GREGORY SHOLL E SANTOS

GUILHERME CARLOS POLITTA

TRABALHO SEM TITULO

Trabalho apresentado como requisito parcial à Obtenção de grau de Bacharel em Ciências da Computação no curso de graduação em Bacharelado em Ciências da Computação, Departamento de Informática da Universidade Federal do Paraná.

Orientador: Prof. Bruno Muller Junior

CURITIBA

2014

1 INTRODUÇÃO

Esta primeira frase não diz nada. O tema é aplicações com bds legados, com alto custo de migração para tecnologias mais novas.

Torna-se mais frequente casos de empresas que já possuem um banco de dados para armazenar informações importantes tanto para empresa quanto para seus clientes. Empresas com mais tempo no mercado e que, portanto já possuem banco de dados criados a muitos anos, possuem dificuldades de migrar seus sistemas para uma tecnologia mais recente.

Essa dificuldade se dá principalmente pelo custo da migração manual de uma banco de dados para uma tecnologia mais contemporânea. Nesse custo está vinculado o aprendizado de ambas tecnologias, a original e a que se deseja assumir. Ao contar o tempo para fazer dita migração, o custo fica tal que é inviável para um empresa fazer tal processo sem correr riscos de falhas nos seus dados. Existem muitas empresas que não podem se dar o luxo que ocorram tais falhas.

O termo "falha" é muito amplo. Não seria melhor ficar só no custo (homens/hora) e tempo necessário para a migração?

Um processo automático de criação de uma réplica de um banco de dados em outra tecnologia é algo bastante esperado, mas como explicado anteriormente, é inviável de se fazer em uma empresa. Esse processo precisa ser capaz de analisar o código de criação do banco de dados original e interpretá-lo para seu formato equivalente na outra tecnologia.

melhor não ser tão incisivo: "pode ser inviável" é melhor.

A vantagem principal desse método de migração é a rapidez com que é feita toda a análise e criação do novo banco de dados. Essa rapidez facilita a mudança de tecnologia imensamente. Entretanto, somente se o usuário realmente acha necessária tal mudança, as vezes ocorrem situações que as impedem. (1)

Este trabalho propõe um software que gere os código de geração de um banco de dados em Ruby on Rails, dado um código SQL do banco original. Note que a proposta é a geração do banco de dados somente. Toda e qualquer lógica envolvida ainda terá de ser feita manualmente pelo usuário.

O trabalho está organizado da seguinte maneira: conceitos e ferramentas de Rails, banco de dados e analisadores léxicos e semânticos no capítulo 2, proposta de um software que gere nos códigos Rails no capitulo 3, uma implementação da proposta no capítulo 4 e conclusão no capítulo 5.

2 Revisão Bibliográfica

Esse capítulo apresenta as ferramentas e conceitos utilizados no trabalho. A seção **X** apresenta a linguagem declarativa SQL, a seção **X** os Sistemas de Gerenciamento de Banco de Dados (SGBD), a seção **X** o *framework* Ruby on Rails, a seção **X** os analisadores sintáticos e a seção **X** os analisadores léxicos. Falta bibliografia. Converso pessoalmente para explicar.

2.1 Linguagem SQL

O SQL (do inglês, *Structured Query Language*) é uma linguagem de programação feita para gerenciar sistema de banco de dados relacionais que foi desenvolvida no começo dos anos 70 por Donald D. Chamberlin e Raymond F. Boyce. Atualmente, SQL é muito utilizado em gerenciamento e desenvolvimento de banco de dados *Web*.

A linguagem é originalmente baseada em álgebra relacional, facilitando o seu entendimento. Ela é dividida nas linguagens de definição e de manipulação de dados.

2.1.1 Linguagem de definição de dados

A linguagem de definição de dados é a que altera as estruturas e restrições do dados, no caso do SQL essas estruturas são as tabelas do banco de dados. A figura 1 mostra um exemplo de criação de uma tabela Pessoa.

FAZER IMAGEM

```
CREATE TABLE Pessoa (
    id          INTEGER         PRIMARY_KEY,
    nome        VARCHAR(50)     NOT NULL,
    sobrenome   VARCHAR(100)    NOT NULL,
    telefone    INTEGER         NULL
);
```

Além do CREATE para criação de tabelas, pode-se destruir parcial ou totalmente os dados, alterá-los ou inclusive renomeá-los (todas funções

consideram que já foram criadas as tabelas previamente). As chamadas para essas funções são DROP, ALTER e RENAME, respectivamente.

A função DROP pode, por exemplo, tirar somente a coluna de data de nascimento da tabela, assim como também pode remover a tabela inteira. Já a função ALTER altera elementos do objeto do banco de dados, como nome, adicionar/remover colunas etc. A função RENAME é a que renomeia objetos.

2.1.2 Linguagem de manipulação de dados

A linguagem de manipuação de dados é aquela que lida com os dados, propriamente dito. Essas linguagem são as que inserem, deletam, atualizam e selecionam os dados desejados.

apagam. Prefiram os termos em portugues.

Na figura 2, pode-se ver que adicionamos alguns dados no banco de dados na tabela Pessoa (definida na figura 1). Note que não estamos alterando as propriedades da tabela, mas somente inserindo um novo dado que poderá ter todas as características previamente definidas na tabela.

FAZER IMAGEM

INSERT INTO Pessoa VALUES (“João”, “Nascimento”, 5555-5555)

Neste caso, utilizamos o método mais curto de inserção, onde sabemos a ordem das colunas da tabela, então só precisa passar os valores corretamente. No entanto, se a inserção não for preencher todos os campos definidos na tabelas, utiliza-se uma sintaxe como está na figura 3.

FAZER IMAGEM

Acho desnecessário entrar nestes detalhes. Melhor explicar genericamente e incluir uma referência.

INSERT INTO Pessoa (nome, sobrenome) VALUES (“João”, “Nascimento”)

2.2 SGBD

SGBD vem da sigla Sistema de Gerenciamento de Banco de Dados (em inglês, *Database Management System*) e que é um conjunto de programas que gerenciam o acesso e manutenção de um banco de dados. Os SGBDs de banco de dados relacionais são geralmente acompanhados de uma interface

simples e intuitiva para o usuário, retirando a necessidado do usuário intervir na funcionalidade do gerenciamento. ref

Um banco de dados geralmente não é portável para outro SGBD que não seja aquele onde foi criado, mas SGBDs distintos podem se comunicar através de algum protocolo para fornecerem dados entre si. ref

Existem três características fundamentais que todos SGBDs devem ter: linguagem de definição de dados, linguagem de manipulação de dados e processamento eficaz de consultas. ref As duas primeiras já foram definidas na seção 2.1.

INSERIR ALGUMA IMAGEM PARA ESCLARECER

O processamento eficaz é uma parte essencial do SGBD, pois se uma consulta de grau baixo de complexidade demora muito para ser completada, o banco não está com um desempenho bom. Este parágrafo parece-me dispensável

Um SGBD precisa ter integridade semântica, ou seja, manter os seus dados sempre correto em relação ao domínio da aplicação. Caso contrário, perde-se o principal propósito de se utilizar um banco de dados.

É preciso zelar pela segurança dos dados ali armazenados. Isso é obtido por meio de restrição de acesso a certos dados, dando permissões mais elevadas de visualização para certos grupos de usuários. No caso de uma falha, como falta de energia, todo SGBD precisa ter um meio de, quando voltar a funcionar, poder ter os dados corretos até o momento em que a falha ocorreu. Isso é geralmente resolvido por meio de *logs* no sistema, para verificar o que foi feito ou estava sendo feito até o momento da falha.

A característica mais visível de um banco de dados é a sua concorrência. Ao efetuar uma operação, o banco de dados não pode ignorar outras operações que são chamadas enquanto ele está executando uma anterior. É preciso organizar as operações, conforme elas cheguam, de um modo que dê para executar o maior número possível de tarefas simultaneamente, se possível. (2)

2.3 *Framework* Ruby on Rails

Ruby on Rails, ou somente *Rails*, é um *framework* gratuito e de código aberto destinado a aumentar a velocidade e facilidado com que *sites* orientados a bancos de dados são feitos. reef

Esse *framework* utiliza diversos padrões de engenharia de *software* e paradigmos, como convenção sobre configuração (do inglês, *Convetion over Configuration*), ref *Don't Repeat Yourself* (traduz para Não Se Repita), ref o padrão *Active Record* ref e o modelo MVC. ref

Como em muitos *frameworks web*, o Rails utiliza o modelo *model-view-controller* para organizar a programação do projeto. ref Esse modelo é dividido em três componentes: modelo, visão e controlador.

O componente modelo é responsável por gerenciar diretamente os dados, lógicas e regras da aplicação. Ele também notifica os outros dois componentes que mudanças foram feitas no seu estado, permitindo mudança na saída da visão e no conjunto de comandos disponível no controlador.

A visão é responsável por gerar alguma forma de representação dos dados armazenados, seja por texto, diagrama, tabelas etc.

O controlador pode enviar comandos para o modelo para que mudanças sejam feitas nos dados, assim como também pode mandar comandos para a visão associadas para mudar a forma de exibição dos dados (por exemplo, rolar por um documento mais extenso).

A figura 4 abaixo mostra um diagrama da interação desse componentes entre si e com o usuário. Note que o usuário não sabe de que forma os dados estão armazenados, pois para ele isso não importa, geralmente. Ele visualiza apenas as respostas dos comandos gerados por ele. Os comandos são interpretados pelo controlador que os traduz para um comando que o modelo possa executar. Ao realizar o comando, o modelo adquire um retorno e o passa para a visão atualizar a sua mostragem. ref

INSERIR IMAGEM AQUI

O *Active Record* ref é um padrão de projeto de *softwares* que utilizam bancos de dados relacionais. É uma abordagem para o acesso ao banco de dados. Uma tabela desse banco de dados é embrulhada (do inglês, *wrapped*) em uma classe de programação orientada a objetos. Assim, um objeto dessa classe fica ligado a somente uma tupla dessa tabela.

Ao criar um novo objeto, uma nova tupla também é criada na tabela. Mudanças como atualização e seleção de objetos utilizam o banco de dados, porém de uma forma mais intuitiva para os programadores.

A figura 5 mostra um exemplo de um pseudo-código do padrão *Active Record* e também a sua versão equivalente em SQL.

INSERIR CÓDIGO ACTIVERECORD/SQL

A biblioteca *Active Record* do Ruby implementa ORM (do inglês, mapeamento objeto-relacional), que é uma técnica de desenvolvimento utilizada para diminuir os conflitos entre programação orientadas a objetos e banco de dados relacionais. Isso cria um modelo de domínio persistente, onde a lógica e os dados são apresentados como um pacote unificado.

O Rails também possui uma série de linhas de comandos para facilitar o seu uso no dia-a-dia. De todas elas, uma possui maior importância nesse trabalho que é o *generate*. (3)

O *rails generate* possui várias utilizações, como criar controladores, visões, etc. Sua importância aqui é para criação de modelos. É possível passar os atributos que compõem o modelo e quais as suas relações com outros modelos. Ao final de tudo, ao executar o modelo, será criada uma tabela em um banco de dados relacional com as informações passadas pelo *generate*.

A figura 6 mostra um exemplo de utilização do *rails generate* e a figura 7 retrata a tabela que será gerada ao final da execução.

INSERIR FIGURAS DO GENERATES E DA TABELA FINAL

2.4 Analisadores sintáticos

Análise sintática é uma técnica emprega no estudo da estrutura sintática de uma linguagem (citação), conforme as regras de uma gramática formal.

Essa análise resulta em uma árvore ou outra estrutura hierárquica que mostra o relacionamento entre cada símbolo reconhecido.

Um analisador sintático tem a função de determinar como a entrada pode ser derivada a partir do símbolo inicial da gramática. Existe duas maneiras de realizar essa tarefa:

1. *Top-down* ou descendente. O analisador começa do símbolo mais alto da hierarquia de análise e tenta transforma-lo na entrada
2. *Bottom-up* ou Ascendente. O analisador começa com a entrada de dados e tenta reescreve-la até chegar ao símbolo inicial mais alto da hierarquia.

O analisador LL (Left-to-right, Leftmost derivation) faz a análise *Top-down* indo da direita pra esquerda, preferindo sempre derivar o símbolo mais à esquerda. Já o analisador LR (Left-to-right, Rightmost derivation) faz a análise *Botton-up*, indo da direita pra esquerda, preferindo derivar o símbolo mais à direita. Ambos são normalmente acompanhados de um número k para indicar que são autorizados a olhar k entradas a frente para evitar *backtracking* ou adivinhação.

2.4.1 Analisador LALR

O analisador LALR (do inglês, *Look-Ahead Left-to-Right parser*) foi inventado por Frank DeRemer para a sua dissertação de doutorado titulada *Practical Translators for LR(k) languages* em 1969 como uma forma simplificada do analisador regular LR.

O objetivo dessa analisador era facilitar a implementação de analisadores LR(1) sem que perder o seu poder. DeRemer conseguiu mostrar em seu trabalho que o seu analisador era mais potente que o LR(0), mas que haviam linguagens que pertenciam a LR(1) e que não pertenciam a LALR. Mais tarde foi demostrado que, mesmo sendo mais fraco que o LR(1), o seu analisador era potente o suficiente para muitas das principais linguagens de programação. (não esquecer citação aqui). (4)

Os primeiros algoritmos para gerar analisadores LALR foram feitos em 1972 (citação), hoje em dia os geradores mais comuns são o YACC e o GNU Bison.

2.4.2 YACC

YACC (do inglês, *Yet Another Compiler Compiler*) é um gerador de analisadores sintáticos LALR, desenvolvido no começo dos anos 1970 por Stephen C. Johnson.

Normalmente utilizado para a construção de compiladores, o YACC proporciona uma ferramenta na qual o usuário específica uma estrutura de entrada junto com um código para ser invocado cada vez que uma estrutura é reconhecida.

A figura 1 mostra o YACC sendo utilizado. No painel A coluna temos um trecho de código em Pascal que será utilizado como entrada para o código YACC, no painel B temos a estrutura definida por um usuário com o YACC e no painel C a saída gerada em Java. Como pode ser visto, o código YACC detecta quando uma linha começa com a palavra-chave *var* seguida de um dois pontos, dois identificadores quaisquer e um ponto e vírgula, gerando a saída em Java "*String x;*". Em blocos entre colchetes é possível escrever qualquer código em C, portanto poderíamos, por exemplo, fazer a verificação de que se *String* é um tipo suportado em nossa estrutura.

FAZER A IMAGEM DE EXEMPLO.

2.5 Analisadores Léxicos

Gostei mais da explic. do Lex: sucinta e direta.S

Análise léxica é o processo de converter uma sequência de caracteres em símbolos, permitindo que seja feita a verificação de que esse caracteres pertencem ao alfabeto de análise. Portanto, um analisador léxico implementa um autômato finito que reconhece os símbolos como sendo válidos ou não a uma certa linguagem.

A implementação desses analisadores requerem a descrição do autômato que reconhece a gramática ou expressão regular desejada. A sequência de

caracteres de entrada é estruturada como uma lista de símbolos, que o analisador vai utilizar como entradas para o autômato. Se o analisador terminar de consumir os símbolos em um estado final, a entrada é dado como válida, se acabar em um estado não final ou não houver estado para o qual possa ir, a entrada é considerada inválida para a gramática.

2.5.1 Lex

Lex é um gerador de analisadores léxicos, escrito por Mike Lesk e Eric Schmidt em 1975. Ajuda a escrever programas cujo fluxo de controle é dirigido por instâncias de expressões regulares, ou seja, as entradas podem ser interpretadas por essas expressões.

Um código que utiliza Lex pode ser dividido em duas partes, a declaração da expressão regular e a sequência de ações de devem ser executadas quando essa expressão é reconhecida.

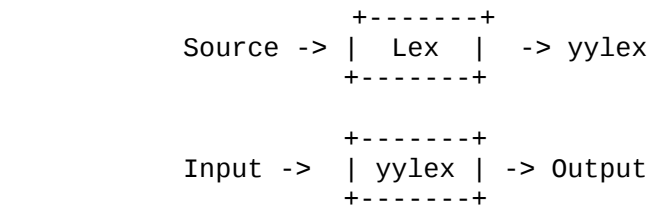
A figura 2 mostra um trecho de código que utiliza Lex. Na parte superior temos a expressão que vai reconhecer qualquer sequência de caracteres que comece com uma letra (maiúscula ou minúscula) e o restante de qualquer letra ou número. Na parte inferior temos o código que deve ser executado quando a expressão for reconhecida, nesse caso só imprime um texto no terminal.

FAZER IMAGEM (ou talvez código mesmo) COM O CODIGO ABAIXO:

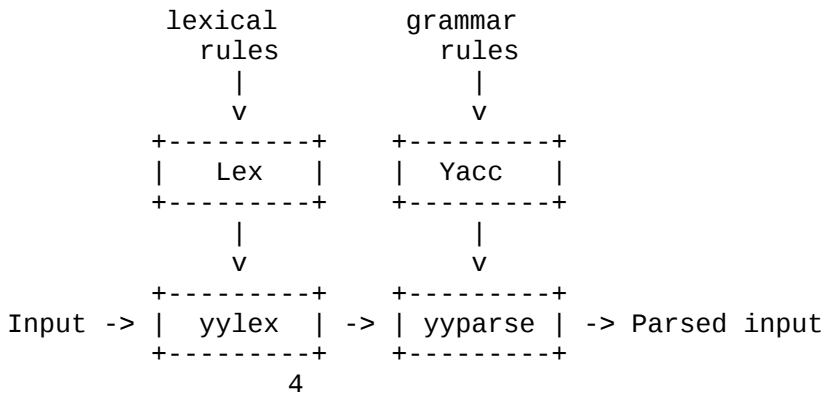
```
ident [a-zA-Z][a-zA-Z1-9_]*


{ident} { printf(“encontrei um identificador válido!” ) }
```

Como o Lex só consegue trabalhar com máquinas de estado finito e o YACC não consegue ler simples entradas de dados, trabalhando apenas com uma séria de *tokens,* eles são utilizados em conjunto, da forma que o Lex serve como um pré-processador do YACC gerando os *tokens* que ele necessita. A

figura 3 e 4 explicam essa relação. O Lex recebe a entrada de dados e as expressões regulares, gerando a rotina chamada *yylex,* a saída gerada serve como entrada para a rotina *yyparse* criada pelo YACC usando as regras gramaticais. Portanto, cada vez que o YACC precisa de um novo *token* ele invoca o Lex, que processa os dados de entrada e retorna a primeira expressão identificada.



3



4

Parte referente a: Capitulo 2: Rails (generates), SGBD/SQL (sintaxe), Gramáticas / Parser, YACC (mais sintaxe). **Reescrever objetivos com nova "linguagem"**

Faltou essa parte de reescrever os objetivos, acho melhor escrever a introdução antes.

3 Proposta do Trabalho

A implementação proposta nesse trabalho é da construção de uma fermenta que, a partir de uma entrada com tabelas em SQL, gere comandos *rails generate* para migrar um projeto legado para uma aplicação em *Ruby on Rails.*

Para isso, utilizamos as ferramentas YACC e Lex, como interpretadores do arquivo de entrada, e a linguagem de programação C, para unir as duas ferramentas e gerar os comandos de saída.

A figura X mostra um exemplo da entrada da nossa ferramenta. Essa é composta de uma ou mais tabelas descritas em SQL e separadas por uma linha em branco. A saída esperada para esta entrada, exemplificada na figura X, são comandos *rails generate* que descrevem o mesmo modelo passado como entrada, mantendo as mesmas relações entre as tabelas. Podemos notar que os campos customer_id e order_id não são explicitamente declarado, já que o *rails generate* cria esses campos automaticamente.

```
CREATE TABLE customers (
    customer_id INT AUTO_INCREMENT PRIMARY KEY,
    customer_name VARCHAR(100)
);


CREATE TABLE orders (
    order_id INT AUTO_INCREMENT PRIMARY KEY,
    customer_id INT,
    amount DOUBLE,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

rails generate model customers customer_name:string --force

rails generate model orders customers:references amount:decimal{42.6} --force

//parte antiga do 3 ignorar

Atualmente não existe nenhuma proposta automática e consolidada para a migração de projetos com base SQL para *Rails*, portanto é necessário analisar todas as tabelas, estudar a estrutura utilizada pelo *Rails* e, finalmente, criar manualmente os comandos *rails generates* ou arquivos de modelos apropriados. Isso tudo dificulta a adoção e popularização do *framework* pelos desenvolvedores, que na maioria das vezes já possui sistemas funcionando e por isso não querem deixar tudo para trás.

A nossa solução se diferencia por facilitar e automatizar todo esse trabalho. O usuário precisa apenas passar um arquivo de texto com as tabelas em SQL do seu projeto atual que o **NOME DO PROJETO** gera um arquivo de saída com comandos necessários para a migração para *Rails,* mantendo as mesmas características de relacionamento entre as tabelas que existiam inicialmente.

Para o desenvolvimento do **NOME DO PROJETO,** utilizamos as ferramentas YACC e Lex para interpretar o arquivo de entrada. A interação entra as duas ferramentas e a geração dos comandos de saída foram feitos na linguagem C.

Conceitual, proposta do trabalho;

Gramática de entrada -> Parser -> Saida (comando rails)

Não esqueçam de citar que o analisador sintático que vocês criaram foi para fazer parsing de SQLschema.

.

4 IMPLEMENTAÇÃO

//mal foi começado, ignorar

Para interpretar o arquivo de entrada, criamos um arquivo Lex que identifica todos os símbolos possíveis de entrada, que podem ser visto na Tabela 1. Também foi criado um arquivo YACC que espera uma sequência símbolos em uma ordem pré-estabelecida. A cada símbolo da sequência, o YACC pede para o Lex se o próximo símbolo da entrada bate com o que é esperado, se sim, o YACC vai para o próximo passo, senão o programa aborta pois o arquivo de entrada está em formato inválido.

Cada vez que é reconhecida a criação de uma tabela (normalmente pelos símbolos CREATE TABLE <nome da tabela>) , é

CRIAR UMA TABELA COM AS COLUNAS: EXPRESSÃO REGULAR NO LEX E SÍMBOLOS RECONHECIDOS PELO COMANDO

Como funciona, exemplos, aplicações, desempenho, possíveis problemas (**todos os tipos de relacionamento funcionam? 1- 1, 1-n, n-n? etc**)

Não precisam entrar neste nível de detalhe. Um apêndice indica a gramática de entrada, e este capítulo cita o apêndice e concentra-se nos "melhores momentos" do parser (os que geram código rails).

5 CONCLUSÃO