

Universidade do Minho
Conselho de Cursos de Engenharia
Licenciatura em Engenharia de Sistemas e Informática

Disciplina LI4 - Projecto

Ano Lectivo de 2007/08



escola de engenharia



departamento de
informática

Layer SQL on Berkeley DB

47030 Boris Victorovitch Tchikoulaev
47104 Bruno Silvestre Medeiros
47050 Adelino Miguel Portela

Julho de 2008

| | |
|------------------|--|
| Data de Recepção | |
| Responsável | |
| Avaliação | |
| Observações | |

Layer SQL on Berkeley DB

Boris Victorovitch Tchikoulaev

Bruno Silvestre Medeiros

Adelino Miguel Portela

Julho de 2008

Resumo

Este trabalho consiste na realização de um parser entre a Linguagem de manipulação de Bases de Dados - SQL e as funções criadas por nós para a realização dos respectivos comandos sobre as Berkeley DB.

Inicialmente tivémos de escolher as melhores estruturas de dados a usar, para guardar toda a informação respectiva aos comandos SQL recebidos. De seguida, elaborámos as funções que interagem directamente com o sistema das Berkeley DB, nomeadamente para criar tabelas, inserir dados, selecciona-los etc. Finalmente, usamos as ferramentas Flex/Yacc para a realização do parser entre os comandos SQL e as nossas funções que trabalham com as referidas Bases de Dados.

Este trabalho pode ser considerado uma base bastante completa para uma layer de SQL sobre as Berkeley DB, que pode ser adaptada, de modo a ser possível trabalhar sem limites com estas Bases de Dados através de comandos SQL.

Área de Aplicação: Desenho e arquitectura de Sistemas de Base de Dados Berkely.

Palavras-Chave: Berkeley DB, SQL, Parser.

Conteúdo

| | |
|--|-----------|
| Resumo | i |
| Conteúdo | ii |
| Lista de Figuras | iii |
| Lista de Tabelas | iv |
| 1 Introdução | 1 |
| 1.1 Contextualização | 1 |
| 1.2 Motivação e Objectivos | 1 |
| 1.3 Estrutura do Relatório | 2 |
| 2 Descrição | 3 |
| 2.1 Berkeley DB | 3 |
| 2.1.1 Conceitos | 3 |
| 2.1.2 Estruturas de Dados | 7 |
| 2.1.3 Funções C para Berkeley DB | 8 |
| 2.1.4 Exemplos de Algoritmos | 9 |
| 2.2 Parser | 12 |
| 2.2.1 Gramática Tradutora | 12 |
| 2.2.2 Analisador Léxico | 15 |
| 3 Conclusão e Trabalho Futuro | 17 |
| Bibliografia | 18 |
| Referências WWW | 19 |
| Lista de Acrónimos | 20 |

Lista de Figuras

| | |
|--|----|
| 2.1 Ilustração da nossa Gramática Tradutora. | 13 |
|--|----|

Lista de Tabelas

| | | |
|-----|--|---|
| 1.1 | Relação entre termos SQL e Bekeley DB. | 2 |
|-----|--|---|

1 Introdução

Este documento diz respeito ao relatório do Projecto da disciplina Laboratórios de Informática IV do 3º ano de LEI.

Este projecto consiste em complementarmos as Berkeley DB, de modo a poderem funcionar com comandos SQL.

A Berkeley DB é uma biblioteca que proporciona um alto desempenho a base de dados embutidas, que trabalha com C, C++, Java, Perl, Python, Ruby, Tcl, Smalltalk, e muitas outras linguagens de programação.

Em primeiro lugar, analisámos o modo de funcionamento deste tipo de base de dados. De seguida criámos funções em C para interagir com a Berkeley DB, de modo a implementar todas as operações necessárias para realizar queries. Finalmente, elaborámos um Parser entre a linguagem SQL com as referidas funções criadas, usando as ferramentas Flex/Yacc.

A linguagem SQL é bastante utilizada para realizar 'queries' a Bases de Dados e após a realização deste projecto, estas eficientes Berkeley DB tornam-se bastante mais práticas da perspectiva do utilizador.

1.1 Contextualização

As Berkeley BD são um sistema de Bases de Dados super eficientes. No entanto, não têm a capacidade de reconhecer comandos SQL, linguagem mais usada e mais prática no que diz respeito a Bases de Dados. Este tipo de Bases de Dados poderiam tornar-se então bastante mais usadas e conhecidas se tal fosse possível.

1.2 Motivação e Objectivos

Berkeley DB não é uma BD relacional. Como tal não suporta queries SQL, tal como referimos anteriormente.

Todos os acessos á informação fazem-se então através da sua API. Os utilizadores são assim obrigados a aprender um novo conjunto de *interfaces* de modo a poderem trabalhar com este tipo de Bases de Dados. Apesar dessas *interfaces* serem relativamente simples, não são *standard*, tornando-se pouco prático o seu uso.

Neste projecto, vamos implementar a linguagem SQL sobre as Berkeley DB, de modo a ser possível aceder estas Bases de Dados através de simples queries, não sendo necessário aprender a sua API.

Apesar de não interpretar comandos SQL, existe alguma relação entre SQL e Berkeley DB. Na tabela seguinte mostrámos a relação entre alguns conceitos equivalentes:

| Termo SQL | Berkeley DB equivalente |
|-----------------|-------------------------|
| Database | Environment |
| Table | Database |
| Tuple/row | Key/data pair |
| Primary index | Key |
| Secondary index | Secondary database |

Tabela 1.1: Relação entre termos SQL e Berkeley DB.

Vamos implementar todos os comandos SQL mais relevantes, deixando os restantes para um Trabalho Futuro.

1.3 Estrutura do Relatório

O relatório está segmentado em várias partes: a introdução, onde contextualizamos o nosso projecto e especificamos o porquê do enunciado escolhido; descrição, através da qual se fica a perceber o conteúdo do nosso trabalho mais detalhadamente, quer em relação ao Parser, quer no que diz respeito ás funções criadas para interagir com a Berkeley DB; e por fim a conclusão e a proposta de Trabalho Futuro.

2 Descrição

2.1 Berkeley DB

2.1.1 Conceitos

Vamos dar a conhecer alguns dos muitos conceitos que serão úteis para quem quer trabalhar com este sistema de base de dados.

As Bases de Dados Berkeley contêm registos, e cada um deles representa uma única entrada nessa BD. Cada registo é composto por um par de informação (chave, dados), e estes podem ser compostos por qualquer tipo de dados de C, incluindo estruturas de dados complexas, o que faz com que possámos transformar um tabela de duas colunas numa de 'n' colunas.

Nas *Berkeley DB*, aplicação é similar a uma tabela num sistema relacional de base de dados. Estas aplicações podem utilizar múltiplas Bases de Dados, utilizando para isso um mecanismo opcional chamado *enviromments*.

Berkeley DB providenciam um tipo especial de Bases de Dados, chamadas "secondary databases", servindo estas como índices para as tabelas normais.

Métodos de Acesso

- *BTree*
- *Hash*
- *Queues*
- *Recno*

Utilizámos o método de acesso *Hash* que passamos a descrever de seguida:

A informação é gravada numa tabela de *Hash* linearmente extensiva, a chave e a informação usadas para os registos da *Hash* podem ser estruturas de dados complexas. Registos duplicados são opcionalmente suportados.

Operações Básicas

Neste tópico serão mostradas as funções principais e secundárias, utilizadas no âmbito deste trabalho, de modo a efectuar operações necessárias nas Bases de Dados.

Abrir uma BD

```
DB *dbp; // handle da estrutura DB.

u_int32_t db_flags; // database flags de abertura

int ret; // vai server para retornar o resultado da funcao

// Funcao que abre uma database
int criarTabelas(char *nomeDaTabela){
    ret = db_create(&dbp,NULL,0); // Inicializar o DB handle
    if(ret != 0) printf("\nSucesso."); // tratamento de erro
    else printf("\nInsucesso.");

    // Abrir as flags da tabela
    db_flags = DB_CREATE; // Criar a database no caso dela não existir.

    // Abrir a database
    ret = dbp->open(dbp,NULL,nomeDaTabela,NULL,DB_HASH,db_flags,0);
    dbp->sync(dbp,0); // salvar em disco esta operacao

    // tratamento de erro
    if(ret != 0) printf("\nOcorreu um erro ao abrir a tabela");
    else printf("\nTabela aberta com sucesso.");

    return ret;
}
```

Fechar uma BD

```
void fecharBD_Tabelas(){
    if(dbp != NULL) dbp ->close(dbp,0);
}
```

Flags de Abertura

Vejamos agora um pequeno conjunto de flags que poderão ajudar na altura de abrir uma BD

- 'DB_CREATE' - Se a BD não existir, será criada. Por definição, a BD falha ao abrir caso não exista.
- 'DB_RDONLY' - Abre a BD apenas para operações de leitura. Qualquer operação de gravação originará um erro.

Métodos Administrativos

Alguns métodos que poderão ser úteis para o caso de trabalhar com Berkeley DB:

- 'DB->get_open_flags()' - Retorna a actual flag de abertura.
- 'DB->remove()' - Remove uma BD específica. Se não for dado nenhum valor para o parâmetro da BD, então todo o ficheiro referenciado por este método será removido.
- 'DB->rename()' - Muda o nome de uma BD específica. Se nenhum valor for dado ao parâmetro da BD, então será mudado o nome do ficheiro referenciado por este método.

Funções para reportar erros

Para simplificar o relatório de erros, a estrutura Berkeley DB oferece vários métodos, que estão especificados a seguir:

- 'set_errcall()' - Define a função que será chamada quando um mensagem de erro é emitida pela BD.
- 'set_errpfx()' - Coloca o prefixo "used to" para qualquer mensagem de erro lançada pela "DB library".
- 'err()' - Uma mensagem de erro é enviada para a "callback function" como definido por 'set_errcall'. Se este método não foi utilizado, então a mensagem de erro é enviada para o ficheiro definido por 'set_errfile()'. Se nenhum destes métodos foi utilizado a mensagem de erro é enviada para o "standard error".
- 'errx()' - Comportamento idêntico ao 'err()' , exceptuando a mensagem de texto DB, associada com o valor do erro que não são anexadas a um string de erro.

Gravar na Base de Dados

Os registos são gravados na BD usando qualquer tipo de organização que seja requerida pelo método de acesso que nós selecionamos (Hash). Em qualquer dos casos, a mecânica de inserir ou retirar registos não se altera. Assim que tivermos selecionado o método de acesso, configura-se as rotinas, e abre-se a BD.

Em termos de código do utilizador, o 'put' e o 'get' da BD é o mesmo, independentemente do método de acesso que se utilize. Utilizamos o 'DB -> put ()', para colocar ou gravar, no registo da BD.

Este método requer que providenciemos um registo chave e *data* em forma de par das DB estruturas. Pode-se também providenciar uma ou mais flags de controlo do comportamento das DB's para gravar a BD. Das flags disponíveis para este método, 'DB_NOOVERWRITE' será a mais útil. Esta flag, não permite a gravação ou substituição de um registo já existente. Se a chave providenciada já existir na BD, então este método vai retornar 'DB_KEYEXIST', mesmo que a BD suporte duplicação.

Para recuperar informação armazenada

Pode-se usar o método 'DB->get()' para recuperar os registos da BD. De notar que se a BD suportar duplicados, então por definição este método vai apenas retornar o primeiro registo no conjunto de duplicados. Por esta razão, se a BD suportar duplicados, a solução mais comum será usar um cursor para recuperar os registos desta.

Por definição, 'DB->get()' retorna o primeiro registo que encontrar a qual a chave associada é igual a que é passada como argumento do método. Se a BD suportar registos duplicados, podemos alterar este comportamento, através do fornecimento da flag 'DB_GET_BOTH'. Esta flag faz com que 'DB->get()' retorne o primeiro registo que é igual ao par (*key,data*) providenciados. Se a chave específica e/ou *data* não existir na base de dados, este método vai retornar 'DB_NOTFOUND'.

Eliminar registos

Para eliminar registos da BD podemos utilizar o método 'DB->get()'. Se a base de dados suportar duplicados, então todos os duplicados associados á chave providenciada serão eliminados. Se quisermos eliminar apenas um dos registos da lista de duplicados teremos que usar *cursors*.

Data persistente

Quando se altera a base de dados, esta alteração é feita na memória cache. Isso significa que a informação modificada não é necessariamente carregada para o disco, e tal significa que os dados poderão não aparecer na BD após reiniciar a aplicação.

Se quisermos garantias de que as modificações da BD serão guardadas, então deve-se periodicamente usar o método 'DB->sync()'. Este método faz com que a entrada na memória cache e os ficheiros contidos na cache do sistema operacional sejam gravados em disco.

2.1.2 Estruturas de Dados

Para guardar toda a informação necessária para realizar o nosso projecto, vamos usar as seguintes estruturas de dados: 'TAB' e 'LIN', cujas descrições vamos apresentar de seguida.

A estrutura de dados 'TAB', vai ser usada para guardar a informação referente às colunas da tabela, isto é, vai representar uma lista de colunas. Ora esta estrutura é ideal para o nosso trabalho, na medida em que o número de colunas que vamos receber não é á priori conhecido, e deste modo é possível criar 'n' colunas.

Passamos agora a explicar os campos da estrutura 'TAB':

- 'char *nomeDaColuna' : Vai servir para guardar o nome da coluna;
- 'char *tipo'; Vai servir para guardar o tipo de dados que esta coluna suporta (ex : varchar(), int , number, ect..);
- 'char *tabelaExterior': Vai servir para indicar se esta tabela está associada a outra (caso seja uma chave estrangeira), no caso de não estar associada 'char *tabelaExterior = "NULL" ';
- 'char *flag': Vai servir para indicar se estamos perante uma chave primária (flag = 1) , chave estrangeira (flag = 2), ou nenhuma das duas (flag = 0).

```
typedef struct TAB{  
  
    char *nomeDaColuna;  
    char *tipo; //tipo de dados  
    char *tabelaExterior;  
    char *flag; // 0, 1 chave primaria, 2 chave estrangeira  
    struct TAB *next;  
  
}Node, *TABELA;
```

A estrutura de dados 'LIN', vai ser usada para guardar a informação referente a uma célula de uma tabela. Ora como esta estrutura de dados é uma lista ligada, vai permitir guardar

informação para toda uma linha (ou seja um conjunto de células). Existe apenas um campo - 'char *valor', que vai guardar o conteúdo da célula.

```
typedef struct LIN{  
  
    char *valor;  
    struct LIN *next;  
  
}Node_LIN, *LINHAS,*COLUNAS;
```

2.1.3 Funções C para Berkeley DB

Create

Para fazermos a função 'create' tivémos que ter em conta 2 pontos:

- 1) Como vamos definir as estruturas se não sabemos o número de colunas que a tabela vai receber?
- 2) E onde vamos guardar o nome das colunas, se não podemos definir à priori estruturas de dados com o nome delas, uma vez que são colunas genéricas?

O primeiro ponto foi facilmente resolvido, com a criação da estrutura 'TAB', descrita anteriormente, que é o mais genérico possível e que representa uma lista de colunas. Para cada coluna vamos ter então a informação mais relevante sobre a mesma (nome, tipo de dados, tipo de chaves, etc). Como se trata de uma lista ligada, o número de colunas não vai originar problemas.

O segundo ponto, deve-se ao facto de, usando listas ligadas em 1), não vamos poder guardar na base de dados o nome das estruturas como sendo o nome das colunas, uma vez que o nome da estrutura vai ser o mesmo para todas as colunas, isto é, sempre que se reiniciasse a aplicação e a informação gravada na lista ligada fosse libertada, não poderíamos recuperá-la novamente. Para o caso de querermos fazer um 'insert' por exemplo, a única maneira seria o utilizador fazer sempre 'create table' antes de um 'insert', e assim ficaríamos novamente com a informação referente as colunas carregada em memória - tal implementação não seria aceitável!

A solução foi simples, e consiste no seguinte: sempre que o utilizador efectuar o comando 'creat table', vamos criar uma BD com o nome da tabela dado como argumento, e será carregada a informação da estrutura de dados 'TAB'. Posteriormente essa informação será guardada num ficheiro com extensão '.info' (por ex. "minhatabela.info").

Insert

A função que vai inserir valores em colunas recebe como argumentos o nome da tabela onde se pretende inserir, o nome das colunas e as respectivas linhas a inserir. Em primeiro lugar, separa-se as chaves da *data*, concatenando cada um deles em strings. De seguida, a lista dos valores das respectivas linhas vão ser metidas por ordem de inserção na tabela, comparando-os com o nome das colunas. Concatena-se então todas as linhas das colunas, e todas as strings resultantes desta última acção. Finalmente, o par (*key,data*) é inserido através do método 'put'.

Select

Ao fazer o *Select* de uma key, vai-se pesquisar directamente por esta na BD, e se/quando for encontrada devolve a respectiva data associada. No caso de se fazer um *Select* de várias keys, então teremos de ver todas as chaves de cada linha. Se uma das keys pedidas encontrar-se em alguma linha então a respectiva data associada será devolvida. No entanto se não estiver contida na linha então a pesquisa irá prosseguir para a linha seguinte.

2.1.4 Exemplos de Algoritmos

Vamos mostrar alguns dos algoritmos utilizados para construirmos as nossas funções em C, de modo a interligar os argumentos recebidos dos comandos SQL com a Berkeley DB.

Função Create

```
void creatSQL (char *nomeDaTabela, TABELA tab){

    criarTabelas(nomeDaTabela);          //cria database (a tabela) em DB
    saveInfoTabela (tab,nomeDaTabela); //guarda a tabela em ficheiro (.info)
    fecharBD_Tabelas(); // fecha a database

}

//Funcao que vai guardar uma tabela em ficheiro.
int saveInfoTabela (TABELA table,char *nomeTabela)
{
    FILE *fp;
    int i, j;
    fp = fopen(nomeTabela,"w");
    if(!fp) return 0;
}
```

```

else{

// vamos percorrer as colunas da tabela
while(table){
    fprintf(fp,table->nomeDaColuna);
    fprintf(fp,"\n");
    fprintf(fp,table->tipo);
    fprintf(fp,"\n");
    fprintf(fp,table->tabelaExterior);
    fprintf(fp,"\n");
    fprintf(fp,table->flag);
    fprintf(fp,"\n");
    table = table->next;
}
}
fclose(fp);

return 1;
}

```

Utilizámos também as funções de abrir e fechar *database* descritas anteriormente:

- 'criarTabelas(char *nomeDaTabela);'
- 'fecharBD_Tabelas()';

Vejamos agora o formato de gravação textual da tabela:

«Nome da coluna»

«Tipo de dados»

«Tabela Exterior»

«Flag»

Podemos concluir então que o número de linhas do ficheiro vai ser o número de colunas multiplicado por quatro.

Função *Insert*

```
void inserirLINHAS(char *tabela,COLUNAS col,LINHAS lin){
```

```

TABELA tab = NULL;
LINHAS listaKeys = NULL;
int num_keys = 0;
int all_keys[100];
int buffsize,bufflen;
int buffsizekey,bufflenkey;
char *databuff;
char *buffer;
char *bufferkey;

// vamos carregar a informacao do ficheiro sobre a tabela
tab = loadInfoTabela("ana.info",tab);
criarTabelas("ana.bd");

// vamos procurar todas as key's primarias existentes na tabela
num_keys = lookForAllKey(tab,col,all_keys);

// vai gravar numa lista todas as keys
listaKeys = todosValores(lin,all_keys,num_keys);

// vai gravar as linhas por ordem de colocação nas colunas
lin = gravarLinha(tab,col,lin,all_keys,num_keys);

// Vamos achar o buffer.
buffsize = memoryNeed(lin); //tamanho da data
databuff = malloc(buffsize); //criar o espaco para o data

memset(databuff,0,buffsize);

// copiar tudo para dentro do buffer
bufflen = bufferCopy(lin,databuff);

// guardar a informacao
memset(&data,0,sizeof(DBT));
memset(&key,0,sizeof(DBT));
data.data = databuff;
data.size = bufflen;

lin = limpaMemoria(lin);
col = limpaMemoria(col);

buffsize = memoryNeed(listaKeys);

```

```

bufferkey = (char *) malloc(sizeof(buffsize));
gravarKeys(listaKeys,bufferkey);

key.size = buffsize;
key.data = bufferkey;

printf("\nData [%s]\nkey [%s],keysize -> %d",
        data.data,key.data,key.size);
getchar();
ret = dbp->put(dbp,NULL,&key,&data,DB_NOOVERWRITE);
if(ret == DB_KEYEXIST)
    dbp->err(dbp,ret,"\nERRO ->");
else{
    printf("Chave colocada com sucesso");
    dbp->sync(dbp,0);
}
free(databuff);
fecharBD_Tabelas();
}

```

2.2 Parser

2.2.1 Gramática Tradutora

Nesta secção vamos descrever a nossa Gramática Tradutora para os comandos SQL. Após ter elaborada esta última, usámos as ferramentas Flex/Yacc para criar o nosso Parser.

Apresentamos de seguida a nossa Gramática Tradutora completa, criada para reconhecer os comandos mais relevantes da Linguagem SQL:

$G = \langle T, N, S, P \rangle$

$T = \{\text{SELECT FROM CREATE TABLE INSERT INTO VALUES}$
 $\text{WHERE AND INDEX ON DROP DELETE num var str data_type opd restr}\}$

$N = \{\text{Tabelas Colunas Tabela Coluna Restricao Valores Val}$
 $\text{DadosTabela LinhaTabela Condicoes Condicao}\}$

$S = \{\text{ComandosSQL}\}$

$P = \{$

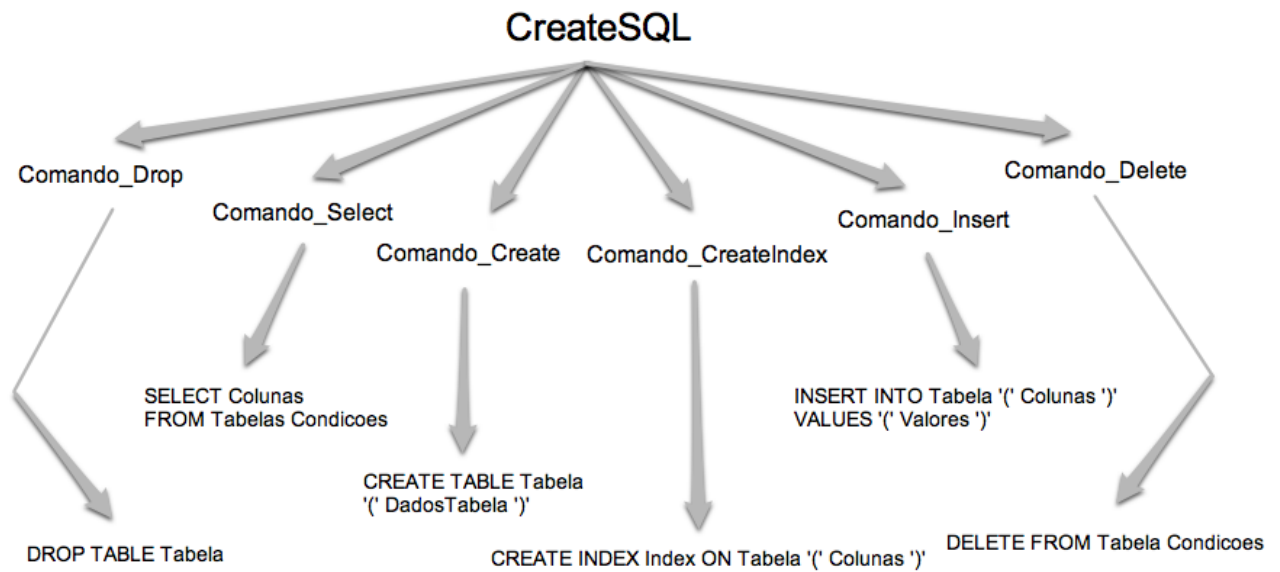


Figura 2.1: Ilustração da nossa Gramática Tradutora.

```

ComandosSQL : Comando
              | ComandosSQL Comando
  
```

```

Comando : Comando_Select
         | Comando_Create
         | Comando_Insert
         | Comando_Delete
         | Comando_Drop
         | Comando_CreateIndex
  
```

```
Comando_Select : SELECT Colunas FROM Tabelas Condicoes
```

```
Comando_Create : CREATE TABLE Tabela '(' DadosTabela ')'
```

```
Comando_Insert : INSERT INTO Tabela '(' Colunas ') VALUES '(' Valores ')'
```

```
Comando_Delete: DELETE FROM Tabela Condicoes
```

```
Comando_Drop: DROP TABLE Tabela
```

Conando_CreateIndex: CREATE INDEX Index ON Tabela '(' Colunas ')'

Tabelas : Tabela
| Tabelas ',' Tabela

Colunas : Coluna
| Colunas ',' Coluna

Valores : Val
| Valores ',' Val

DadosTabela: LinhaTabela
| DadosTabela ',' LinhaTabela

LinhaTabela: Coluna data_type Restricao

Restricao: restr
| &

Condicoes: WHERE Condicao
| Condicoes AND Condicao
| &

Condicao: Coluna opd Val

Val : num
| str

Tabela : var

Coluna : var

Index: str

}

2.2.2 Analisador Léxico

Para identificarmos num texto os comandos SQL, usámos Expressões Regulares para reconhecer as palavras reservadas (Terminais), e através do ficheiro *y.tab* fazemos a ligação com o nosso Tradutor de Sintaxe. Apresentamos de seguida as Expressões Regulares usadas no nosso Analisador Léxico para reconhecer os Terminais:

```
sel      [Ss][Ee][Ll][Ee][Cc][Tt]

from     [Ff][Rr][Oo][Mm]

where    [Ww][Hh][Ee][Rr][Ee]

create   [Cc][Rr][Ee][Aa][Tt][Ee]

table    [Tt][Aa][Bb][Ll][Ee]

insert   [Ii][Nn][Ss][Ee][Rr][Tt]

into     [Ii][Nn][Tt][Oo]

values   [Vv][Aa][Ll][Uu][Ee][Ss]

prim     [Pp][Rr][Ii][Mm][Aa][Rr][Yy]

key      [Kk][Ee][Yy]

foreign  [Ff][Oo][Rr][Ee][Ii][Gg][Nn]

refs     [Rr][Ee][Ff][Ee][Rr][Ee][Nn][Cc][Ee][Ss]

and      [Aa][Nn][Dd]

or       [Oo][Rr]

variavel [a-zA-Z]+|""

inPlicas \'[^\']*\'

inAspas  \"[^\"]*\"

numeros  [0-9]+
```

datatype [Cc][Hh][Aa][Rr] | [Vv][Aa][Rr][Cc][Hh][Aa][Rr]
| [Nn][Uu][Mm][Bb][Ee][Rr] | [Ii][Nn][Tt]

op "=" | "<" | ">" | "<=" | ">=" | "<>"

like [Ll][Ii][Kk][Ee]

3 Conclusão e Trabalho Futuro

Este projecto complementa as eficientes Berkeley DB, que não tinham capacidade de lidar com a popular linguagem SQL, tornando-se assim uma Base de Dados ainda mais prática e melhor, no que a interacção com o utilizador diz respeito.

Após a realização deste projecto, pensamos ter atingido todos os objectivos propostos da melhor maneira, quer no que diz respeito ao manuseamento com as Berkeley DB - funções C que lidam com este tipo de base de dados para efectuar as operações necessárias, quer em relação ao Parser entre a linguagem SQL e as referidas funções C - Gramática Tradutora para os comandos SQL.

Em termos de Trabalho Futuro, propomos o melhoramento da Gramática Tradutora, de modo a lidar de forma completa com a linguagem SQL. Nomeadamente acrescentar os restantes tipos de dados das variáveis (floats, char, etc), e incluir os restantes comandos SQL, tal como aprofundar os já existentes. No entanto o nosso projecto inclui todos os aspectos mais relevantes para a interacção entre as Berkeley DB e a referida linguagem.

Bibliografia

- “Yacc: Yet Another Compiler-Compiler” - Stephen C. Johnson
- “Uma não tão pequena introdução ao LATEX 2” - Tradução portuguesa por Alberto Simões.

Referências WWW

01 **www.oracle.com**

Página principal da Oracle. Toda a informação correspondente á Berkeley DB foi retirada deste site.

Lista de Acrónimos

DB Database

BD Base de Datos

SQL Structured Query Language