo, algumas ferramentas semelhantes, mas que não tem a mesma proposta:

### 2.1 *Delver* [3]

Esta ferramenta tem uma grande semelhança ao proposto nesse projeto, pois traduz perguntas em linguagem natural para SQL, e pode ser usada a partir de API ou SDK. Entretanto, ela ainda não foi lançada e suas únicas informações são teóricas e encontradas em seu site. Assim não temos como comprovar de que forma será seu funcionamento.

### 2.2 Microsoft English Query [4]

Ferramenta utilizada para tradução de linguagem natural em SQL, mas por meio de mapeamento prévio da linguagem natural. O usuário deve inicialmente definir palavras chaves e regras de sintaxe no programa para que o mesmo possa traduzir a pergunta e responder em SQL. As palavras chaves são entidades que podem se relacionar, e as regras são verbos que relacionam as entidades. A figura 1 apresenta como é criada uma palavra chave no sistema. Neste imagem pode-se notar a criação da ligação entre tabelas e colunas do banco com palavras chaves. Já na figura 2 existe um exemplo de ligação de duas palavras chaves por meio de um verbo. Esse método de tradução cria uma dificuldade enorme para o usuário, pois necessita que o mesmo introduza regras para cada sintaxe diferente da realizada no banco.



**Entity: author**

**Semantic Properties**

Words/phrases identifying entity:  Add Synonym...

Entity type:

**Database Properties**

☐ Entity is not associated with a database object ☒ Entire table is associated with this entity

Table:

Display fields:  ...

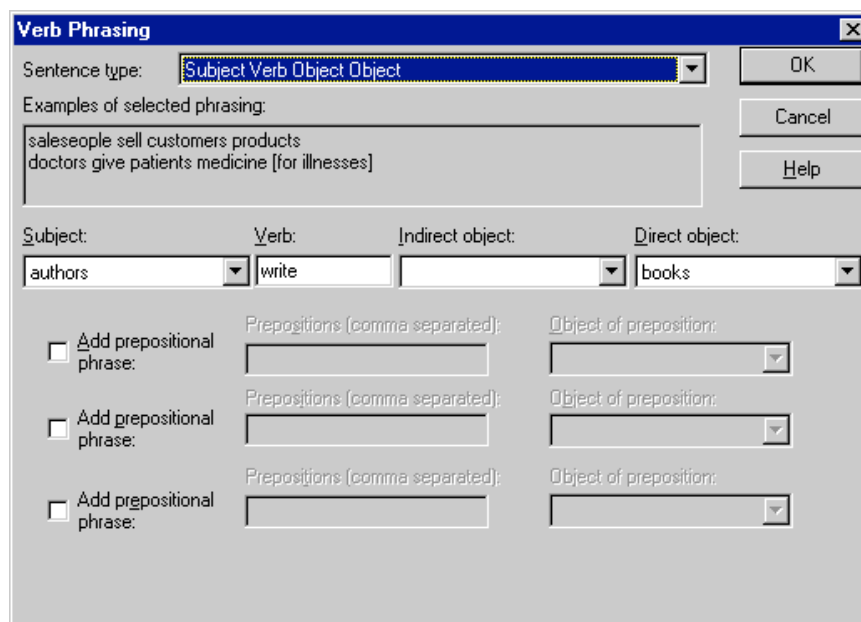
Name type:

☐ Add values of this entity to the domain

Show Relationships... Autoname... Autotrait... Advanced...

Module:  Apply Cancel Delete Help

Figura 1: Tela para criar entidades. O usuário cria palavras chaves que se relacionam com tabelas e colunas existentes no banco



**Verb Phrasing**

Sentence type:  OK

Examples of selected phrasing:

salespeople sell customers products  
doctors give patients medicine [for illnesses]

Cancel Help

Subject:	Verb:	Indirect object:	Direct object:
<input type="text" value="authors"/>	<input type="text" value="write"/>	<input type="text"/>	<input type="text" value="books"/>

☐ Add prepositional phrase: Prepositions (comma separated):  Object of preposition:

☐ Add prepositional phrase: Prepositions (comma separated):  Object of preposition:

☐ Add prepositional phrase: Prepositions (comma separated):  Object of preposition:

Figura 2: Tela para criação de regras. Com as regras o usuário pode relacionar entidades criadas previamente.

### 2.3 PrimeQue [5]

É a ferramenta que mais se aproxima da proposta desse projeto, pois ajuda o usuário na formulação de perguntas para um sistema específico com uma interface limpa e simples. A figura 3 exibe um exemplo das opções fornecidas pelo sistema para a próxima palavra que pode ser usada pelo usuário.

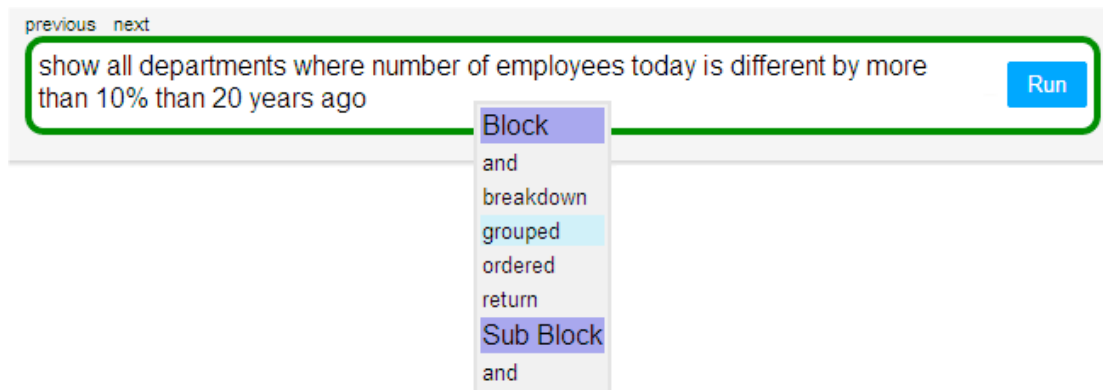


Figura 3: Formulação da pergunta pelo usuário na ferramenta PrimeQue

O problema é que ela também necessita que o usuário insira informação semântica sobre a base de dados. O usuário pode criar sua própria linguagem definindo um mapa semântico com nomes para campos, conexões entre os campos, entre outros. A figura 4 mostra como pode ser criado um mapa semântico para a tabela *employees*.

Builder - HR System (Dev)

Tables ▾

Table: employees ▾ Table Data Refresh Data

Main Topic: ☒

Default Meaning: <Choose Default> ▾

Topic Name	Singular	Plural	Type	Where
	employee	employees	person ▾	
Add more				

Table Fields	Name	Modified Name	Type	Key	Empty	Hidden	Retu
	first_name	first name	String ▾	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	last_name	last name	String ▾	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	gender		Null ▾	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	birth_date	birth date	Date ▾	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	hire_date	hire date	Date ▾	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Table Extra: ☐ History, Verbs

Help Preview

Figura 4: Construção da semântica na ferramenta PrimeQue



A *PrimeQue* também possui diferentes funcionalidades como *dashboard*, histórico de perguntas e diferentes níveis de usuários. O *dashboard* pode ser personalizado com gráficos e estatísticas sobre o banco, bem como um histórico de perguntas para análise posterior e reutilização. Os usuários podem ser: *PrimeQue Admin*, *DB Admin* e *User*. O usuário *PrimeQue Admin* tem privilégio de criar conexão com novos bancos e novos usuários. O usuário *DB Admin* é o responsável por fornecer as informações semânticas sobre o banco, e *User* é o usuário padrão de um banco de dados que pode consultar os dados e ter seu próprio *dashboard*.

Entretanto, o maior problema é que a *PrimeQue*, infelizmente, é voltada para o mercado e não possui um âmbito educativo.

Finalizando, concluímos que não foi possível encontrar uma ferramenta de auxílio ao aprendizado de banco de dados que consiga, a partir de uma pergunta em linguagem natural e informações extraídas do banco, responder com uma consulta SQL. Todas as ferramentas existentes atualmente necessitam de uma predefinição sintática sobre o banco.

### **3.Proposta e Objetivos do trabalho**

O projeto se baseia no desenvolvimento de uma ferramenta gratuita e de código-aberto para auxiliar no ensino de banco de dados, trazendo uma nova solução para os alunos aprenderem de forma mais simples e intuitiva a linguagem SQL. Essa ferramenta tem o propósito de apoiar alunos com pouco ou nenhum conhecimento de SQL.

Seu principal objetivo será receber uma pergunta em linguagem natural e retornar o equivalente em SQL para que o aluno possa avaliar e executar o comando, com o intuito de que ele entenda a ligação entre sua pergunta e as funções da linguagem SQL. Seu requisito mínimo será um banco de dados, e o aluno poderá optar pelo disponibilizado na ferramenta ou por conectá-la ao seu próprio banco de dados, conforme representado na figura 5.

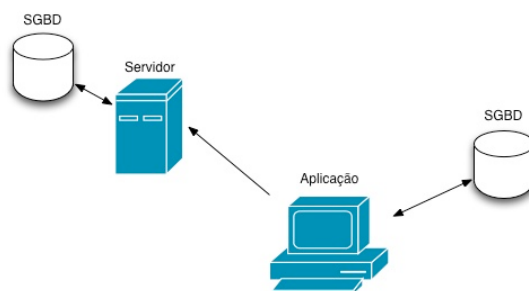


Figura 5: Diagrama da arquitetura

Ao iniciar, o programa recolherá dados do banco como, por exemplo, nomes de tabelas, colunas e restrições, de forma a poder identificar palavras-chaves nas sentenças informadas pelos usuários. Como entrada, o programa irá receber uma pergunta feita pelo aluno com base no banco de dados fornecido e também irá guiar o usuário na criação dessa pergunta. O guia funcionará de forma a restringir as opções de palavras que o aluno pode usar com relação às funcionalidades existentes no programa, conforme representado na Figura 6. A cada comando de SQL implementado, a lista de palavras e opções de frases aumentará.

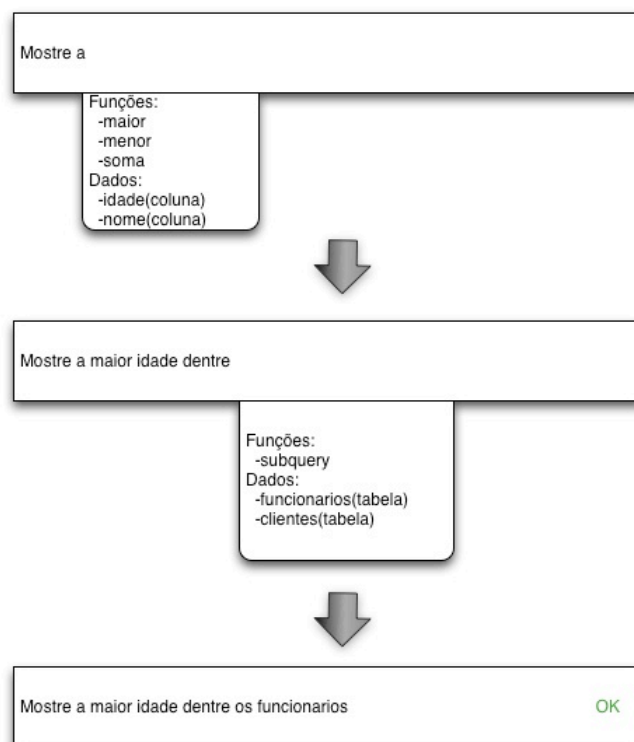


Figura 6: Exemplo do guia para formulação de perguntas pela ferramenta

Será também estudada uma forma de outros desenvolvedores implementarem novos comandos a partir de suas necessidades. De saída o aluno irá

receber o SQL referente à sua pergunta e o resultado que o mesmo obteve no banco, vide figura 7.

SELECT max(t1.idade) FROM funcionarios t1	
	idade
1	57

Figura 7: Exemplo de um resultado a partir de um pergunta no programa

## 4.Plano de Ação

### 4.1 Projeto Final 1

O projeto final 1 é focado no estudo sobre os componentes que farão parte do sistema, conforme representado no cronograma da Figura 8. Esse cronograma foi dividido em quatro fases: (i) estudo sobre SQL, (ii) estudo sobre compiladores, (iii) estudo sobre *natural language processing* (NLP), e (iv) definição do sistema. Segue uma explicação para fase do cronograma.

Primeiramente, serão aprofundados os estudos dos comandos SQL para se criar um escopo do que será implementado pela ferramenta. A princípio os principais comandos de DML (*data manipulation language*) estão previstos no escopo. Vale lembrar que para cada comando implementado existirá uma ou mais palavras no guia que o representem.

Na segunda fase do estudo, com maior duração será avaliada a forma de implementação da ferramenta, que é um compilador de uma linguagem natural

pré-definida para SQL. Nessa mesma fase será dado foco ao estudo de como pode ser garantido um funcionamento correto do sistema.

Para finalizar os estudos, foi adicionado o conhecimento sobre *natural language processing* (NLP). O intuito dessa fase é definir a liberdade que o aluno terá para elaborar suas perguntas. O programa terá um grande desafio para compreender a semântica das palavras, sinônimos entre elas e até suas ambiguidades. Esse estudo será muito importante para definir uma sintaxe que seja compatível e capaz de ser implementada na ferramenta.

Com os estudos anteriores concluídos, conforme representado na Figura 9, já pode ser iniciada a etapa de definição do sistema, onde o objetivo é concretizar a arquitetura do mesmo, definir os seus componentes e especificar os requisitos.

TAREFA	Março	Abril	Maio	Junho
Elaboração da Proposta				
Estudo sobre SQL				
Estudo sobre Compiladores				
Estudo sobre NLP				
Definição do Sistema				
Elaboração do Relatório Final				

Figura 8: Plano de ação para o projeto final 1

TAREFA	Março	Abril	Maio	Junho
Elaboração da Proposta				
Estudo sobre SQL				
Estudo sobre Compiladores				
Estudo sobre NLP				
Definição do Sistema				
Elaboração do Relatório Final				

Figura 9: Executado ao final do projeto final 1

## 4.2 Projeto Final 2

O projeto final 2 será dividido por duas etapas principais, a fase de projeto e a de implementação do sistema, conforme representado na Figura 9.

A primeira fase consiste na especificação da ferramenta. Essencialmente, serão produzidos casos de uso, casos de teste, e diagramas de sequência.

Na próxima fase será iniciada a implementação da ferramenta, que terá início pelo compilador, com as seguintes etapas: léxica, sintática, semântica e geração de código. A ideia inicial é implementar o léxica e sintática de forma bem definida, com frases simples e poucas possibilidades de palavras. A partir desse ponto será iniciada uma evolução lenta das alternativas de escrita que a ferramenta estará apta a compreender. Essa evolução tem por base o conhecimento adquirido sobre NLP no projeto final 1.

A interface de entrada e saída será feita após o funcionamento do compilador, pois se necessita dele para que a interface funcione como desejado.

As funcionalidades, como guia, tabela do resultado e expansão da ferramenta por desenvolvedores, serão desenvolvidas no final e com baixa prioridade. O escopo de funcionalidades do sistema pode aumentar dependendo do tempo restante.

Ao final de todas as etapas, serão iniciados testes para medir o nível de acerto da ferramenta, sendo importante e esperado que ela tenha um alto nível de acerto por se tratar de uma ferramenta voltada para o ensino.

Para concluir será realizada uma apresentação de todo o projeto consolidado e uma demonstração prática da ferramenta.

TAREFAS	Julho	Agosto	Setembro	Outubro	Novembro
Especificação do Sistema					
Implementação do Compilador					
Implementação do Interface					
Implementação das Funcionalidades					
Realização de testes					
Elaboração do Relatório Final					

Figura 10: Plano de ação para o projeto final 2

## 5. Estudos

Na seção 5.1, onde o assunto majoritário é SQL, temos uma explicação de como funcionara a parte léxica do programa. Na seção 5.2, será explicado a análise sintática do programa. Nas seções 5.3 e 5.4, serão estudadas melhorias para a funcionalidade do sistema.

A análise léxica [6] é a forma de reconhecer o alfabeto definido para sua linguagem. Nessa análise verifica-se os caracteres de entrada com regras pré-definidas para descobrir a que *token* (palavras chaves conhecidas pelo programa) aquele grupo de caracteres corresponde.

A análise sintática [7] se utiliza do resultado da análise léxica para identificar uma sequência de entrada e mapear as mesma em um dos comandos reconhecidos pelo sistema. Com o estudo sintático da sequência de entrada pode-se criar uma árvore de *tokens* (códigos reconhecido pelo sistema), que vai gerar o resultado final.

A seção 5.3, estudo sobre funcionalidades, contém idéias a serem implementadas para melhorar a performance e uso do sistema pelo usuário. Nela serão explicadas três funcionalidades referentes a ajuda de criação da pergunta pelo usuário e melhoria no aprendizado de palavras e comandos pelo sistema.

O estudo sobre NLP, seção 5.4, explicará funcionalidades de NLP que serão úteis para o sistema ao interligar palavras a *tokens*. Essas funcionalidades são responsáveis por reconhecer semelhança entre palavras, descobrir todas as palavras de uma frase, entre outros.

### 5.1 Estudo sobre SQL

O desenvolvimento do projeto tem como foco o reconhecimento de comandos SQL principalmente comandos DML (*Data Manipulation Language*). Como o próprio nome indica, são comando voltados a manipulação de dados. Os comandos DML englobados são: *SELECT*, *INSERT*, *UPDATE*, *DELETE*. Esses comandos foram escolhidos por ter uma maior relevância no aprendizado de SQL para alunos iniciantes como pode ser notado em sala de aulas e livros de banco de dados.

Para a tradução dessas palavras foram usadas duas técnicas, a primeira diz respeito sobre a tradução literária da palavra entre inglês e português, já a segunda sobre os sinônimos dessas palavras. Na primeira fase, a palavra original em inglês foi procurado no dicionário de traduções on-line da Michaelis [8]. Dele foram retiradas as três traduções

mais coerentes com o sentido da palavra original. Na segunda fase foram procurados os sinônimos das três palavras encontradas na primeira fase, para essa etapa foi usada uma ferramenta [9] on-line de sinônimos. Esses sinônimos ainda foram submetidos novamente à fase dois. Essa fase foi repetida por três vezes como é ilustrado a seguir.

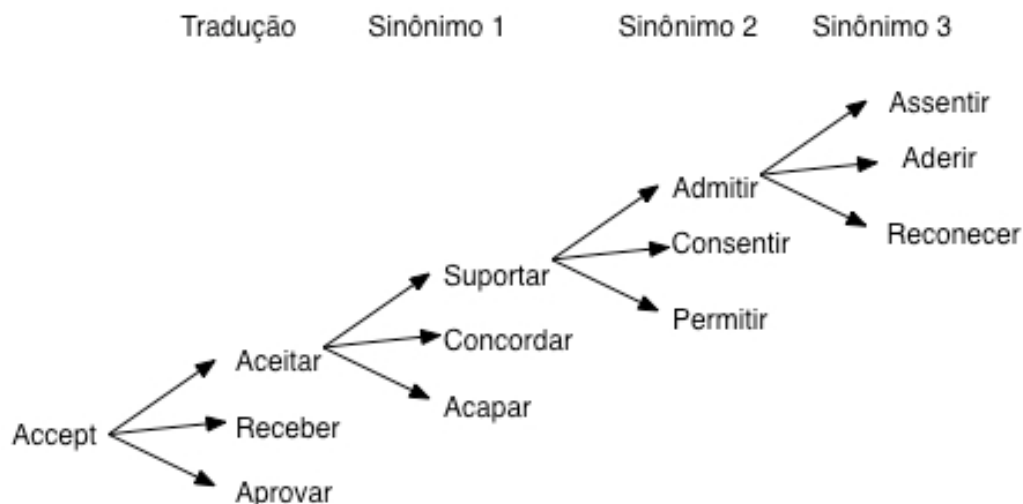


Figura 11: Exemplo da tradução da palavra *accept*

#### 5.1.1 DML

A tabela 1 ilustra os comandos DML e suas possíveis traduções aceitas pelo programa.

DML	Tradução	Sinônimos	Exemplos - Linguagem
select	selecionar, escolher	apurar, definir, destacar, assinalar, determinar, estabelecer, indicar, especificar, mostrar, citar	Mostre a idade da pessoa com o nome igual a João
insert	inserir, introduzir, intercalar	colocar, implantar, fixar, adicionar, agregar, anexar, incorporar, juntar, unir, criar, pôr,	Adicione a pessoa com o nome João, com idade de 20 anos e email igual a joao@joao.com

update	atualizar, modernizar	mudar ,alterar, renovar	Atualize o email da pessoa com nome João para <a href="mailto:joao@alberto.com">joao@alberto.com</a>
delete	apagar, anular, cancelar	extinguir, dissipar, tirar, limpar, retirar	Retire a pessoa com nome de joao

Tabela 1: Comandos DML suportados pelo sistema

Na construção de comandos DML também são utilizadas outras palavras para filtrar e melhorar a sua pesquisa. Um ponto positivo da linguagem SQL é a manipulação de seus comandos para gerar diversos resultados. Na criação desses comandos DML podem ser usados diversos subcomandos SQL, que organizamos em três grupos: cláusulas, funções e condições.

#### 5.1.2 Cláusulas

As cláusulas são palavras utilizadas para se realizar uma busca por dados, algumas dessas palavras são obrigatórias e outras opcionais. Para se iniciar um comando SQL precisa-se usar uma cláusula considerada como obrigatória. Alguns exemplos são os citados acima nos comandos DML. Cláusulas consideradas opcionais são palavras usadas para refinar uma pesquisa. Essas palavras têm como objetivo ajudar o usuário a restringir e organizar as informações que deseja visualizar.

Para esse projeto foram escolhidas algumas cláusulas opcionais que contém grande importância para o ensino de SQL. Na tabela 2 ilustramos as palavras e suas respectivas traduções e sinônimos que serão utilizados nesse projeto.

Cláusulas	Tradução	Sinônimos	Exemplos Linguagem
where	onde, aonde,com	-	Mostre a pessoa aonde o nome é igual a João.
group by	agrupar, associar, combinar	agregar, classificar, ajuntar, juntar, reunir, concatenar, vincular, integrar, anexar	Mostre as pessoas agrupando por idade.



having	ter, haver, possuir	dispor, portar, abranger, apresentar, conter, obter, com	Mostre as pessoas contendo idade maior que 15.
order by	ordenar, dispor, arranjar	arrumar, organizar, classificar, alinhar	Mostre as pessoas ordenadas por nome.
join	ligar, juntar, unir	ajuntar, acoplar, agregar, interligar, reunir, adicionar, anexar, associar, somar, unir	Mostre as pessoas agregando emprego por id com id_pessoa.

Tabela 2: Cláusulas suportadas pelo sistema

Dentre as cláusulas a cima duas precisam de um suporte maior para atuar corretamente com os comandos SQL.

A primeira seria *order by* que, pela linguagem original, pode ter duas variações com as palavras *desc* e *asc* (decrecente e crescente). Neste projeto, vamos definir o uso das palavras “crescente” e “decrecente” para ter o objetivo de suprir essa funcionalidade. Elas devem ser usadas junto com outra palavra que tenha como significado *order by*. Importante mencionar que como em SQL se não for encontrada nenhuma das palavras o padrão considerado será “crescente”.

A segunda clausula para complemento de possibilidades é a clausula *join*. Na linguagem padrão SQL, existem várias formas de uso da clausula *join*, como: *natural join*, *inner join*, *left outer join*, *right outer join*, *full outer join*, ente outros. Nesse projeto focaremos em 3 tipos dessa clausula: *inner join (join)*, *left outer join*, *right outer join*. Ao usar uma das palavras reservadas que signifiquem a clausula *join* será adotado como padrão o método *join*, sinônimo para *inner join*, se o aluno desejar especificar um dos outros tipos da clausula ele poderá usar as palavras reservadas “externa esquerda” e “externa direita” para *left outer join* e *right outer join*, respectivamente.

### 5.1.3 Funções

As funções são métodos para se manipular informações de uma única coluna do banco de dados. As cláusulas definem uma condição, entretanto ao usar funções manipulamos um conjunto definido da mesma informação, como exemplo: nome de pessoa ou data de nascimento.

Para escolher as funções englobadas no projeto procurou-se definir as funções com mais uso e relevância. A seguir, na tabela 3, se encontram as funções consideradas, com traduções e sinônimos, além de exemplos utilizados.

<b>Funções</b>	<b>Tradução</b>	<b>Sinônimos</b>	<b>Exemplos - Linguagem</b>
distinct	distinto, diverso	diferente, dissemelhante, dessemelhante, distinguir, particular, peculiar, heterogêneo	Mostre os nomes distintos de pessoas.
count	contar	o numero de, a quantidade, quantos	Mostre o numero de nomes de pessoas.
avg(average)	média	-	Mostre a idade média de pessoas
min(minimum)	mínimo	menor	Mostre a menor idade entre as pessoas
max(maximum)	máximo	maior	Mostre a maior idade entre as pessoas
sum	soma	total	Mostre o total de nomes das pessoas
first	primeiro	-	Mostre a primeira idade de pessoas
last	último	-	Mostre o ultimo nome entre pessoas
len(length)	comprimento, extensão	tamanho	Mostre o comprimento do nome entre pessoas

Tabela 3: Funções suportadas pelo sistema

#### 5.1.4 Condições

Uma condição é a combinação de uma ou mais expressões e operadores lógicos que retornam como resposta verdadeiro ou falso. Uma condição pode estar relacionada a uma ou mais colunas do banco avaliando se as mesmas estão de acordo com a operação lógica.

As condições representadas na tabela 4 foram classificadas como usuais e de necessidade para esse projeto. Como algumas condições são símbolos, o método de tradução proposto no início do item 3 não pode ser utilizado e em seu lugar foi feita uma tradução literária do significado do símbolo.

Condições	Tradução	Sinônimos	Exemplos - Linguagem
>	maior que	-	Mostre as pessoas com idade maior que 10.
<	menor que	-	Mostre as pessoas com idade menor que 10.
=	igual a	semelhante	Mostre as pessoas com idade igual a 10.
>=	maior ou igual a	-	Mostre as pessoas com idade maior ou igual a 10.
<=	menor ou igual a	-	Mostre as pessoas com idade menor ou igual a 10.
!=	diferente de	distinto de	Mostre as pessoas com idade diferente de 10.
in	dentro	pertencer, entre	Mostre as pessoas com idade entre 10 e 20.
not null	não nulo	existe, não vazio	Mostre as pessoas com nome existente.
null	nulo	não existe, vazio	Mostre as pessoas com nome vazio.

like	parecido	semelhante, contem	Mostre as pessoas com nome semelhante a 'jo'.
------	----------	-----------------------	---

Tabela 4: Condições suportadas pelo sistema

## 5.2 Estudo sobre compiladores

Como resultado do estudo sobre compiladores, será explicado o funcionamento da parte sintática do sistema. Nessa etapa o programa tem o objetivo de identificar um grupo de *tokens* reconhecidos pelo léxico e definir a qual comando existente no sistema essa sequência de *tokens* se refere.

Nos tópicos seguintes será explicado como é definida a gramática de cada comando existente no programa. Com base nas regras da gramática de cada comando o compilador poderá chegar a conclusão sobre qual código deve ser gerado como resposta.

Iremos dividir o estudo como feito na etapa anterior. Todas as regras e informações foram obtidas com base no estudo sobre a documentação publicada pela faculdade *Stanford* [10] e dados no site *w3school* [11] em relação a linguagem SQL.

Legenda:

{table} : Uma tabela do banco.

{table}+: Uma ou mais tabelas do banco.

{table}\*: Zero ou mais tabelas do banco.

{table},{column}: Uma tabela e/ou uma coluna do banco.

### 5.2.1 DML

A seguir iremos apresentar a regra sintática para os comandos DML. Como já citamos anteriormente os comandos são *select*, *update*, *insert*, *delete*.

Comandos: DML

Nome *token*: DML\_START

-SELECT

Nome *token*: CMD\_SELECT

Descrição: Comando usado para selecionar dados do banco.

Sintaxe:

SELECT	{COLUMN}+, {FUNC}+	FROM	{TABLE, CLA_JOIN} +,{CMD_SELECT}	{CLAUSES}* 
--------	-----------------------	------	-------------------------------------	----------------

#### -INSERT

Nome *token*: CMD\_INSERT

Descrição: Comando utilizado para inserir novas informações.

Sintaxe:

INSERT INTO	{TABLE}	VALUES	{VALUE_LIST}
INSERT INTO	{TABLE}	{CMD_SELECT}	

#### -UPDATE

Nome *token*: CMD\_UPDATE

Descrição: Utilizado para atualizar informações.

Sintaxe:

UPDATE	{TABLE}	SET	{COLUMN}	CMD_EQUAL	{VALUE}	{CLAUSES}* 
--------	---------	-----	----------	-----------	---------	----------------

#### -DELETE

Nome *token*: CMD\_DELETE

Descrição: Comando para remoção de informação.

Sintaxe:

DELETE FROM	{TABLE}	{CLAUSES}* 
-------------	---------	----------------

### 5.2.2 Cláusulas

Na etapa b, serão apresentadas as regras sintáticas para todos os comandos do tipo cláusula. Os comandos são *where*, *group by*, *having*, *order by*, *join*.

Comandos: Cláusulas

Nome *token*: CLAUSES

#### -WHERE

Nome *token*: CLA\_WHERE

Descrição: Cláusula usada para filtrar informações.

Sintaxe:

WHERE	{COND}+
-------	---------

#### -GROUP BY

Nome *token*: CLA\_GROUPBY

Descrição: Comando para agrupar informações iguais de uma ou mais colunas.

Sintaxe:

GROUP BY	{COLUMN}+
----------	-----------

#### -HAVING

Nome *token*: CLA\_HAVING

Descrição: Semelhante a cláusula *where* com a possibilidade do uso de funções de agregação.

Sintaxe:

#### -ORDER BY

Nome *token*: CLA\_ORDERBY

Descrição: Utilizado no sentido de ordenar o resultado em relação a uma ou mais colunas.

Sintaxe:

ORDER BY	{COLUMN}+
----------	-----------

#### -JOIN

Nome *token*: CLA\_JOIN

Descrição: Comando utilizado para associar colunas de diferentes tabelas.

Sintaxe:

{TABLE}	JOIN	{TABLE}	ON	{COLUMN}	=	{COLUMN}
---------	------	---------	----	----------	---	----------

### 5.2.3 Funções

Os comandos do tipo função possuem sintaxe semelhante. Nesse caso iremos exibir a sintaxe de todos ao final da apresentação de cada comando.

Comandos: Funções

Nome *token*: FUNC

-COUNT

Nome *token*: FUNC\_COUNT

Descrição: Quantidade de tuplas dentro do conjunto.

-AVG

Nome *token*: FUNC\_AVG

Descrição: Média em valor do conjunto de dados.

- MIN

Nome *token*: FUNC\_MIN

Descrição: Determina o menor valor entre o conjunto de dados.

-MAX

Nome *token*: FUNC\_MAX

Descrição: Determina o maior valor entre o conjunto de dados.

-SUM

Nome *token*: FUNC\_SUM

Descrição: Determina o soma dos valores entre o conjunto de dados.

-FIRST

Nome *token*: FUNC\_FIRST

Descrição: Determina o primeiro valor entre o conjunto de dados.

-LAST

Nome *token*: FUNC\_LAST

Descrição: Determina o ultimo valor entre o conjunto de dados.

-LEN

Nome *token*: FUNC\_LEN

Descrição: Função para determinar o tamanho em caracteres do dado.

Sintaxe:

FUNC	(	{COLUMN}	)
------	---	----------	---

#### 5.2.4 Condições

A seguir iremos apresentar a regra sintática para os comandos do tipo condição. As condições compartilham a mesma sintaxe.

Comandos: Condições

Nome *token*: COND

- <

Nome *token*: COND\_LESS

Descrição: Operador “menor que”.

- >

Nome *token*: COND\_GREATER

Descrição: Operador “maior que”.

- <=

Nome *token*: COND\_LESSEQUAL

Descrição: Operador “menor ou igual a”.

- >=

Nome *token*: COND\_GREATEREQUAL

Descrição: Operador “maior ou igual a”.

- =

Nome *token*: COND\_EQUAL

Descrição: Operador “igual a”.

- !=

Nome *token*: COND\_NOTEQUAL

Descrição: Operador “diferente de”.

{EXP}	COND	{EXP}
-------	------	-------

- IN

Nome *token*: COND\_IN

Descrição: Operação que permite o uso de um grupo de variáveis.

Sintaxe:

{COLUMN}	COND_IN	(	{VALUE_LIST}	)
----------	---------	---	--------------	---



### 5.2.5 Extras

Pode se notar que existem alguns *tokens* que não são explicados em nenhuma regra anterior. Nessa etapa apresentaremos esses *tokens* que podem ser palavras ou regras.

#### - Expressão

Nome *token*: EXP

Descrição: Grupo de comandos que tem o mesmo grupo de regras de uso.

Sintaxe:

{COLUMN}, {FUNC}, {VALUE_LIST}
--------------------------------

#### - Lista de Valores

Nome *token*: VALUE\_LIST

Descrição: Conjunto de variáveis.

Sintaxe:

(	{STRING}*{,}*{NUMBER}*	)
{STRING},{NUMBER}		

#### - Coluna

Nome *token*: COLUMN

Descrição: Referente a uma coluna do banco.

#### - Tabela

Nome *token*: TABLE

Descrição: Referente a uma tabela do banco.

#### - Palavra

Nome *token*: STRING

Descrição: Palavra utilizada como variável para se efetuar algum comando.

- Número

Nome *token*: NUMBER

Descrição: Número utilizado como variável para se efetuar algum comando.

### 5.3 Estudo sobre funcionalidades

Nessa etapa são discutidas algumas das funcionalidades interessantes para o programa: *auto-complete*, *keep learning*, *improve yourself*.

#### 5.3.1 *Auto-complete*

Na explicação dada sobre o programa na etapa 3 foi demonstrado o guia para formulação de perguntas, conhecido aqui como *auto-complete*. Essa funcionalidade foi inserida nesse estudo por se tratar de uma implementação um tanto quanto complexa para o sistema, mas que traz um grande valor para o aprendizado final do usuário. Essa funcionalidade também facilita o uso da ferramenta, a qual possui um restrito limite de palavras reconhecidas.

A ideia por trás de sua funcionalidade é sugerir possibilidades válidas para a próxima palavra a ser inserida pelo usuário. Como vimos na etapa 4.2, o programa tem sequências de comandos válidos dentro de suas regras, com esse conhecimento podemos traçar o que já foi escrito pelo usuário e o que pode ser aceito na próxima palavra. A dificuldade nessa implementação é o chamado *real timing*. Isto é, enquanto o usuário digita o programa tem que ir atualizando a lista de próximas possibilidades.

#### 5.3.2 *Keep learning*

A funcionalidade *keep learning* foi pensada no intuito de aumentar a quantidade de comandos conhecidos pelo programa. A ideia é dar um ponto de acesso ao usuário onde ele, através de uma interface intuitiva conseguirá inserir um novo comando com suas regras léxicas e sintáticas. Um exemplo de comando novo com suas informações necessárias seria:

Comando: round

Tipo de comando: Função

Regra: 

round	(	VALUE	)
-------	---	-------	---

Palavras semelhantes: arredondar, aperfeiçoar

A maior complexidade dessa funcionalidade é a validação das informações adicionadas pelo usuário, como a regra e a própria existência do comando. Na regra, todas as informações adicionadas pelo usuário têm que ser reconhecidas como *tokens* já existentes no programa ou teria que existir uma nova funcionalidade para inserir *tokens* ao sistema, o que aumentaria ainda mais a complexidade da implementação.

Como parte dessa funcionalidade, deve-se ainda permitir a inserção desses dados no código original do programa. A programação para tal funcionalidade teria que ser dinâmica em uma lista de comandos carregados ao se iniciar o programa ou se adicionar nova funcionalidade.

### 5.3.3 *Improve yourself*

Enquanto era pensado como resolver o problema da variedade linguística da língua portuguesa e das várias possibilidades de tradução para um mesmo comando, além do estudo sobre NLP que será visto na etapa 5.4, também se teve a ideia de adicionar a funcionalidade *improve yourself*, onde o programa aceita entrada de novos sinônimos por usuários.

A ideia é que a funcionalidade funcione da seguinte forma:

- 1-Usuário adiciona uma frase para ser traduzida
- 2-Programa tenta reconhecer a frase
- 3-Programa avisa que não pode gerar a resposta e mostra os comandos reconhecidos na sentença.
- 4-Usuário marca as palavras não reconhecidas pelo programa e as adiciona em um dos comandos existentes.
- 5-Programa confirma a inserção da nova palavra.
- 6-Programa tenta reconhecer a palavra novamente.

Na etapa 4 o programa vai inserir a opção em todas as palavras não reconhecidas a serem adicionadas, ao clicar na opção o programa deve mostrar uma lista de comandos e o usuário escolhe o comando referente a palavra.

## 5.4 Estudo sobre NLP

Por se tratar de um desafio complexo no qual o programa deve reconhecer a língua portuguesa no intuito de gerar seu resultado em SQL, foi levantado que um estudo sobre NLP (*natural language processing*) seria importante

para se entender e descobrir melhorias que poderiam ser inseridas no sistema a fim de agregar um maior reconhecimento da linguagem.

Nessa etapa é bom esclarecer que o intuito é ajudar o sistema a entender as diferentes formas de escrita de uma mesma palavra e não o reconhecimento de um numero maior de palavras. Um exemplo seria o usuário ao inserir a palavra 'selecione', o processamento da linguagem conseguiria reconhecer que sua palavra no infinitivo é 'selecionar' e que a mesma tem ligação com o comando *select*. Abaixo estão listadas as funcionalidades interessantes do processamento de linguagem, retiradas de bibliotecas on-line e que podem ser usadas nesse sistema.

#### 5.4.1 Tokenizers

Essa técnica é responsável pela criação de *tokens* a partir de uma sentença. Essa técnica pode ser muito utilizada na primeira etapa do programa, onde ele deve reconhecer os *tokens* a partir da entrada do usuário. Com isso será possível reconhecer as diferentes palavras em português e aplicar outras técnicas para reconhecer seu específico *token* dentro do sistema.

Essa técnica está contida em mais de uma biblioteca, mas foi levada em consideração a *NaturalNode*[12] para o estudo dessa funcionalidade. A biblioteca tem algumas possibilidades de uso da técnica, a mais interessante para esse estudo é a técnica que retorna apenas os *tokens* referentes às palavras, retirando símbolos e pontuação. Para a separação das palavras é usada um *regex* de espaço e símbolos. A biblioteca também tem uma implementação para receber expressões em *regex* para separar as palavras por espaços e símbolos. O código dessa biblioteca se encontra em javascript, mas pode ser facilmente adaptada para outra linguagem.

```
var natural = require('natural'),
    tokenizer = new natural.WordTokenizer();
console.log(tokenizer.tokenize("your dog has flees."));
// [ 'your', 'dog', 'has', 'flees' ]
```

Figura 12: Exemplo sem pontuação e símbolos.

```
tokenizer = new natural.TreebankWordTokenizer();
console.log(tokenizer.tokenize("my dog hasn't any flees."));
// [ 'my', 'dog', 'has', 'n\'t', 'any', 'flees', '.' ]
```

Figura 13: Exemplo com todas informações.

```
tokenizer = new natural.RegexpTokenizer({pattern: /\-/});
console.log(tokenizer.tokenize("flee-dog"));
// [ 'flee', 'dog' ]
```

Figura 14: Exemplo com inserção de *regex*.

#### 5.4.2 Stemmer

A palavra *stemmer* significa decorrente. Como o próprio nome diz essa técnica informa a palavra de origem. Essa técnica também está presente em mais de uma biblioteca e, nesse caso, iremos usar como fonte a biblioteca *uea-stemmer*[13]. Essa biblioteca é escrita em ruby, uma linguagem simples de ser estudada.

No projeto essa técnica pode ser muito utilizada para reconhecer plural ou formas verbais. Esse é um dos maiores desafios do projeto já que a língua portuguesa nos dá a liberdade de escrever uma frase com o mesmo significado de diferentes maneiras. A biblioteca cria regras da língua inglesa para gerar as palavras originais. Primeiramente, a biblioteca confere se a palavra entrada tem caracteres especiais e os retira ou substitui (exemplo: “m” por “am”). Finalizando, a biblioteca aplica uma lista de regras para descobrir a palavra original de cada caso. Existem três tipos de regras:

##### 1- Regra básica com *regex*

Palavra: *helpers*

Regra: @rules << Rule.new(/^[A-Z]+s\$/, 1, 91.1)

Resultado: *helper*

##### 2- Regra para remoção de sufixo.

Palavra: *scarred*

Regra: @rules << EndingRule.new('arred', 3, 19.1)

Resultado: *scar*

##### 3- Regra para substituição de sufixo

Palavra: *dying*

Regra: @rules<<ConcatenatingEndingRule.new('dying', 4, 58, 'ie')

Resultado: *die*

Pode se notar que para cada tipo diferente de regra existe uma classe que sabe lidar com o caso. Na primeira regra, a inicialização recebe um *regex*, o tamanho do sufixo a ser removido e o número da regra. Na segunda, recebe a inicialização da classe, o sufixo, o tamanho do sufixo a ser removido, o número da regra e, na terceira regra, recebe o sufixo a ser inserido.

Infelizmente, todas as regras dessa biblioteca só se aplicam à língua inglesa. Ainda não foi encontrada uma biblioteca semelhante para a língua portuguesa. Uma estratégia que pode ser usada no projeto é uma biblioteca de palavras (irá ser explicado na etapa 6).

```
stemmer = UEAStemmer.new

stemmer.stem('helpers')    # helper
stemmer.stem('dying')      # die
stemmer.stem('scarred')    # scar
```

Figura 15: Exemplo de uso do técnica *stemmer*.

#### 5.4.3 Post-tagged

*Post-tagged* é responsável por reconhecer o tipo da palavra. Ao se inserir um número qualquer de *tokens* entrados pelo usuário, a técnica de *tag* retorna esses *tokens* com seus respectivos tipos (verbos, pronomes, substantivos, entre outros). Para a explicação dessa técnica iremos usar como exemplo a biblioteca *Treat*[14]. Essa biblioteca foi implementada na linguagem *ruby*, linguagem simples de ser estudada, mas infelizmente, utiliza a ferramenta *Stanford Tagger*[15].

A biblioteca de *Stanford* é disponibilizada para o uso, mas é feita em java e acessada por *.jar*, assim não foi encontrado o código fonte. Usando uma técnica para descompactar os arquivos *.jar* consegue-se chegar aos arquivos *.java* e *.class*. Os arquivos *.java* não foram suficientes para entender o funcionamento da biblioteca. Pela leitura do site e da documentação se entendeu que a abordagem principal é a lista de palavras para cada tipo.

```
(pprint (pos-tag (tokenize "Mr. Smith gave a car to his son on Friday.")))
(["Mr." "NNP"]
 ["Smith" "NNP"]
 ["gave" "VBD"]
 ["a" "DT"]
 ["car" "NN"]
 ["to" "TO"]
 ["his" "PRP$"]
 ["son" "NN"]
 ["on" "IN"]
 ["Friday." "NNP"])
```

Figura 16: Uso da função *post-tag*

Nesse exemplo uma frase de entrada é transformada em *tokens* e, depois, são adicionados os *tags* para cada token separadamente.

#### 5.4.4 String distance

Nessa técnica podemos receber um valor referente à comparação entre duas palavras. A técnica pode ser usada em última ocasião se uma palavra não tiver sido reconhecida ainda, essa é uma boa prática para reconhecer erros de digitação. A resposta do algoritmo é um número entre 0 e 1, onde 0 é sem semelhança e 1 significa palavras iguais. Para explicar o funcionamento dessa técnica iremos usar a biblioteca *NaturalNode*.

A biblioteca usa três algoritmos para calcular a diferença entre as palavras. Os algoritmos são:

##### 1- *Jaro-Winkler string distance*[16]

Esse algoritmo foi inicialmente criado por *Jaro*, e *Winkler* criou uma variação com o nome *Jaro-Winkler distance*. Nesse método leva-se em consideração o tamanho de cada palavra, o número de correlação entre seus caracteres e o número de transposições. Para se explicar o algoritmo vai ser usado o exemplo 'estudar' e 'estdar'.

d1 - tamanho da primeira palavra = 7      m - número de caracteres semelhantes = 6

d2 - tamanho da segunda palavra = 6      t - transposições = 0

L - tamanho até a primeira diferença entre as palavras vezes constante(p) = 3\*0.1

J - valor de *Jaro*

JW - valor de *Jaro-Winkler*

$$J = \frac{1}{3} \left( \frac{m}{d1} + \frac{m}{d2} + \frac{m-t}{m} \right) = J = \frac{1}{3} \left( \frac{6}{7} + \frac{6}{6} + \frac{6}{6} \right) = 0.952380$$

$$JW = J + L(1 - J) = JW = 0.952380 + 3 \times 0.1 (1 - 0.952380) = 0.96666$$

```
var natural = require('natural');
console.log(natural.JaroWinklerDistance("dixon", "dicksonx"));
console.log(natural.JaroWinklerDistance('not', 'same'));
```

Output:

```
0.7466666666666666
0
```

Figura 17: Uso de *Jaro-Winkler* na biblioteca

## 2- Levenshtein Distance[17]

O segundo algoritmo tem uma abordagem diferente, ele não é muito voltado para esse projeto. Nele o resultado varia de 0-∞ dependendo da diferença entre as palavras de entrada. O algoritmo se baseia na quantidade de mudanças que deve ser feita na segunda palavra para alcançar a semelhança com a primeira palavra. As mudanças podem ser troca, remoção e inserção de caracteres. Na biblioteca *NaturalNode* pode-se escolher o peso de cada ação na palavra. Como exemplo, usando o algoritmo com as palavras 'estudar' e 'edtudars', podemos notar que devemos trocar o caracter 'd' na posição 2 pelo caracter 's' e remover o último caracter 's'. Com isso o valor resultante seria 2.

```
var natural = require('natural');
console.log(natural.LevenshteinDistance("ones", "onez"));
console.log(natural.LevenshteinDistance('one', 'one'));
```

Output:

```
2
0
```

Figura 18: Uso de *Levenshtein* na biblioteca



### 3- *Dice's co-efficient*[18]

O último algoritmo trabalha de maneira simples, mas com uma precisão menor que o primeiro. Nesse método o valor retornado também é entre 0-1. O algoritmo divide as palavras em conjunto de dois caracteres seguidos e calcula a quantidade de interseções entre eles. Como exemplo vamos usar as palavras 'estudar' e 'esputar'.

Palavra 'estudar': {'es','st','tu','ud','da','ar'}

Palavra 'esputar': {'es','sp','pu','ut','ta','ar'}

t - número de interseções = 2

x - número de duplas da primeira palavra = 6

y - número de duplas da segunda palavra = 6

D - valor de *Dice*

$$D = 2 \times \frac{t}{x+y} = D = 2 \times \frac{2}{6+6} = 0.3333$$

```
var natural = require('natural');  
console.log(natural.DiceCoefficient('thing', 'thing'));  
console.log(natural.DiceCoefficient('not', 'same'));
```

Output:

```
1  
0
```

Figura 19: Uso de *Dice* na biblioteca

## 6.Aplicação

Nesse tópico será mostrado o funcionamento por completo da aplicação proposta. A aplicação se baseia em três etapas principais, duas delas estudadas anteriormente e uma última, mais simples, que é a criação do código SQL.

A aplicação deve aguardar inicialmente pela entrada do usuário, essa entrada tem que ser referente ao banco conectado pela aplicação. O banco deve conter nomes de tabelas e colunas usadas nas perguntas feitas pelo usuário e

ligações bem definidas por chaves estrangeiras para as tabelas que as necessitam, essas chaves serão usadas no caso de ligação entre duas tabelas.

Após a inserção da frase pelo usuário se inicia a primeira fase da aplicação, que se constitui na análise léxica das palavras. Nessa etapa são aplicadas as técnicas necessárias de NLP para reconhecer as palavras escritas pelo usuário, também são procuradas palavras chaves como nome de tabelas e colunas. Para cada palavra finalizada após o todo o processo é feita a tentativa de conectá-la com *tokens* referente as mesmas. A seguir será demonstrado as diversas técnicas e abordagens desde o momento do recebimento da frase até a classificação de cada palavra.

Entrada: Selecione o nome dos alunos com idade maior que 25.

-O primeiro processo é a divisão da frase em *tokens*.

*Tokens*: {"Selecione", "o", "nome", "dos", "alunos", "com", "idade", "maior", "que", "25"}

-A primeira etapa é procurar por palavras chaves. Nesse caso vamos deduzir que:

Tabela: alunos

Coluna: nome, idade (nesse exemplo levaremos em conta que as colunas pertencem a tabela)

Restantes: {"Selecione", "o", "dos", "com", "maior", "que", "25"}

-Palavras restantes procurar por *tokens* conhecidos

TK\_WHERE: {"com"}

TK\_GREATER: {"maior", "que"}

TK\_NUMBER: {"25"}

Restantes: {"Selecione", "o", "dos"}

-Após essas etapas começa a aplicação de NLP. Nesse exemplo usaremos a técnica de *stemmer*.

{"Selecione"}: {"selecionar"}

-Com isso procuramos novamente por *tokens*:

TK\_SELECT: {"selecionar"}

-Frase final gerada pelo léxico:

```
{TK_SELECT, "o", TK_COLUMN, "dos", TK_TABLE, TK_WHERE,  
TK_COLUMN, TK_GREATER, TK_NUMBER}
```

Então, inicia-se a segunda fase da aplicação: a partir dos *tokens* reconhecidos pela etapa léxica, o sistema deve poder descobrir a que regra e em consequência a qual comando a pergunta original do usuário se relaciona. Nessa etapa o compilador terá a inteligência de eliminar palavras que não foram reconhecidas ou até tentar encontrar o seus significados. Na frase descrita a cima, por exemplo, é esperado que o compilador reconheça a palavra “dos” como sendo o *token* TK\_FROM. Essa lista de entradas será encaminhada para a biblioteca *yacc*[19] onde deve ser gerada uma árvore de comandos e funções a serem chamadas para geração do código final.

```
tipoBase : TK_INT {$$ = FunTipoBase(TIPOINT);}
          | TK_CHAR {$$ = FunTipoBase(TIPOCHAR);}
          | TK_FLOAT {$$ = FunTipoBase(TIPOFLOAT);}
          ;
```

Figura 20: Exemplo de código *yacc*

Na figura 20, podemos notar um código na linguagem *yacc*, nele o tipo de estrutura chamado *tipoBase* pode ser reconhecido pelos *tokens*: TK\_INT, TK\_CHAR, TK\_TKFLOAT. Para cada *token* no seu lado direito pode se reparar a chamada a função pela geração de código para o devido *token* reconhecido. A idéia de funcionamento da aplicação assemelha-se muito ao exemplo descrito a cima. O *yacc* tem como objetivo principal criar a árvore de execução da aplicação, no caso desse sistema, o *yacc* deve gerar uma árvore com a ordem que os comandos devem ser inseridos na resposta final.

A árvore de sintaxe abstrata, como é conhecida, é uma parte bem importante para o programa, então usaremos um exemplo simples matemático para explicá-la. Para iniciar devemos declarar as regras *yacc* para expressão matemática.

```
exp: '(' exp ')'
    | exp '+' exp {$$ = FunExpBinario($1,$3,EXPSOMA);}
    | exp '-' exp {$$ = FunExpBinario($1,$3,EXPSUB);}
    | exp '*' exp {$$ = FunExpBinario($1,$3,EXPMULT);}
    | exp '/' exp {$$ = FunExpBinario($1,$3,EXPDIV);}
```

Quando o *yacc* começa a ler a entrada ele procura as regras que se encaixam com o formato da entrada. Nesse caso simples apenas com as regras de expressões o *yacc* vai procurar como gerar a árvore em relação as hierarquias na regra em relação a entrada. A árvore final para uma expressão como  $((37+4)*3)+45$  seria:

$$\begin{array}{c} + \\ * \quad 45 \\ + \quad 3 \\ 37 \quad 4 \end{array}$$

Finalmente, ao final de todo os processos citados a cima o programa deve retornar o comando gerado e executá-lo no banco conectado. Como vimos anteriormente esse comando é gerado pela árvore de chamadas de funções geradas na segunda etapa. Na explicação anterior geramos uma árvore final, porém o *yacc* gera uma árvore de execução de funções (as que se encontram a direita de cada regra), neste caso cada função retorna o resultado para o seu comando e executadas na ordem da árvore retornam ao fim o comando SQL final para a entrada fornecida.

## 7. Referências bibliográficas

1. Coursera, INC; Mountain View, USA; <http://www.coursera.org>
2. Code School LLC; Orlando, USA; <http://www.codeschool.com/>
3. Hotwoofy Ltd; Sheffield, UK; <http://delver.io/>
4. Microsoft; Redmond, USA; <http://technet.microsoft.com/en-us/library/cc966482.aspx>
5. PrimeQue Ltd; Tel-Aviv Israel; <http://primeque.com/>
6. Informações sobre análise léxica: [http://pt.wikipedia.org/wiki/An%C3%A1lise\\_%C3%A9xica](http://pt.wikipedia.org/wiki/An%C3%A1lise_%C3%A9xica)
7. Informações sobre análise sintática: [http://pt.wikipedia.org/wiki/An%C3%A1lise\\_sint%C3%A1tica\\_\(computa%C3%A7%C3%A3o\)](http://pt.wikipedia.org/wiki/An%C3%A1lise_sint%C3%A1tica_(computa%C3%A7%C3%A3o))
8. Michaelis tradutor on-line: <http://michaelis.uol.com.br/moderno/ingles/index.php>
9. Sinônimos on-line: <http://www.sinonimos.com.br/>
10. Dados de Stanford sobre sintaxe SQL: <http://www.stanford.edu/dept/itss/docs/oracle/10g/server.101/b10758/>
11. W3School: <http://www.w3schools.com/sql/default.asp>
12. Biblioteca NaturalNode: <https://github.com/NaturalNode/natural>
13. Biblioteca Uea-Stemmer: <https://github.com/ealdent/uea-stemmer>
14. Biblioteca Treat: <https://github.com/louismullie/treat>
15. Stanford Tagger: <http://nlp.stanford.edu/software/tagger.shtml>
16. Jaro-Winkler string distance: [http://en.wikipedia.org/wiki/Jaro-Winkler\\_distance](http://en.wikipedia.org/wiki/Jaro-Winkler_distance)
17. Levenshtein distance: [http://en.wikipedia.org/wiki/Levenshtein\\_distance](http://en.wikipedia.org/wiki/Levenshtein_distance)
18. Dice's co-efficient: [http://en.wikipedia.org/wiki/Sørensen-Dice\\_coefficient](http://en.wikipedia.org/wiki/Sørensen-Dice_coefficient)
19. Informações sobre yacc: <http://en.wikipedia.org/wiki/Yacc>