Instituto Nacional de Pesquisas Espaciais (INPE)
Centro Regional de Natal (CRN)

Aspectos Temporais em PostgreSQL

Relatório de Iniciação Científica

Francisco Ricardo Batista Cardoso
Natal-RN, 13 de Julho de 2001

Apresentação

Este relatório é a documentação oficial do projeto: **Aspectos temporais em PostgreSQL**, que se encontra em desenvolvimento no Centro Regional de Natal do Instituto Nacional de Pesquisas Espaciais (INPE/CRN). O projeto é financiado pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPQ). O responsável pelo projeto no INPE de São José dos Campos é o Prof. Dr. Ijar Milagre da Fonseca (*ijar@dem.inpe.br*), a orientação do projeto está a cargo do Msc. Romualdo Alves Pereira (*romualdo@crn.inpe.br*) e o bolsista em trabalho é Francisco Ricardo Batista Cardoso (*fricardo@crn.inpe.br*). O projeto visa implementar em um sistema gerenciador de bancos de dados um suporte a conceitos temporais.

Motivação

Convencionalmente, os dados encontrados nos bancos de dados são estáticos. Isto quer dizer que a informação só está disponível no seu estado corrente, ou seja, não sabemos como a informação relativa ao dado variou com o tempo. Para diversas aplicações, é tão importante saber a história do dado armazenado quanto o seu valor no instante de acesso. Os bancos de dados temporais propõem suporte a este tratamento temporal da informação. Nossa motivação, ao desenvolver este trabalho é contribuir com os estudos na área, obtendo uma implementação final num sistema de gerenciamento de bancos de dados (SGBD) capaz de ser utilizado em aplicações científicas práticas.

Resumo

O primeiro capítulo deste relatório é a respeito de bancos de dados temporais. Ele está dividido em três seções: a primeira, sobre a Structured Query Language (SQL); a segunda, discorre sobre os Bancos de Dados Temporais e a terceira, sobre o PostgreSQL. Sobre o SQL, veremos o modelo relacional de dados, sua estruturação e operações e a sintaxe dos principais comandos SQL. A seção a respeito de bancos de dados temporais explica o que é um banco de dados temporal e os modelos temporais existentes. A última seção do capítulo 1 trata do Sistema Gerenciador de Bancos de Dados (SGBD) PostgreSQL, destacando os seus aspectos mais relevantes (um tratamento pormenorizado sobre o PostgreSQL pode ser encontrado na documentação do sistema que vem com as mais diversas distribuições do sistema operacional Linux e também na página oficial do PostgreSQL na internet `http://www.postgresql.org/site.html` e o espelho no Brasil[1]).

O segundo capítulo aborda a implementação do projeto. Para isso, vemos as ferramentas lex e yacc do Unix (no Linux temos o flex e o bison como ferramentas respectivamente equivalentes às citadas), que são fundamentais na implementação do PostgreSQL. A seguir temos os aspectos práticos de implementação que são vistos em duas partes: o tratamento léxico e sintático dado a uma consulta temporal e o processamento desta para obter uma saída que de fato trate os dados de maneira temporal. O capítulo final trata-se de uma conclusão que aborda o que já foi feito e também o que ainda poderá ser feito em uma continuação deste trabalho.

[1] dbExperts: `http://pgsql.dbexperts.com.br/users-lounge/docs`

Sumário

Lista de Figuras

Lista de Tabelas

Símbolos e Abreviaturas

ANSI American National Standards Institute

BCDM Bitemporal Conceptual Data Model

DB2 Sistema gerenciador de bancos de dados da IBM

DDL Data Definition Language

DML Data Manipulation Language

gcc GNU C Compiler

HRDM Historical Relational Data Model

HSQL Historical Structured Query Language

INET Tipo de socket no Unix

ISO International Standards Organization

lex Gerador de analisador léxico no Unix (flex no Linux)

Postquel Postgres Query Language

SGBD Sistema de Gerenciamento de Bancos de Dados

SQL Structured Query Language

SSL Secure Socket Layer

Tquel Temporal Query Language

TSQL Temporal Structured Query Language

TSQL2 Temporal Structured Query Language 2

UNIX Tipo de socket no Unix

yacc Gerador de analisador sintático do Unix (bison no Linux)

$\forall$ Para todo

$\vee$ OU lógico

$\wedge$ E lógico

$\Leftrightarrow$ Se e somente se

$\neg$ NÃO lógico

Capítulo 1

Bancos de Dados Temporais

O tempo está intimamente relacionado com os sistemas de informação. Os calendários que temos, bem como o tempo relacionado à criação e existência de arquivos mostram o quanto o tempo assume um papel importante nos sistemas mencionados. Apesar disto, o tratamento do tempo como parte da informação armazenada ainda não é satisfatório. Por este motivo e pela importância crucial da manipulação dos dados temporais em aplicações práticas de forma precisa e adequada, os estudos a respeito de bancos de dados temporais tem crescido. Neste capítulo abordaremos como aspectos temporais são tratados em bases de dados. Iniciaremos com um estudo breve a respeito da conhecida linguagem SQL, abordando a seguir bancos de dados temporais e finalmente o PostgreSQL como um sistema gerenciador de bancos de dados.

1.1 A Linguagem SQL

1.1.1 Introdução

Nesta seção veremos as bases sobre a linguagem SQL (Structured Query Language). O SQL é implementado como linguagem de pesquisa em sistemas gerenciadores de bancos de dados relacionais (assunto que veremos na seção seguinte). Sua história se iniciou no laboratório da IBM em San Jose, Califórnia nos anos 70. Inicialmente era chamada "sequel" e foi desenvolvida para DB2 (sistema relacional de bancos de dados da IBM). O SQL torna possível a abordagem relacional dos bancos de dados. Provavelmente por isso, se popularizou, tornando-se o padrão de fato para bancos de dados. Os padrões correntemente utilizados na indústria são estabelecidos pela ANSI (American National Standards Institute) e pela ISO (International Standards Organization).

1.1.2 Modelo de Dados Relacionais

Num banco de dados, temos dados do mundo real armazenados em sistemas computadorizados. Obviamente, é fundamental saber como representar estes dados de maneira que os computadores possam lidar com eles. Precisamos, então, de um *modelo de dados* capaz de representar os dados e também manipulá-los. Assim, podemos restringir nossos dados de entrada para que eles sejam estruturados conforme o estabelecido no banco de dados. A especificação da estruturação dos dados é feita pelo uso de linguagens de definições de dados (DDL[1]) e como eles serão manipulados pela linguagem de manipulação de dados (DML [2]).

As DMLs proveêm duas capacidades fundamentais para um banco de dados: a de *operação* (querying) que define como extrair os dados do banco; e a de *atualização* (updating) para que o estado do banco de dados seja mudado com as operações realizadas.

Os primeiros estudos sobre modelos de dados relacionais estão no artigo de Codd[3]. Neste artigo, Codd estabeleceu relações como as estruturas de organização de dados, uma álgebra para operações, mas não estabeleceu mecanismos para exprimir atualizações e restrições. Os artigos subseqüentes de Codd introduziram como primeiras restrições de integridade as dependências funcionais, bem como estruturou linguagens de operações em torno do cálculo de predicados. Os mais recentes avanços permitiram a obtenção de linguagens de operação complexas baseadas na álgebra e no cálculo de que incluem suporte a diversos operadores (lógicos, de atualização, aritméticos, etc.), além de diversos tipos de restrição de integridade.

Como relações são estruturas de dados muito simples, o modelo relacional pode ser facilmente entendido, isto torna os bancos de dados relacionais mais acessível aos usuários. Outra vantagem também é a maior facilidade de conceber linguagens que manipulem relações de maneira eficiente. Isto torna as linguagens de bancos relacionais simples de usar e poderosas nas aplicações em que são utilizadas. Estas facilidades levaram o modelo relacional a ser amplamente aceito e se tornar o padrão de fato no segmento de bancos de dados.

1.1.3 A estruturação do modelo relacional

A estrutura do modelo relacional baseia-se em *relações*. Uma relação é um subconjunto de um produto cartesiano. Seja A e B conjuntos, então o produto cartesiano de A por B (representa-se AxB) é o conjunto dos pares ordenados (a, b) tal que $a \in A$ e $b \in B$, ou seja:

[1] DDL: Data Definition Language

[2] DML: Data Manipulation Language

[3] *A relational model of data for large shared data banks*, 1970.

$$A \times B = \{(a, b) \mid a \in A,\ b \in B\}.$$

Desta forma, sendo $A = \{a_1, a_2, \ldots, a_n\}$ e $B = \{b_1, b_2, \ldots, b_m\}$ dois conjuntos, então o produto cartesiano de A por B é:

$A \times B = \{(a_1, b_1), (a_1, b_2), \ldots, (a_1, b_m), (a_2, b_1), (a_2, b_2), \ldots, (a_2, b_m), \ldots, (a_n, b_m)\}$ onde $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n \in \mathbb{R}$.

Uma *relação*, por definição, *é qualquer subconjunto de um produto cartesiano*. De acordo com o exemplo acima $\{(a_1, b_1), (a_n, b_m)\}$ e $\{(a_2, b_2)\}$ são algumas das 2^{nm} relações possíveis. Na terminologia usada no modelo relacional, os conjuntos A e B são chamados *domínios*, cada membro de uma relação é chamado pelo nome *tupla*.

As relações podem ser visualizadas como uma tabela, onde as colunas são denominadas **tuplas** e os cabeçalhos das colunas **atributos**. As tuplas e atributos formam o chamado *esquema relacional*. Um esquema relacional R denotado por $R(A_1, A_2, \ldots, A_n)$ é formado de uma relação R e uma lista de atributos $A_1, A_2, \ldots, A_n$ sendo cada atributo A_i tomado num domínio D do esquema relacional R. Um esquema relacional descreve uma relação; R é chamado **nome** da relação e o número de atributos n da relação é chamado grau da relação. A seguir temos a tabela ESTUDANTES, onde temos listados alguns alunos matriculados na disciplina *Álgebra*.

ESTUDANTES

Número	Matrícula	Aluno	Curso
1	9822343	André Castro	Matemática
2	9843432	Beatriz Silva	Ciências da Computação
3	9734351	Carlos Pires	Matemática
4	9721254	Filipe Costa	Engenharia de Computação
5	9882457	Mateus Júnior	Engenharia Elétrica
6	9632381	Paulo Silva	Física
7	9845672	Pedro Marques	Geologia

Tabela 1.1: Tabela ESTUDANTES com tuplas e atributos

Um domínio é um conjunto de valores atômicos, ou seja, cada valor no domínio é indivisível da mesma maneira que o modelo relacional. Um exemplo de domínio é o número_de_telefones_no_Brasil, o qual se constitui no conjunto de números de telefone de 10 dígitos válidos no território brasileiro. Cada domínio possui um tipo de dado e um formato para expressá-lo. O domínio número_de_telefones_no_Brasil, por exemplo, pode ser criado como uma cadeia de caracteres da forma (nnnnn)nnn nnn, sendo cada n um caracter numérico (um

dígito decimal) e onde os primeiros cinco números entre parênteses indentificam todos os telefones de uma determinada área. Um domínio, então pode ser plenamente caracterizado por um nome, tipo de dado e formato.

O esquema relacional da tabela 1.1 encontra-se abaixo:

ESTUDANTES(Número, Matrícula, Aluno, Curso).

Neste esquema de quarto grau, ESTUDANTES é o nome da relação, que se caracteriza por quatro atributos: o número que identifica os alunos matriculados em Álgebra, a matrícula na instituição de ensino, o nome pessoal e do curso do estudante em questão.

1.1.4 Operações no modelo relacional

É natural que, além de armazenar os dados em tabelas precisemos ainda manipulá-los. Por isso necessitamos de operações que podem manipular os dados armazenados. Com este propósito temos no modelo de dados relacional duas notações distintas: o **cálculo relacional** e a **álgebra relacional**. O *cálculo relacional* provê uma *notação lógica* onde as operações são expressas por formulação lógica das restrições que as tuplas devem satisfazer. O *álgebra relacional* é uma *notação algébrica* onde operadores algébricos são utilizados para realizar as operações desejadas. Veremos a seguir algo sobre cada abordagem.

A Álgebra Relacional

E.F.Codd introduziu a álgebra relacional em 1972. Ela define um conjunto de operações, dentre as quais as principais são:

SELECT(σ): extrai *tuplas* que satisfazem uma restrição dada para relação. Escrevendo matematicamente, seja R uma tabela que possui um atributo A, então $\sigma_{A=a}(R) = \{t \mid t(A) = a\}$, onde t denota uma tupla de R e $t(A)$ é o valor de atributo A da tupla t.

PROJECT(π): extrai *atributos*, isto é, colunas especificadas da relação. Se R é uma relação que possui como um dos seus atributos A, então $\pi_A(R) = \{t(A) \mid t\}$, sendo $t(A)$ o valor de atributo A de tupla t.

PRODUCT(x): faz o produto cartesiano de duas relações dadas. Seja R e S duas relações, sendo a cardinalidade [1](ou aridade) de R, k_1 e de S, k_2, então o produto R x S é o conjunto de todas as $k_1 + k_2$-tuplas onde as primeiras k_1 são tuplas em R e as últimas k_2 constituem tuplas em S.

[1]número de tuplas da relação

UNION(∪): realiza a união de duas relações R e S dadas. Ou seja, se R e S possuem a mesma cardinalidade, então $R \cup S = \{t \mid t \in S \textbf{ ou } t \in S\}$.

INTERSECTION(∩): faz a intersecção de duas relações. Seja R e S duas relações, então $R \cap S = \{t \mid t \in A \textbf{ e } t \in B\}$.

DIFFERENCE(−): faz uma operação de diferença similar a teoria dos conjuntos. Seja R e S relações, então $R - S = \{t \mid t \in R \textbf{ e } t \notin S\}$.

JOIN(⋈): conecta duas tabelas com atributos comuns. Seja R uma tabela com atributos A, B e C e S com atributos C, D e E. Isto quer dizer que C é um atributo comum entre R e S. Então $R \bowtie S = \pi_{R.A,R.B,R.C,S.D,S.E}(\sigma_{R.C=S.C}($ R x $S))$. Para fazer uma junção, devemos primeiro fazer o produto cartesiano de R e S. Depois, selecionamos as tuplas de R e S que possuem o atributo C em comum e por fim utilizamos a operação de projeção (project) para eliminar as redundâncias (repetição da coluna C).

O Cálculo Relacional

O cálculo relacional baseia-se na lógica de primeira ordem. Existem duas variantes no cálculo relacional: O *cálculo relacional de domínio* e o cálculo relacional de tuplas. Aqui vamos nos deter um pouco no cálculo relacional de tuplas (CRT).

O *cálculo relacional de tuplas* faz operações da seguinte forma: $x(A) \mid g(x)$, onde x é uma variável tupla, A é um conjunto de atributos e g é uma fórmula. O resultado é o conjunto de tuplas t que satisfaz g.

A álgebra relacional e o cálculo relacional possuem a mesma capacidade de expressão. Uma operação feita através de um pode igualmente ser feita pelo outro. E.F.Codd mostrou isso em 1972 através do algoritmo de redução de Codd, no qual uma expressão do cálculo relacional pode ser transformada a uma semanticamente equivalente na notação algébrica.

1.1.5 O SQL

O SQL, como a maior parte das linguagens relacionais, baseia-se no cálculo relacional de tuplas. Toda query que é formulada usando o cálculo relacional (ou alternativamente, a álgebra relacional), pode também ser elaborada utilizando-se o SQL. No entanto, ela também possui alguns recursos que não pertencem a nenhuma das duas notações vistas na seção anterior, como por exemplo:

- comandos para inserção, deleção e modificação de dados;

- capacidade algébrica que permite operações de soma, subtração, multiplicação, divisão, além de comparações tais como maior, menor. Assim podemos, por exemplo, somar aos valores de atributos números desejados, bem como comparar tais valores nas colunas entre si;
- funções agregadas: existem funções que fazem média (average), encontram o valor máximo de uma coluna (max) e a soma de valores dos elementos de uma coluna;
- facilidades para atribuir e imprimir queries;

A forma completa do comando SQL está abaixo:

```
SELECT [ALL|DISTINCT]
       { * | expr_1 [AS c_alias_1] [, ...
             [, expr_k [AS c_alias_k]]]}
FROM nome_tabela_1 [t_alias_1]
     [, ... [, nome_tabela_n [t_alias_n]]]
[WHERE condição]
[GROUP BY nome_atrib_i
       [,... [, nome_atrib_j]] [HAVING condição]]
[{UNION [ALL] | INTERSECT | EXCEPT} SELECT ...]
[ORDER BY nome_atrib_i [ASC|DESC]
       [, ... [, nome_atrib_j [ASC|DESC]]]];
```

, onde os termos entre colchetes são opcionais e os entre chaves são itens repetitivos opcionais e a barra vertical indica que podemos usar uma opção *ou* outra.

O comando SELECT faz a recuperação das colunas de nomes expr_1, ..., expr_kna tabela especificada na cláusula FROM. O * indica que serão recuperadas todas as colunas da tabela e o uso de DISTINCT elimina repetições de valores nas colunas. Um comando SQL deve ter pelo menos uma cláusula SELECT e uma cláusula FROM. Deve-se perceber também que o SELECT não corresponde a "seleção"da álgebra relacional, mas a "projeção".

A palavra chave WHERE permite estabelecer condições as quais os itens recuperados devem obedecer. GROUP BY faz agregação por grupo e permite rodar a função agregada no comando SELECT para cada agrupamento das colunas que seguem a cláusula GROUP BY. A cláusula HAVING é usada para considerar apenas os grupos por ela especificados e deve envolver funções agregadas.

A existência da palavra-chave SELECT no meio do comando SQL determina a existência de *subqueries*. Cada SELECT define uma subquery. As subqueries devem ser escritas entre parênteses. As cláusulas UNION e INTERSECT fazem a união e a intersecção, respectivamente, entre as tuplas de duas subqueries. O comando ORDER BY permite ordenar os resultados encontrados.

Outros comandos importantes do SQL estão abaixo:

CREATE TABLE: é um comando para definição de dados. Permite criar uma nova tabela. Sua sintaxe geral está abaixo:

```
CREATE TABLE nome_tabela
          (nome_atrib_1 tipo_do_atrib_1
           [, nome_atrib_2 tipo_do_atrib_2
           [, ...]]);
```

Os tipos de dados que podem ser definidos são:

- INTEGER: palavra binária inteira com 31 bits de precisão;
- SMALLINT: meia palavra inteira com 15 bits de precisão;
- DECIMAL(p,[q]): número decimal de p dígitos de precisão com q deles do lado direito do ponto decimal. O parâmetro q, se omitido é tomado como zero;
- FLOAT: número de ponto flutuante com sinal;
- CHAR(n): cadeia de caracteres de comprimento fixo n;
- VARCHAR(n): cadeia de caracteres de comprimento variável máximo n;

CREATE INDEX: índices são usados para acesso mais rápido para uma relação. Se uma relação R possuir um índice sobre o atributo A, então podemos selecionar todas as tuplas nas quais $t(A) = a$ em tempo proporcional a t ao invés de proporcional a R. A sintaxe deste comando é:

```
CREATE INDEX nome_índice
ON nome_tabela ( nome_atrib );
```

CREATE VIEW: cria uma tabela "virtual", isto é, que não existe fisicamente na base de dados mas o usuário pode utilizar como se de fato, existisse. A sintaxe do comando é:

```
CREATE VIEW nome_view
```

```
AS select_stmt
```

sendo select_stmt um comando SELECT válido em SQL.

DROP TABLE, DROP INDEX, DROP VIEW: estes comandos são utilizados para destruir uma tabela, um índice e uma tabela virtual. A sintaxe do comando encontra-se abaixo:

```
DROP TABLE nome_tabela;
DROP INDEX nome_índice;
DROP VIEW nome_view;
```

INSERT INTO: serve para encher com tuplas uma tabela criada através do comando CREATE TABLE. Sintaxe:

```
INSERT INTO nome_tabela (nome_atrib_1
                [, nome_atrib_2 [,...]])
VALUES (valor_atrib_1
   [, valor_atrib_2 [, ...]]);
```

UPDATE: comando usado para mudar os valores dos atributos de uma ou mais tuplas. Sua sintaxe é:

```
UPDATE nome_tabela
SET nome_atrib_1 = valor_1
    [, ... [, nome_atrib_k = valor_k]]
WHERE condição;
```

DELETE FROM: apaga uma tupla da tabela desejada. Sintaxe:

```
DELETE FROM nome_tabela
WHERE condição;
```

1.2 Bancos de dados temporais

1.2.1 Conceitos Básicos

Embora não tenhamos visto ainda nada a respeito do tempo em sistemas de informação, para a maioria dos sistemas é importante o modelamento do tempo em que eventos relevantes ocorrem e também armazenar o contexto temporal no qual o evento ocorre de forma que eventos que aconteçam posteriormente não afetem este contexto. Em bancos de dados temporais este conceito é fundamental e denomina-se *completeza temporal.*

Num banco de dados relacional comum, os dados como vimos são armazenados em tabelas. Uma tabela possui duas dimensões: as instâncias (linhas) e os atributos (colunas). Num banco de dados temporal, no entanto, temos de acrescentar mais uma dimensão: o tempo, já que os dados possuem informações temporais associadas a eles. O tempo em que os dados foram definidos no banco de dados chama-se *tempo de transação.* Quando a informação modela perfeitamente a realidade temos o *tempo de validade.* Num banco de dados comum, o tempo de transação e de validade não são diferenciados, ou seja, os dados são efetivos tão logo sejam inseridos na base. Além dessas propriedades dos dados em si, o usuário pode definir algumas propriedades temporais para as informações armazenadas e elas devem ser manipuladas pelos aplicativos que rodam no sistema de computador.

Para um dado ser perfeitamente definido, temos a necessidade dele incluir o tempo de transação e o de validade. Caso os dois sejam inclusos, o sistema terá 4 dimensões (tupla-instância-tempo de transação-tempo de validade). Outra possibilidade é que o rótulo possua apenas o tempo de validade ou apenas o tempo de transação. A escolha de um rótulo bitemporal ou não depende da aplicação em questão.

Em relação a *ordem temporal*, o tempo pode ser: linearmente ordenado, ramificado ou circular. Como o nome diz quando o tempo é linearmente ordenado, entre quaisquer dois pontos t_1 e t_2 no tempo, ou t_1 está antes de t_2 ou vice-versa. Em símbolos: $\forall t_1,\ t_2 : t_1,\ t_2 \in T \wedge t_1 \neq t_2 \Rightarrow t_1$ antes $t_2 \vee t_2$ antes t_1, onde T é o conjunto (não vazio) de todos os pontos do tempo. A relação de antecedência deve satisfazer a três propriedades principais:

Não reflexibilidade: $\forall t : t \in T \Rightarrow \neg(t$ antes $t)$;

Transitividade: $\forall t_1,\ t_2,\ t_3 : t_1,\ t_2,\ t_3 \in T \wedge\ t_1$ antes $t_2 \wedge t_2$ antes $t_3 \Rightarrow t_1$ antes t_3;

Assimetria: $\forall t_1,\ t_2 : t_1, t_2 \in T\ \wedge t_1$ antes $t_2 \Rightarrow \neg(t_1$ antes $t_2)$.

No tempo ramificado, dois ou mais pontos diferentes podem ser sucessores ou antecessores imediatos de um determinado ponto no tempo. No caso de tais pontos sucederem o ponto mencionado, temos uma ramificação no passado e se forem antecessores teremos uma ramificação no futuro. No tempo circular os instantes são recorrentes no tempo. Ou seja, dado t_1 como um ponto no tempo, então existe t_i, tal que $t_i = t_1$, sendo $i = 2c$, onde c é uma quantidade constante. O tempo circular serve para modelar eventos ou processos que se repetem no tempo.

Outro conceito básico importante é a *variação temporal*. Neste caso, o tempo pode ser contínuo e discreto. O tempo contínuo pode ser dividido por mais que o intervalo de tempo seja reduzido. Já o tempo discreto é formado por uma sucessão de intervalos de tempo de comprimento fixo e indivisíveis denominados *chronons*. O tamanho de um chronon é fixado durante a implementação do modelo de dados e de acordo com a aplicação a que se destina, além de ser único no sistema. O chronon facilita a implementação do modelo de dados.

Veremos agora os tipos de dados temporais:

evento: algo que ocorre em um instante determinado de tempo;

ponto no tempo (instante de tempo): corresponde a 1 chronon na linha de tempo. O *instante atual* (*now*) move-se constantemente na linha de tempo e define o que é presente, passado e futuro. Se a variação temporal é contínua, os instantes de tempo tem duração infinitesimal a caso a variação seja discreta, os eventos ocorrem no chronon correspondente eos instantes de tempo são discretos;

intervalo temporal: tempo decorrido entre dois instantes de tempo. Um intervalo temporal é representado pela notação $[li,\ ls]$, sendo li o limite inferior do intervalo, ls o limite superior e $li < ls$. Seja um intervalo de tempo I, com $I \subseteq T$ então o primeiro elemento de I (first(I)) e o último elemento de I é (last(I)) são tais que:

$$\text{first}(I)=t \mid \forall t' \in I \Rightarrow t \text{ antes } t' \ \vee t' = t$$
$$\text{last}(I)=t \mid \forall t' \in I \Rightarrow t' \text{ antes } t \ \vee t' = t$$

para um conjunto totalmente ordenado

elemento temporal: união finita de intervalos temporais. Sejam I_1 e I_2 dois intervalos temporais, então $I_1 \cup I_2$ é um elemento temporal. A definição de intervalo temporal engloba as de instante temporal e intervalo temporal e facilita a elaboraç ao de consultas ;

De acordo com o exposto acima, o tempo de validade pode ser representado através de um ponto no tempo, por um intervalo temporal, ou ainda por dois pontos no tempo, indicando a existência de um período de validade.

1.2.2 Tipos e Definição

Um banco de dados temporais armazena informações que variam com o tempo, e dados passados, presentes futuros. Desta forma, faz sentido se falar de *estado atual de um banco de dados* que é a sua configuração nu momento determinado de tempo. De acordo com a maneira com que tratam temporalmente os dados, existem os seguintes tipos de bancos de dados: instantâneo (snapshot), de tempo de transação (rollback), de tempo de validade (histórico), bitemporais (temporais) e os multitemporais. Vamos nos deter nos bitemporais.

Algumas das características dos *bancos de dados bitemporais* são: os dados são armazenados com um rótulo que contêm o tempo de transação e o tempo de validade, toda a hosória do banco de dados fica armazenada, todos os estados pelos quais o banco passou podem ser acessadas, o estado atual é constituído dos valores atualmente válidos e podem ser definidos valores futuros, além da possibilidade de recuperação de valores atualmente válidos, valores passados válidos, valores passados modificados devido ao fato de não serem válidos e estimativas para o futuro.

Um banco de dados bitemporal necessita de uma linguagem de definição de dados temporal (pois os dados são temporais), uma lingugem de consulta temporal, linguagem de manipulação de dados temporal e restrições temporais. Há uma aplicação comercial temporal denominada TimeDB, na internet [2]. Ele é bitemporal e a linguagem que usa é um subconjunto extendido do SQL (o TSQL2), desenvolvida por Michael Boehlen, Christian Jensen, Richard Snodgrass e Andreas Steiner. Este último desenvolveu o TimeDB como uma aplicação para o banco de dados Oracle.

Quanto ao *modelo de dados*, necessitamos de uma linguagem de consulta adequada com o intuito de fazer consultas temporais de informações, representação das propriedades temporais, objetos a serem modelados temporalmente, uma maneira de atualizar os objetos armazenados, restrições de integridade que possam ser utilizadas na abordagem em questão. O modelo adotado deve permitir a modelagem e consulta de dados em qualquer instante ou em dois instantes de tempo dstintos de tempo e ainda permitir que tuplas diferentes possuam valores temporais associados de diferentes naturezas. A linguagem de consulta, por sua vez deve permitir a realização de operações sobre os rótulos, o retorno de objetos temporais análogos ao que atua e ainda o agrupamento de objetos segundo algum critério específico desejado.

No caso de uma consulta, um banco de dados temporal ideal retornaria uma saída compatível a operação de seleção feita. Assim caso fosse feita uma seleção temporal, teríamos uma saída temporal, no caso de uma seleção temporal uma saída temporal e com a ocorrência de uma seleção mista obteríamos igualmente

[2] endereço do banco de dados TimeDB na internet: `http://www.timeconsult.com`

um retorno misto.

Alguns problemas que encontramos em consultas relacionais são dificuldade de acesso quando a quantidade de informação armazenada é muito grande. Normalmente há necessidade de novos métodos de ordenação, pois os métodos tradicionais podem ser utilizados apenas quando é possível realizar algum tipo de ordenação. Outro problema é que existe a necessidade de lidar com informações incompletas, quando, por exemplo, um determinado dado nãoexiste em alguns pontos do tempo e quando eventos ocorrem em um tempo indeterminado.

Os modelos de dados temporais existentes normalmente são extensões de modelos tradicionais. Assim, temos os modelos:

- relacional;
- entidade relacionamento;
- orientado a objetos.

O modelo relacional temporal é uma extensão do modelo relacional já visto neste capítulo,

1.2.3 Os Modelos Relacionais Temporais

No modelo relacional temporal os bancos de dados trabalham com relações, as quais são constituídas por tuplas. A estes três elementos são associadas informações temporais no modelo. As relações possuem um início, uma duração, um término, podem voltar a existir depois de um tempo de inexistência, dentre outras possibilidades. Ao longo do tempo, surgiram várias abordagens deste modelo:

- Modelo de Dados Relacional Histórico (HRDM), desenvolvido por Clifford em 1987
- SQL Histórico (HSQL), de autoria de Sarda (1990);
- Linguagem de Consulta Temporal (Tquel), proposta por R. Snodgrass em 1987;
- Modelo Relacional Temporal, que foi explicitado por Navathe e Ahmed em 1987;
- SQL Temporal 2 (TSQL2), de 1995.

O Modelo Relacional Temporal

O modelo relacional temporal de Navathe e Ahmed, é a base da linguagem *TSQL* (Temporal SQL). Nele há coexistência de relações temporais e não temporais (estáticas) no mesmo banco de dados. O tempo é adicionado as tuplas e estas variam sincronamente de acordo com os atributos. As relações temporais possuem tempos de início e fim associados a elas.

Alguns conceitos importantes associados ao modelo temporal relacional são os de tempo de validade e elemento temporal. O elemento temporal pode ser um ponto no tempo $[t, t]$ ou intervalo no tempo $[t_i, t_f]$, sendo t_i o tempo inicial do intervalo e t_f o tempo final do intervalo. Neste caso, a validade é associada ao dado durante todo o intervalo temporal. Neste modelo, qualquer relação temporal é constituída de tuplas, como no modelo relacional tradicional, no entanto, temos o acréscimo do tempo de início e do tempo de fim as tuplas da relação.

A linguagem TSQL

O TSQL (Temporal SQL) é uma extensão temporal da linguagem SQL vista no capítulo 1. Desta forma TSQL possui os mesmos comandos do SQL padrão, como SELECT, WHERE, GROUP BY, etc. Como no modelo relacional temporal temos relações estáticas e temporais, existe o uso de SQL e TSQL para aqueles dois tipos de relação, respectivamente. A principal diferenca entre estas duas linguagens é a existência, no TSQL, da cláusula WHEN. Semelhantemente a cláusula WHERE existente no SQL clássico, WHEN também é condicional. Para expressão de tais condições, temos definido um conjunto de operadores temporais como se pode ver na tabela 1.2, onde a, b, c, $d \in T$ e $[a, d]$ é um intervalo (denotado pela palavra-

Operador	Definição
BEFORE	$[a, b]$ BEFORE $[c, d] \Leftrightarrow b < c$
AFTER	$[a, b]$ AFTER $[c, d] \Leftrightarrow a > d$
DURING	$[a, b]$ DURING $[c, d] \Leftrightarrow (a \geqslant c) \wedge (b \leqslant d)$
EQUIVALENT	$[a, b]$ EQUIVALENT $[c, d] \Leftrightarrow (a = c) \wedge (b = d)$
ADJACENT	$[a, b]$ ADJACENT $[c, d] \Leftrightarrow (c - b = 1) \vee (a - d = 1)$
OVERLAP	$[a, b]$ OVERLAP $[c, d] \Leftrightarrow (a \leqslant d) \wedge (c \geqslant b)$
FOLLOWS	$[a, b]$ FOLLOWS $[c, d] \Leftrightarrow a - d = 1$
PRECEDES	$[a, b]$ PRECEDES $[c, d] \Leftrightarrow c - b = 1$

Tabela 1.2: Operadores temporais no TSQL

chave INTERVAL no TSQL) .

Outra característica do TSQL que vale a pena destacar é que ele identifica diversos tipos de resposta como *null*, no caso do atributo ser nulo (o objeto não se

encontra disponível), das cláusulas WHEN e WHERE não terem sido satisfeitas e devido ao fato dos objetos não estarem disponíveis no intervalo considerado, possivelmente por causa de descontinuidades na história do objeto.

Depois de 20 anos de pesquisa em bancos de dados temporais, tendo muitas propostas de modelos e linguagens de consulta, em 1995, Snodgrass publicou o livro *The TSQL2 Temporal Query Language*, lançando o Temporal Structured Query Language 2 (*TSQL2*) como uma linguagem para uniformizar e universalizar os muitos modelos e linguagens existentes. Algumas das principais inovações do TSQL2 são:

- múltiplas granularidades, tais como semestre, semana, estações do ano (no SQL existia apenas year, month,day, hour, minute, second);
- ao invés da representação dia/mês/ano do SQL, permite representar múltiplas representações, como nona semana de 2001;
- suporte a múltiplos calendários como o lunar e o fiscal;
- tratamento de tempo histórico.

O modelo de dados do TSQL é o BCDM (Bitemporal Conceptual Data Model). Este modelo trata o tempo como linear e discreto, possui capacidade de trabalhar com tempo de transação e tempo de validade e os tempo são associados as tuplas de forma que uma tupla t é dada por:

$$t = (atr_1, atr_2, \ldots, atr_n \mid r_b)$$

sendo atr_1, atr_2, ..., atr_n os atributos da tupla e r_b o rótulo bitemporal associado a ela.

1.3 O SGBD PostgreSQL

PostgreSQL é um sistema de gerenciamento de bancos de dados (SGBD) de código fonte aberto, objeto-relacional e que oferece facilidades adicionais em comparação com a maior parte dos SGBDs relacionais: classes, herança, tipos e funções, os quais permitem que o usuário possa extender o sistema facilmente e ainda restrições, triggers, regras e integridade de transação, o que lhe confere mais poder e flexibilidade. PostgreSQL suporta todos os construtores SQL, controle de concorrência multi-versão, tipos de dados definidos pelo usuário e interface para funções escritas em diversas linguagens (dentre as quais C, C++, Java, tcl, python e perl).

A origem do PostgreSQL remonta à década de 80, no projeto Postgres iniciado em 1986 na Universidade da Califórnia, em Berkeley. Postgres desenvolveu

uma linguagem de consulta própria denominada postquel (de Postgres query language) e seu objetivo era o de iniciar uma nova fase na concepção de bancos de dados. Hoje, Postgres é usado em várias aplicações e a empresa Informix[3] prossegue no seu desenvolvimento.

Em 1994, Andrew Yu e Jolly Chen incluíram um interpretador SQL no Postgres e o desenvolvimento posterior do sistema chegou ao Postgres95, cujo código fonte é completamente escrito em ANSI C incluindo ainda muitas mudanças internamente em relação ao Postgres anterior. Neste período o projeto foi aberto, obtendo ajuda de muitos desenvolvedores via internet. Em 1996 PostgreSQL foi escolhido como novo nome para o Postgres95 e atualmente é distribuído com uma licença Berkeley.

1.3.1 Arquitetura

A arquitetura do PostgreSQL é a de um único processo por servidor, como mostrado na figura a seguir e se constitui de três componentes principais:

- um processo supervisor denominado *postmaster*;
- a aplicação usuária cliente;
- um ou mais processos servidores (denominados "backends servers").

Existe um único postmaster rodando em todo o sistema. O postmaster é inicializado diretamente pelo administrador quando este instala os arquivos que compõem o postgreSQL. Ele roda com o uid[4] de um super-usuário denominado "postgres"(preferentemente não o superusuário Unix "root"), no lado do servidor esperando a conexão de um cliente via socket. Sua função é gerenciar um único host ou um conjunto de bancos de dados.

O usuário cliente que deseja acessar o banco de dados faz pedido de conexão a biblioteca (LIBPQ), que envia a requisição ao postmaster através da rede. O postmaster, então, cria um processo servidor e a comunicação cliente-servidor passa a ser feita sem a intervenção do postmaster. Este permanece rodando, esperando novos pedidos quando origina novos processos servidores para cuidar de cada conexão. Para cada conexão, portanto, temos um cliente e um servidor, mesmo no caso de um mesmo cliente fazer mais de um pedido de conexão, quando teremos um servidor para cada conexão. Na arquitetura do PostgreSQL, o servidor (backend server) e o postmaster rodam sempre na mesma máquina.

A aplicação mais comum de usuário é o terminal interativo Postgres *pgsql*. O pgsql permite fazer conexão com um banco de dados desde que o usuário que o

[3]Endereço da Informix na internet: `http://www.informix.com`

[4]"user identification": número que identifica um determinado usuário Unix no sistema

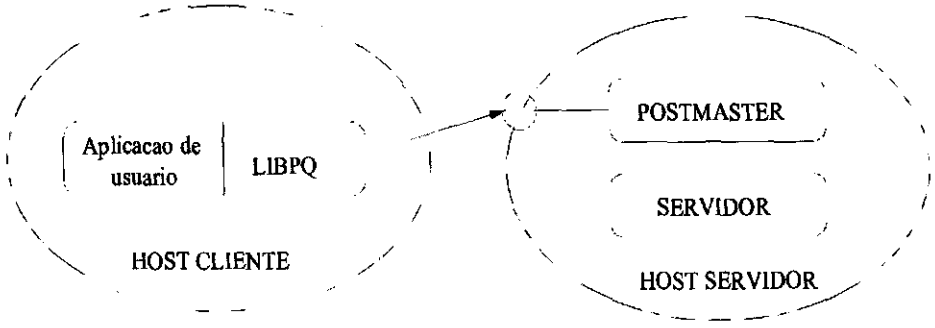

Figura 1.1: O mecanismo de conexão no PostgreSQL

utiliza tenha sido previamente cadastrado no sistema pelo superusuário "postgres". Uma vez conectado, o usuário pode fazer consultas a base de dados, operações nos bancos de dados e, alternativamente, um conjunto de meta-comandos e recursos de shell para fazer scripts e automatizar uma grande quantidade de tarefas.

1.3.2 O sistema

Agora veremos, como o sistema em si funciona. Muitos detalhes pouco relevantes, como a instalação e configuração não serão tratados aqui, uma vez que existe larga documentação para este propósito (ver referências bibliográficas [6], [8], [10]). Nosso objetivo primordial é saber como o sistema funciona, incluindo aí o mecanismo de conexão e o caminho percorrido por uma query até o resultado ser retornado para o programa de usuário.

Consideraremos o código fonte do PostgreSQL instalado em *path*, pois tal localização varia de acordo com a versão do sistema operacional utilizado e com a configuração do mesmo. A versão utilizada do gerenciador foi a 7.0.

O postmaster

O postmaster, que normalmente está em /usr/bin, aceita muitas opções na linha de comando (ver resultado do comando **man postmaster**), e seu código fonte está em path/backend/postmaster (arquivo postmaster.c). O postmaster cria um socket em /tmp, de nome .s.PGSQL.5432, para receber as conexões através dele. Internamente, o programa trabalha com sockets do tipo UNIX e INET e define o código da estrutura Backend, que se constitui numa estrutura de dados que permite a criação dos servidores para atendimento dos clientes. Como pode haver diversas conexões, o que torna necessário a criação de diversos servidores, existem listas para armazenar os backends ativos e também as portas de comunicação abertas,

cuja solicitação não foi ainda atendida.

O postmaster cria também segmentos de memória compartilhada e semáforos (ver resultado do comando Unix **ipcs**). Isto quer dizer que o postmaster divide a memória disponível com os backends gerados, no entanto, estes possuem preferência no acesso aos segmentos compartilhados da memória. Existe um número máximo de servidores que podem ser criados, definidos no script *configure*, embora a opção -N do comando postmaster possa ajustar tal número. Claramente, quanto maior a quantidade de servidores criada mais difícil a gerência de memória, mais memória é necessária e o desempenho do programa pode ser afetado.

O postmaster possui ainda suporte a conexões seguras, com Secure Socket Layer (SSL), havendo possibilidade, por exemplo, dele aceitar apenas conexões não locais seguras, o que pode ser definido através da opção -l do comando postmaster, desde que seja possível conectar-se via internet (opção -i). Na inicialização, o postmaster escreve um arquivo **postmaster_opts** e um arquivo **postmaster_pid**, realizado por um processo filho do postmaster (criado através da primitiva Unix **fork**). O arquivo de opções possui as opções utilizadas na inicialização e o poatmaster_pid o número do processo postmaster correntemente rodando.

Como já foi mencionado, PostgreSQL utiliza um mecanismo de um único servidor por conexão. A *multiplexação* de servidores backends (um servidor para várias conexões) ainda não é suportada. O postmaster possui também extensivo código para tratamento dos erros que podem ocorrer devido a erros do usuário ou a falhas do sistema quando o processo postmaster não consegue alocar os recursos necessários ao seu funcionamento.

O caminho de uma consulta

Veremos agora o caminho de uma query desde a sua formulação pelo cliente até quando este recebe o resultado da sua consulta. Todo este processo pode ser dividido nas partes abaixo descritas para seu melhor entendimento:

1. **Conexão** entre o cliente e o servidor, através do postmaster. Este processo já foi visto e inclusive o funcionamento do postmaster foi explicitado na subseção anterior.

2. **Envio de consultas**: uma vez aberta a conexão, o cliente (ver figura 1.1 na página 16) pode se conectar diretamente ao servidor. As consultas então são enviadas via rede através das portas de comunicação abertas entre o cliente e o servidor.

3. **Parser**: o código fonte desta fase está no diretório path/backend/parser. O parser realiza a *verificação sintática* e cria uma árvore de consulta (*query*

tree). A verificação sintática é feita através do executável gerado pelo programa do arquivo gram.y. Este arquivo é escrito pelo programador e é constituído de um conjunto de regras sintáticas para verificação da estrutura de entrada. O programa Unix *yacc*, então cria o analisador sintático (arquivo gram.c). A entrada do analisador de sintaxe é um conjunto de *identificadores* (tokens) retornados pelo analisador léxico scan.c. Este analisador é gerado do aplicativo *lex* do Unix que faz isto a partir do arquivo scan.l, que é escrito pelo programador do sistema. Os programas lex e yacc serão vistos no capítulo seguinte. De forma geral, o analisador léxico conhece um conjunto de palavras-chaves (ver keywords.c, no mesmo diretório), sinais de pontuação, etc, que ao serem encontrados na entrada são retornados para o analisador sintático. Este então analisa as regras que possui e então gera a saída. No nosso caso, a saída é uma árvore (chamada em gram.c, *savetree*). Nela estão as palavras chave da linguagem SQL, os operadores utilizados, as expressões de entrada, identificadores de tabelas, colunas, agregados, cláusulas, etc. A árvore, então é manipulada por diversos arquivos (iniciadas com a palavra parse), que tratam com os vários tipos de elementos encontrados na árvore. Por exemplo, parse_oper.c manipula expressões em operações, parse_expr.c manipula expressões e parse_clause cláusulas. Finalmente, a função analize.c transforma a árvore de parse em uma árvore de consulta (querytree), que será utiliizada nas fases posteriores.

4. **Reescrita**, a qual verifica a árvore de usuário e verifica se há nela alguma regra definida pelo usuário (a forma de definição de regras encontra-se na referência [10]). Se houver, o sistema então deverá reescrever a árvore. Basicamente, as regras definidas pelo usuário são as *tabelas virtuais* (views), vistas na subseção 1.1.5. No tratamento de views, a referência à tabela virtual (após a palavra-chave FROM) e a condição dada através de WHERE são modificadas de acordo com a regra estabelecida. O sistema de reescrita captura a parte de ação da regra, onde não há qualquer menção a tabela virtual e então, o nome da tabela e as condições dadas são verificados e a seguir o nome é reescrito em lugar do nome da tabela virtual e as condições são acrescentadas à cláusula WHERE da consulta feita. Esta fase é intermediária e em algumas versões anteriores do PostgreSQL eram feitas durante a execução da consulta. Os arquivos que realizam a reescrita da árvore de usuário estão em path/backend/rewrite.

5. **Planejamento e Otimização**, cujos arquivos fonte estão localizados no diretório path/backend/optimizer. Esta fase tem como objetivo encontrar um plano para a última fase, a de execução, rodar a query. Existem vários diretórios para o otimizador: o /plan gera o plano de saída atual, o /path en-

contra todos os possíveis caminhos para fazer a junção das tabelas, o /prep cuida de alguns casos especiais como herança e o /geqo faz um plano separado através do mecanismo de "otimização genética"(um mecanismo semi aleatório, no qual se realiza junção de todos os espaços da árvore, buscando todas as possibilidades de junção exaustivamente, dispondo-as em árvore e retornando-as para /path criar caminhos). A tarefa de otimização consiste em escolher entre todos os planos encontrados, aquele de custo mais baixo.

O otimizador gera um plano de otimização ótimo, fazendo uma mais ou menos exaustiva busca através de todas as maneiras possíveis de execução. Cada possibilidade de execução é colocada em uma árvore, e estas árvores representam as diversas maneiras de executar uma query. A melhor árvore é escolhida através de um processo recursivo:

- é feita uma estrutura RelOptInfo para cada relação base. Cada possível maneira de ter acesso à estrutura é encontrada e colocada numa lista de camino para RelOptInfo. Também se gera uma lista de todas as junções para uma relação;
- as diversas estruturas RelOptInfo são analizadas a partir das informações de junção, gerando um caminho para cada possível junção;
- analiza-se o custo do caminho para poder escolher aquele menos custoso.

6. **Execução**: o executor encontra-se em path/backend/executor. O executor possui diversos arquivos, dentre os quais o principal é execMain.c, realiza a execução através de três procedimentos: ExecutorStart(), ExecutorRun() e ExecutorEnd(). O ExecutorStart() é chamado no início da execução de uma query e retorna um ponteiro para os atributos das tuplas que devem ser retornados pela query. O ExecutorRun() que retêm as tuplas na direção da execução da query e o ExecutorEnd() finaliza a execução.

Capítulo 2

Implementação

Este capítulo destina-se os aspectos de implementação de aspectos temporais no PostgreSQL. Já foi dito no capítulo anterior que primeiro analiza-se léxica e sintaticamente a estrutura de entrada por meio de programas criados através dos aplicativos Unix lex e yacc. Logo, naturalmente nossa análise começará neste ponto.

2.1 Lex e Yacc

Lex e yacc são ferramentas desenhadas para compiladores e interpretadores. Eles agem sobre uma estrutura padronizada na entrada, modificando-a. Numerosas aplicações são feitas com estes programas: desde um simples programa de texto até compiladores como o PCC (Portable C Compiler) e o gcc (GNU C Compiler), incluindo verificadores para checagem da sintaxe SQL.

Duas tarefas normalmente são necessárias para programas que possuem entrada estruturada. Primeiro devemos dividir a entrada em unidades significativas e depois descobrir o relacionamento existente entre eles. Estas unidades, nas quais a entrad é dividida são normalmente chamadas *itens léxicos* (ou *tokens*) e o processo de divisão da cadeia de entrada nestes é chamada *análise léxica*. Lex ajuda a realizar este procedimento através de um conjunto de descritores de itens léxicos (especificações lex) e de código C que formam um *analisador léxico* (ou *lexer*).

Os tokens são descritos como expressões regulares, semelhantes as usadas em comandos como grep. Lex usa as especificações para buscar na entrada padrões que combinem com as expressões regulares de forma extremamente rápida, independentemente do número de expressões que ele tenta combinar. Normalmente, uma analisador escrito com lex é mais rápido que um escrito manualmente.

Um analisador léxico transforma uma cadeia de caracteres numa cadeia de tokens. No entanto, a segunda tarefa (a de descobrir os relacionamentos entre os

itens léxicos) não é feito por lex. Para relacionar os elementos da entrada entre si, necessitamos de uma *gramática*. Yacc toma a descrição concisa da linguagem e produz código C que analisa a entrada, buscando o encadeamento de tokens que constituem uma regra da gramática. O programa que realiza esta função é chamado *analisador sintático* (*parser*). Os analisadores sintáticos escritos manualmente normalmente são mais rápidos que os desenvolvidos por yacc, mas através deste programa desenvolvê-los torna-se mais fácil e também mais rápido.

Lex e yacc normalmente trabalham juntos, como mostra a figura abaixo: O

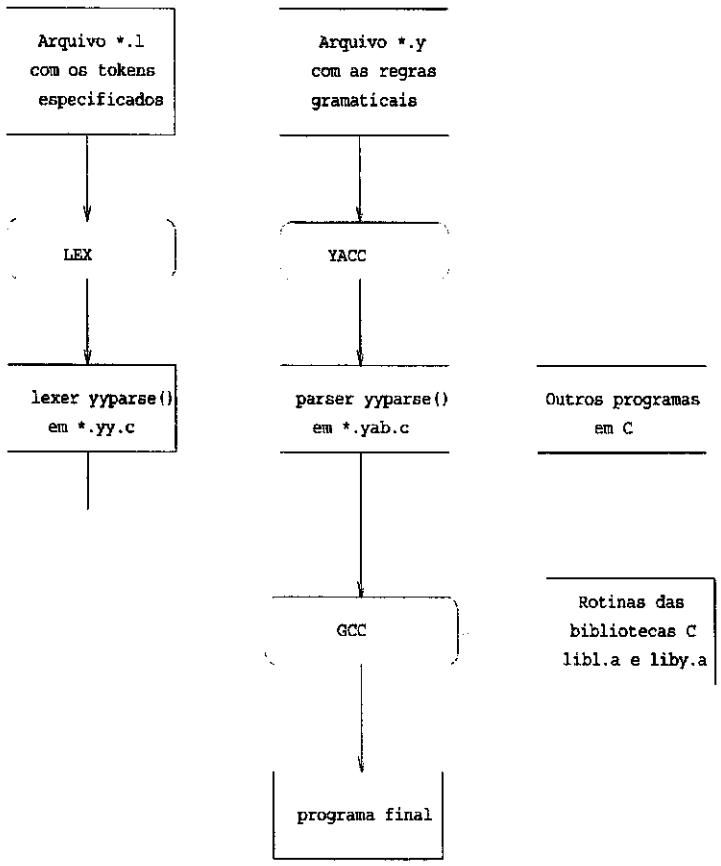

Figura 2.1: Uso de lex e yacc

programa final é o resultado da ligação dos analisadores léxico e sintático produzidos por lex e yacc, respectivamente, com outros programas que permitem fazer tratamento específico, como, por exemplo, de funções de tratamento de tokens ou regras de gramática, e ainda das bibliotecas de sistema libl.a (para o lex) e liby.a (para o yacc). No decorrer do presente capítulo o processo será bem explicitado.

2.1.1 Lex

Lex recebe um arquivo de entrada *.l (ver figura 2.1 na página 21), dividido em três partes, separadas por (%%), segundo o esquema abaixo:

especificações lex
%%
regras
%%
subrotinas

Na parte da especificação das regras lex colocamos algum código inicial que o analisador deve reconhecer (normalmente em C), na parte de regras temos código C que deve ser executado quando o token é encontrado e na parte final do arquivos colocamos subrotinas auxiliares para facilitar a análise léxica.

Nas especificações o código C deve ser inserido entre "%{"e "%}". Normalmente este trecho do analisador possui arquivos de cabeçalho, declarações de variáveis e tipos definidos pelo usuário necessários para se analisar a cadeia de caracteres de entrada. Tudo o que estiver entre "%{"e "%}"é copiado para o arquivo C de saída diretamente pelo programa lex.

As regras possuem uma sintaxe bem definida: *padrão* (*pattern*) e *ação*, separados por espaço em branco. O analisador léxico executará a ação quando encontra o padrão. Os padrões são expressos por expressões regulares (ver tabela 2.1) e

Padrão	Combina com
.	todo caracter único exceto nova linha ("\n")
abc	abc
abc*	ab, abc, abcc, abccc, ...
abc+	abc, abcc, abccc, ...
a(bc)?	a, abc
(01)	01
0|9	0 ou 9

Tabela 2.1: Exemplos de expressões regulares

as ações são normalmente código C a ser executado caso a expressão anterior seja encontrada. Um exemplo de regra simples é

```
[\t ]+ \hspace*{3cm}  /*ignore espaço em branco*/;
```

que por não ter nenhum código que execute uma ação ignora um caracter em branco existente. Um caracter de entrada é combinados com uma expressão regular determinada uma única vez e lex executa a ação determinada para a cadeia mais longa que combina com a entrada.

A parte final do arquivo *.l é a de subrotinas de usuário. Aqui pode-se escrever funções que se dedicam a uma finalidade específica ou modificar rotinas internas do lex. O analisador sintático produzido por lex é uma rotina C chamada yylex(), que normalmente é utilizada nesta parte de código para obter os tokens que são gerados pelo processo de análise.

2.1.2 Yacc

A estrutura de um arquivo de entrada .y para o yacc (ver figura 2.1 na página 21) é semelhante, não por acaso, a estrutura dos arquivos escritos para lex. Veremos que além de serem ligados juntos após o processo de compilação, eles também trabalham em conjunto, cooperando entre si.

Na primeira seção, a de *definições*, temos um bloco literal de código (que pode conter código C delimitado por "%}"e "%}". Ela contém definições de todos os tokens que espera-se receber da análise léxica. A ocorrência do primeiro "%%"marca o início da segunda seção: a de *regras*. Ela possui diversas regras (chamadas *regras de produção, regras* ou simplesmente *produções*). Cada regra consiste de um nome único no lado esquerdo, seguido de ":", uma lista de símbolos, a seguir, código de ação e finalmente ponto-e-vírgula. Estas regras exprimem uma dada gramática.

Seja V_T um conjunto de *símbolos terminais*, V_N um conjunto de *símbolos não terminais*, P um conjunto de *regras* e $S \in V_N$ um *símbolo não terminal inicial* que constitui o ponto de partida de qualquer sentença formada, então uma gramática G é uma quádrupla

$$G = \{V_T,\ \{S\},\ P,\ S\}.$$

A partir do símbolo inicial de uma gramática é possível gerar várias cadeias formadas pelos símbolos terminais, denominadas *sentenças*. Para se aprofundar mais, consulte o capítulo 14 da referência bibliográfica [13].

Com um verificador sintático, queremos o contrário: dado um símbolo, devemos analisar se ele é gerado por uma determinada gramática. Para isso, devemos reduzir a cadeia de símbolos até chegar apenas ao símbolo inicial. Existem várias estrstégias para se atingir este resultado (veja [13], cap.16). Um parser construído com yacc utiliza como técnica de análise o mecanismo de *deslocamento e redução* (*shift-reduce*). Neste caso, o analisador mantém uma tabela de deslocamento e redução (*tabela SR*), que possui as possibilidades de análise (Shift ou Reduce) a partir dos símbolos gramaticais, uma *lista de símbolos* a analisar e uma pilha de símbolos analisados. O analisador funciona da seguinte maneira: realiza-se um deslocamento, colocando um elemento da lista no topo da pilha e a seguir uma redução substiuindo os elementos localizados no topo da pilha por um sím-

bolo não terminal. A tabela SR é verificada para determinar se o passo seguinte é um deslocamento ou uma redução.

Já vimos que para construirmos um analisador completo, devemos escrever um arquivo com terminação .l com os itens léxicos e padrões e um arquivo *.y com a gramática a ser utilizada (ver figura 2.1, página 21). Então, devemos usar lex e yacc para gerar o analisador léxico e sintático e ligá-los juntamente com outros programas gerando um sistema de compilação ou verificação.

Em um sistema como este o lexer e o parser funcionam cooperando entre si, como vemos na figura a seguir. A harmonia entre lex e yacc depende fundamentalmente de dois fatores: a identificação e o valor do token devem ser comum aos dois programas. Para a resolução do primeiro fator basta que especifiquemos os mesmos tokens nas especificações dos arquivos .l e .y. No caso do valor, o lexer yylex() passa os tokens para yyparse() e o valor através do valor de reconhecimento e da variável global yylval, que é do tipo YYSTYPE. O tipo YYSTYPE pode ser definido pelo programador, em caso contrário, o sistema o define como inteiro.

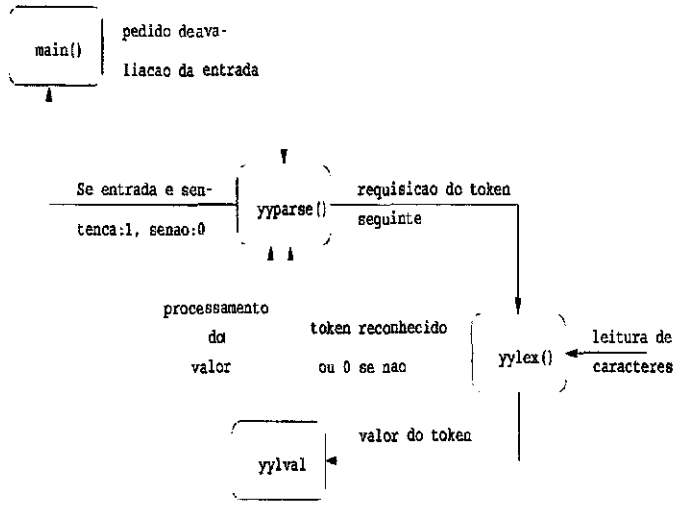

Figura 2.2: Interação entre o lex e o yacc

A parte final do arquivo .y, após o segundo %% como nos arquivos .l, contém as rotinas de usuúario, como por exemplo, um programa main() que faz a chamada a função yyparse(). É importante frisar ainda que o yacc não cria apenas um analisador sintático em código C (arquivo .tab.c), mas também um arquivo de cabeçalho .tab.h contendo as definições dos números dos tokens.

2.2 Implementando o sistema de regras temporais no PostgreSQL

No que se refere a implementação em código podemos agrupar tudo o que foi e deve ser feito em alguns aspectos básicos para facilitar a abordagem dos problemas e soluções sem a perda de uma compreensão adequada do caso. O primeiro aspecto é o sistema de verificação dos padrões de entrada, o segundo é o processamento da consulta realizada e o terceiro aspecto que deve ser abordado é a apresentação do resultado ao usuário que apresentou a consulta ao sistema. No que se refere a implementação dos padrões de entrada, devemos ver como modificar o sistema de tratamento de entradas léxico e gramatical do PostgreSQL para que este possa suportar um padrão de consultas do tipo TSQL. Uma vez reconhecida uma consulta temporal, devemos nos preocupar em como fazer o processamento dessa consulta, ou seja, devemos interpretar a entrada do usuário de modo a realizar as operações desejadas e requeridas e, finalmente, fornecer como saída algo que corresponda as necessidades do usuário.

O objetivo da primeira parte é permitir o uso de queries temporais utilizando os operadores temporais da tabela 1.2 na página 13. Assim podemos fazer consultas temporais da mesma maneira que fazemos consultas não temporais. O processamento feito também deve levar em conta a interpretação da consulta temporal de entrada e dos dados temporais das tabelas armazenadas utilizadas. A saída do sistema também deve levar em conta as características temporais dos dados a fim de que os resultados possam ser retornados ao usuário.

2.2.1 O Sistema de Entrada

O problema básico a resolver é o reconhecimento dos operadores da tabela 1.2. Estes operadores modificam a estrutura do comando de recuperação SELECT no TSQL. Portanto, devemos efetuar uma mudança na regra de reconhecimento do comando SELECT, incluindo na recuperação o suporte a cláusula temporal WHEN.

No subdiretório backend/parser do diretótrio de fontes do PostgreSQL, temos o sistema de verificação léxica e sintática da cadeia de entrada. Para incluir as palavras chaves da linguagem TSQL no PostgreSQL devemos alterar o programa **keywords.c**. Este arquivo possui as palavras chave utilizadas no sistema na estrutura ScanKeywords. Para inserir uma nova palavra-chave no sistema, devemos alterar este arquivo incluindo o termo desejado no formato específico deste arquivo. Alguns dos nomes da tabela de operadores temporais, como WHEN, AFTER, BEFORE, já são palavras reservadas no PostgreSQL. Por esta razão alteramos os nomes dos operadores segundo a tabela 2.2 abaixo

Nome no TSQL	Nome utilizado
WHEN	T_WHEN
BEFORE	T_BEFORE
AFTER	T_AFTER
DURING	T_DURING
EQUIVALENT	T_EQUIVALENT
ADJACENT	T_ADJACENT
OVERLAP	T_OVERLAP
FOLLOWS	T_FOLLOWS
PRECEDES	T_PRECEDES
INTERVAL	T_INTERVAL

Tabela 2.2: Palavras-chaves incluídas no sistema de regras do PostgreSQL

Neste arquivo, o formato das linhas para definição de uma palavra chave é o seguinte:

```
{"nome do token", VALOR},
```

, assim para incluirmos os termos T_BEFORE e T_AFTER, deve-se proceder como abaixo:

```
{"t_before", T_BEFORE},
{"t_after", T_AFTER},
```

Uma recompilação do código fonte[1] (executando *gmake* e *gmake install*) e reinicialização do processo postmaster (ver referências [6], [8], [10] ou o arquivo INSTALL para mais informações) envolve os programas lex e yacc. Neste procedimento é gerado um novo arquivo **parse.h**, com as palavras chaves da segunda coluna da tabela 2.2 inseridas no final do arquivo e acompanhadas por um número que as identifica. O analisador léxico (arquivo **scan.c**) procura as palavras chaves contidas no arquivo **keywords.c** utilizando a função ScanKeywordLookup deste arquivo, que retorna o valor do item léxico encontrado para o analisador sintático (arquivo **gram.c**), como na figura 2.2 da página 24. A tabela 2.3 mostra os nomes dos itens léxicos e os valores retornados a yyparse() pelo mecanismo de análise léxica.

Neste estágio garantimos que o PostgreSQL reconhece, como itens válidos de entrada as palavras chaves da linguagem TSQL. A partir daí precisamos de

[1] ver instruções no arquivo INSTALL do diretório dos fontes do PostgreSQL

mecanismos que garantam que uma consulta temporal possa ser validada pelo sistema sintático de verificação do Postgres. Para isto ser feito precisamos alterar o arquivo **gram.y** que possui as definições e regras gramaticais do sistema gerenciador de bancos de dados utilizado.

Nome	Valor retornado ao parser
t_when	T_WHEN
t_before	T_BEFORE
t_after	T_AFTER
t_during	T_DURING
t_equivalent	T_EQUIVALENT
t_adjacent	T_ADJACENT
t_overlap	T_OVERLAP
t_follows	T_FOLLOWS
t_precedes	T_PRECEDES
t_interval	T_INTERVAL

Tabela 2.3: Itens léxicos retornados pelo processo de análise sintática

Já discutimos neste capítulo a questão da sintaxe de arquivos de terminação .y (formato usado por yacc). Também já abordamos que a cláusula temporal WHEN, T_WHEN no nosso caso, deve ser parte da estrutura do comando SELECT definido como o comando básico de recuperação da linguagem SQL. Seguindo esta direção veremos as alterações realizadas na análise sintática para que o PostgreSQL possa suportar consultas temporais.

1. **Inclusão de t_when no comando SELECT:** na parte do arquivo gram.y que trata do comando SELECT devemos incluir o tratamento a T_WHEN no Subselect. Isto pode ser feito reeditando esta parte do arquivo, tornando-o

```
SELECT opt_distinct target_list
  result from_clause where_clause t_when_clause
  group_clause having_clause
```

com *t_when_clause* declarado nas definições yacc como segue

```
%type <node>    t_when_clause
```

sendo *node* o tipo de variável de t_when_clause. Devemos ainda incluir t_when_clause na pilha de dados do comando SELECT para que o parser (analisador sintático) possa prosseguir para identificar os operadores temporais.

2. **Alteração da lista de tokens em gram.y:** todos os itens léxicos que o lexer (analisador léxico) pode encontrar estão cadastrados com os seus tipos de dados[2] respectivos na parte de definições do arquivo **gram.y**. Abaixo está transcrito a parte onde estão definidas os itens gramaticais necessários ao TSQL.

```
/*Keywords in TSQL
%token          T_WHEN, T_AFTER, T_BEFORE,
T_DURING, T_EQUIVALENT, T_ADJACENT, T_OVERLAP
, T_FOLLOWS, T_PRECEDES
```

3. **Inclusão dos operadores temporais:** devemos incluir os operadores temporais definindo as regras gramaticais da sua identificação. Para definir uma regra gramatical precisamos notar que uma cláusula T_WHEN possui a forma genérica **T_WHEN INTERVALO_DE_TEMPO OPERADOR INTERVALO_DE_TEMPO**. Em termos de notação yacc isto foi implementado separando-se esta estrutura genérica em duas partes: a primeira consistindo do nome da cláusula e o primeiro intervalo de tempo e a segunda do operador e o segundo intervalo temporal. Esta segunda parte foi chamada *t_operations* e os intervalos de tempo pelo nome *time_interval*.

4. **Modificação da estrutura de dados:** muitas das alterações enumeradas nos itens anteriores envolvem modificação na estrutura de dados do PostgreSQL. Isto foi feito diretamente no diretório *nodes* (cuja localização encontra-se na nota de rodapé desta página). Algumas destas alterações é a inclusão de T_WHEN como variável do tipo *Node* na estrutura de definição do comando SELECT localizada no arquivo **parsenodes.c**, a de uma nova lista para suportar *t_operations* e aos intervalos de tempo (*time_interval*) onde devemos tratar a existência de intervalos endereçados pela palavra-chave T_INTERVAL.

As alterações mencionadas acima garantem o reconhecimento de uma entrada escrita em TSQL. Ainda devemos tratar esta entrada, o que deve ser feito processando as operações realizadas na consulta de entrada.

[2]os tipos de dados definidos para o PostgreSQL encontram-se no subdiretório /include/nodes do diretório onde se encontra o código fonte do SGBD

2.2.2 O Processamento das Consultas Temporais

Para fazer o processamento de consultas temporais optamos por transformar as consultas em TSQL em consultas em SQL padrão. Isto é facilmente feito usando as definições dos operadores vistos (tabela 1.2, página 13). A idéia básica é transformar as operações temporais em condições que podem ser tratadas por meio de cláusulas WHERE padrão. Para isso basta acrescentar as condições temporais a cláusula WHERE usando o operador AND e os rótulos temporais $[T_S, T_E]$ envolvidos (para detalhes consulte as referências [1] e [12]).

O princípio portanto é modificar o tratamento da cláusula WHERE para que após a identificação da estrutura de escrita temporal na entrada a consulta seja processada de maneira semelhante a uma consulta SQL padrão. Isto garante o mínimo de alteração possível no sistema além de se constituir numa saída elegante para o problema. No subdiretório /backend/parser do diretório de fontes do PostgreSQL o tratamento da cláusula WHERE é realizado no arquivo **parse_clause.c**. Neste arquivo devemos alterar a estrutura de dados da cláusula WHERE, que é do tipo *Node* (propositadamente o mesmo tipo que usamos para aramazenar a estrutura da cláusula T_WHEN). Isto oferece compatibilidade e facilita o acesso aos dados da consulta utilizadas pela cláusula WHERE. Desta maneira temos uma entrada escrita em TSQL, que é tratada e reconhecida como um padrão temporal, mas processada como uma consulta SQL qualquer. Este procedimento resolve o problema e é transparente ao usuário final, que possuirá suporte real a dados temporais.

Capítulo 3

Conclusão

As características do PostgreSQL justificam sua escolha para este projeto. Ele consegue unir robustez, recursos avançados, escalabilidade, além de um bom modelo de desenvolvimento, evolução e crescimento. Seu código fonte, embora em constante evolução, é bastante estável já que normalmente, as mudanças que ocorrem são melhoramentos de algumas características do sistema ou acréscimo de novas características novas necessárias a um gerenciador de bancos de dados que pretende concorrer com os poderosos sistemas de bases de dados existentes comercialmente.

Atualmente, no desenvolvimento do código fonte final, temos um projeto em pleno andamento. A fase de tratamento de consultas temporais está sem problemas, ou seja, o sistema pode reconhecer uma entrada escrita em TSQL. A fase de processamento, no entanto, ainda possui problemas a serem solucionados. Como se trata da alteração de um código escrito por muitos programadores em diversas partes do planeta, é fácil surgirem problemas quando se altera as estruturas de dados e variáveis que endereçam memória no sistema. A resolução deste problema poderá ser realizada dependendo da continuação deste trabalho. Também é interessante desenvolver um código fonte em um formato de distribuição tal que possa ser utilizado pelos usuários do PostgreSQL ao redor de todo o mundo. Isto deve ser feito, da mesma forma, na continuação deste projeto. Finalmente, pretendemos implementar este sistema em aplicações científicas práticas. Neste aspecto vale ressaltar que um banco de dados temporal seria extremamente importante para esta instituição, por poder tratar dados científicos que dependem do tempo de forma eficiente.

Referências Bibliográficas

[1] Tansel, A. U.[et al.] *Temporal Databases: theory, design and implementation*. The Benjamin/Cummings Publishing, Redwood, CA, 1993.

[2] Elmasri, R., Navathe, S. *Fundamentals of Database Systems*. Addison-Wesley, Menlo Park, CA, 1994.

[3] Abiteboul, S., Hull, R., Vianu, V. *Foundations of Databases*. Addison-Wesley, Reading, MA, 1996.

[4] Date, C.J. *Bancos de Dados*; tradução Newton Dias de Vasconcellos; revisor técnico Nelson Luiz Correa Rabelo. Ed. Campus, Rio de Janeiro, 1988.

[5] Stephens, R.K, Plew, R.R., Morgan, B, Perkins, J. *Teach Yourself SQL in 21 days*, 2nd Edition. Mcmillan Computer Publishing.

[6] The PostgreSQL Global Team. *PostgreSQL Tutorial*. Por Thomas Lockhart, disponível em `http://pgsql.dbexperts.com.br/users-lounge/docs/7.0/tutorial`.

[7] Edelweiss, N. *Tópicos Avançados em Modelos de Bancos de Dados*. Disponível em `http://shark.inf.ufrgs.br/cmp15120001/BDsTemporais/index.htm`

[8] The PostgreSQL Global Development Group. *The PostgreSQL Administrator's Guide*. Por Thomas Lockhart, disponível em `http://pgsql.dbexperts.com.br/users-lounge/docs/7.0/admin`, 2000.

[9] The PostgreSQL Global Development Group. *The PostgreSQL Programmer's Guide*. Por Thomas Lockhart, disponível em `http://pgsql.dbexperts.com.br/users-lounge/docs/7.0/programmer`, 2000.

[10] The PostgreSQL Global Development Group. *The PostgreSQL User's Guide*. Por Thomas Lockhart, disponível em `http://pgsql.dbexperts.com.br/users-lounge/docs/7.0/user`, 2000.

[11] Levine, J.R, Mason, T., Brown, D.. *Lex & Yacc*. O'Reilly & Associates,Inc, 1992.

[12] Alves, R.P.J. *SAT-5 - Um sistema com abordagem temporal de aquisição de telemetria, ativação de telecomandos e armazenamento textual sobre a atmosfera terrestre para os experimentos de carga útil a bordo do Satélite de Aplicação Científica - SACI-1*. Dissertação de Mestrado. Campina Grande, UFPB, 1997.

[13] Ricarte, I.L. *Compiladores*. Curso de Mini e Microcomputadores: Software, disponível em `http://www.dca.fee.unicamp.br/~eleri/ea877/ea877.html`, 1999.