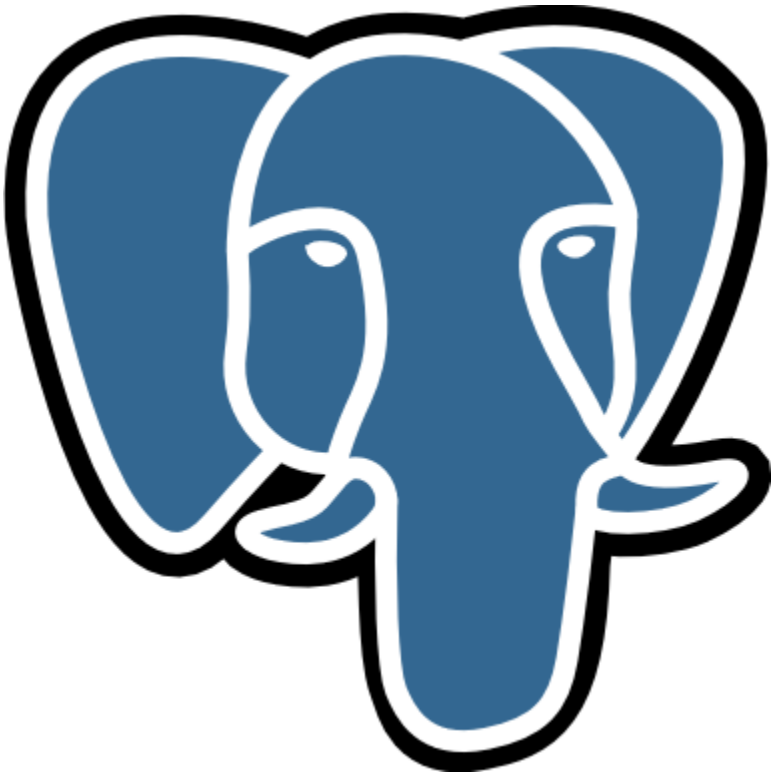
Grupo 26
nº 42356 – David Ferreira
nº 41904 – Joel Reis
nº 42012 - Nuno Ribeiro

Sistemas de Bases de Dados
Trabalho Final
"PostgreSQL"



Docente: José Júlio Alferes

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Junho de 2014

Índice

1 Introdução

Este trabalho foi desenvolvido no âmbito da cadeira de Sistemas de Bases de Dados onde nos foi pedido para fazer uma análise a um sistema de base de dados à nossa escolha, no nosso caso PostgreSQL, onde foi abordado as vertentes estudadas ao longo da disciplina, como por exemplo o tipo de ferramentas, serviços e que *features* adicionais o sistema fornece. Com o final do estudo pretende-se conseguir comparar entre o sistema escolhido e o abordado na disciplina, Oracle 11g.

Ainda neste capítulo será apresentado uma breve introdução histórica do sistema e os casos onde esta tecnologia é usada. No capítulo 2 iremos retratar como funciona o armazenamento e a estrutura dos ficheiros que constituem o PostgreSQL. No capítulo 3 iremos abordar a indexação e hashing de forma a entender que tipos de estruturas para armazenar índices nas tabelas é utilizado. No 4 descrevemos o processamento e otimização de perguntas, entendendo que tipo de linguagem intermedia utiliza e que operações e expressões são compatíveis com o sistema. No 5 explicamos como é gerido as transições e como é feito o controlo de concorrência. No 6 abordamos o suporte para bases de dados distribuídos e no 7 e final introduzimos o leitor a algumas outras características do sistema estudados que consideremos interessantes e vantajosas. Optamos por no final de cada capítulo fazer uma pequena comparação com o Oracle.

Introdução histórica

O sistema de gestão de bases de dados hoje conhecido por PostgreSQL deriva do POSTGRES, originário da Universidade da Califórnia em Berkeley. Tendo o seu desenvolvimento mais de uma década, o PostgreSQL é a base de dados de código aberto disponível mais avançado existente.

A implementação do POSTGRES começou em 1986 e passou por várias versões principais desde essa data. Tem sido usado para implementar diversos aplicativos diferentes de pesquisa e produção, como sistema de análise de dados financeiros, base de dados de informações na área da medicina, etc. Tem também sido usado como ferramenta educacional por várias universidades.

Em 1994, Andrew Yu e JollyChen desenvolveram um interpretador de linguagem SQL ao POSTGRES, passando a Postgres95. Em 1996 o nome "Postgres95" passou a PostgreSQL, para refletir a ligação entre o POSTGRES original e as versões mais recentes com capacidade SQL.

2 Armazenamento e File Structure

Buffer Management

O *PostgreSQL* tem controlo do *buffer management*, este utiliza *buffers* de cache partilhada que podem ter diferentes tamanhos manipulando a variável *shared_buffers* no ficheiro *postgresql*.conf. Este buffer é basicamente um *array* com o tamanho especificado em que cada *cache* aponta para um bloco de 8KB (denominado página) de dados. Trata-se ainda de um *buffer* circular em que quando este chegar ao tamanho final começa novamente da posição 0.

Este sistema segue uma lógica de *least-recently-used* (LRU), tentando assim sempre manter apenas as páginas mais visitas recentemente e descartando as restantes através de uma rotina chamada *VACUUM* que ajuda a limpar quando necessário as páginas armazenadas.Porém um LRU simples não tem memória, logo este processo não consegue distinguir as páginas que foram acedidas imensas vezes das que foram acedidas uma só vez no passado.

O Oracle tem as suas próprias políticas de gestão de Buffer Management, sendo estas ainda de grande complexidade.

Sistema de ficheiros

O *PostgreSQL* baseia-se no *file system*do sistema operativo para guardar a informação relativa á base de dados que usualmente se encontra armazenada na diretória PGDATA sendo a sua localização mais comum */var/lib/pqsql/data*, enquanto que o Oracle implementa o seu próprio sistema de ficheiros. Múltiplos *clusters* são permitidos, sendo estes geridos por diferentes instâncias do servidor existentes na mesma máquina.

A diretoria PGDATA contem várias subdiretorias e ficheiros de controlo como demonstrado na seguinte tabela.

Item	Descrição
PG_VERSION	Um ficheiro que contém o número da versão do PostgreSQL
base	Subdiretoria que contém todas as subdiretorias da base de dados.
global	Subdiretoria que contém as tabelas de cluster inclusive pg_database
pg_clog	Subdiretoria que contém o estado de cada transação
pg_multixact	Subdiretoria que contém o estado de cada multi-transação
pg_notify	Subdiretoria que contém o estado do sistema de notificação (LISTEN/NOTIFY)
pg_serial	Subdiretoria que contém informação sobre transações serializáveis que foram realizadas
pg_snapshots	Subdiretoria que contém *snapshots* exportados
pg_stat_tmp	Subdiretoria que contém ficheiro temporários para o sub-sistema de estatísticas
pg_subtrans	Subdiretoria que contém o estado das sub-transações
pg_tblspc	Subdiretoria que contém as ligações simbólicas para as tablspaces
pg_twophase	Subdiretoria que contém ficheiros WAL (WriteAhead Log)

postmaster.opts	Ficheiro que grava as opções da linha de comando com que o servidor foi inicializado pela última vez
postmaster.pid	Ficheiro que grava o ID do processo *postmaster*actual (PID), a diretoria do *cluster data*, o *timestamp* inicial do *postmaster*, número da porta, diretória da *socket* do Unix-domain (vazia no Windows), o primeiro *listen_address*válido (IP ou vazio se não for TCP) e o ID do segmento de memória partilhada (este ficheiro não é apresentado depois do servidor desligar)

Para cada base de dados no *cluster* existe uma subdiretoria no ficheiro PGDATA/base cujo nome será obtido através do seu OID (ObjectIdentifier) no *pg_database*. Esta subdiretoria é a localização *default* para os ficheiros da base de dados, onde os catálogos do sistema são armazenados.

Cada tabela e índice são armazenados num ficheiro à parte "*pg_class.relfilenode"*, sendo que o nome atribuído vai de acordo com o número *filenode*da respetiva tabela ou índice. Para as relações temporárias, o nome do ficheiro tem o formato tBBB_FFF, onde o BBB é o *backend*ID do criador do ficheiro e o FFF é o número *filenode*.

As tabelas e índices contém também o *free spacemap*, que armazena a informação relativa ao espaço livre disponível num ficheiro como o nome do número *filenode* mais o sufixo *_fsm*. As tabelas também contém um mapa de visibilidade que é armazenado num ficheiro com o respetivo nome mais o sufixo *_vm*, que permite saber quais as páginas que não contêm tuplos mortos.

No caso de uma tabela ou índice exceder 1 GB (este valor é o predefinido que pode ser ajustado usando a opção de configuração --with-segsize na compilação do PostgreSQL), este é então dividido em vários segmentos que terá o nome do ficheiro com o número do segmento como sufixo.

Cada *tablespaces* tem um link simbólico na diretória *PGDATA/pg_tblspc*, que aponta para a diretoria física da *tablespace*. O nome deste é gerado usando o OID da *tablespace*. Os *tablespacespg_default* e *pg_global* não podem ser acedidas a partir do *pg_tblspc*, mas podem respetivamente serem acedidos através do *PGDATA/base* e *PDDATA/global*.

A função *pg_relation_filepath()* exibe a diretoria inteira (relativa ao *PGDATA*) de qualquer relação.

Partições

O *PostgreSQL* tem um mecanismo de partições de tabelas, isto é, permite a divisão da informação de uma grande tabela em várias tabelas mais pequenas. A partição permite assim melhorar o tempo de acesso a tabelas grandes frequentemente acedidas.

Atualmente o *PostgreSQL* suporta partição de tabelas por hierarquia. Cada partição deve ser criada como filha de uma tabela pai. A tabela pai por norma encontra-se vazia e serve apenas para representar os logicamente os dados.

Há dois tipos de partições que podem ser implementadas em *PostgreSQL*: *Range Partitioning*, a tabela é particionada em "*ranges*" definida por uma coluna chave ou por um conjunto de colunas, sem sobreposição entre os *ranges* atribuídos a diferentes partições; *ListPartitioning*, a tabela é particionada especificando os valores correspondentes a cada partição.

De seguida será apresentado o processo de criação de uma bela particionada.

1. O processo começa pela criação da tabela *master* que será particionada:

```
CREATE TABLE measurement(
city_id int not null,
logdate date not null,
peaktemp int,
unitsales int
);
```

2. De seguida cria-se as ditas partições, que são as tabelas filhas da tabela *master*, neste caso cria-se uma partição para cada mês ativo:

```
CREATE TABLE measurement_y2006m02 () INHERITS(measurement);
CREATE TABLE measurement_y2006m03 () INHERITS(measurement);
…
CREATE TABLE measurement_y2007m11 () INHERITS(measurement);
CREATE TABLE measurement_y2007m12 () INHERITS(measurement);
CREATE TABLE measurement_y2008m01 () INHERITS(measurement);
```

3. Pretendemos depois criar restrições de forma a garantir que os tuplos adicionados à tabela *master* sejam também adicionados às respetivas tabelas filhas:

```
CREATE TABLE measurement_y2006m02 (
CHECK(logdate >= DATE'2006-02-01' AND logdate<DATE'2006-03-01'))
INHERITS(measurement);

CREATE TABLE measurement_y2006m03 (
CHECK(logdate >= DATE'2006-03-01' AND logdate<DATE'2006-04-01'))
INHERITS(measurement);
…
CREATE TABLE measurement_y2007m11 (
CHECK(logdate >= DATE'2007-11-01' AND logdate<DATE'2007-12-01'))
INHERITS(measurement);

CREATE TABLE measurement_y2007m12 (
CHECK(logdate >= DATE'2007-12-01' AND logdate<DATE'2008-01-01'))
INHERITS(measurement);

CREATE TABLE measurement_y2008m01 (
CHECK(logdate >= DATE'2008-01-01' AND logdate<DATE'2008-02-01'))
INHERITS(measurement);
```

4. É importante indexar as colunas chaves:

```
CREATE INDEX measurement_y2006m02_logdate ON
     measurement_y2006m02(logdate);
CREATE INDEX measurement_y2006m03_logdate ON
     measurement_y2006m03(logdate);
…
CREATE INDEX measurement_y2007m11_logdate ON
     measurement_y2007m11(logdate);
CREATE INDEX measurement_y2007m12_logdate ON
     measurement_y2007m12(logdate);
CREATE INDEX measurement_y2008m01_logdate ON
     measurement_y2008m01(logdate);
```

5. Uma vez que pretendemos que a nossa aplicação redirecione a informação para a tabela de partição correspondente quando lhe é feito um pedido "INSERT INTO measurement …" criamos uma função *trigger*:

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
  IF(NEW.logdate>=DATE'2006-02-01' AND
   NEW.logdate<DATE'2006-03-01') THEN
     INSERT INTO measurement_y2006m02 VALUES(NEW.*);
  ELSIF(NEW.logdate>=DATE'2006-03-01' AND
   NEW.logdate<DATE'2006-04-01') THEN
     INSERT INTO measurement_y2006m03 VALUES(NEW.*);
  …
  ELSIF(NEW.logdate>=DATE'2008-01-01' AND
   NEW.logdate<DATE'2006-02-01') THEN
  INSERT INTO measurement_y2008m01 VALUES(NEW.*);
  ELSE
   RAISE EXCEPTION 'Date out of range. Fix the
   measurement_insert_trigger() function!';
  END IF;
  RETURN NULL;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER insert_measurement_trigger
  BEFORE INSERT ON measurement
  FOR EACH ROW EXECUTE PROCEDURE
  measurement_insert_trigger();
```

Neste sistema é ainda possível remover ou alterar as partições geradas o que permite gerir pedaços de informação, sendo ainda possível criar novas tabelas sobre as removidas.

Respetivamente, remoção:

```
DROP TABLE measurement_y2006m02;
```

E alteração:

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

O Oracle permite a partição de tabelas através de Range partitioning, Hashpartitioning e Compositepartitioning.

Sistema de armazenamento

O Postgresql tem um sistema de armazenamento para as tabelas e índices num *array* de páginas (heap) de um tamanho fixo (usualmente 8kB). Uma vez que numa tabela, todas as páginas são logicamente equivalentes, estes podem armazenar um tuplo em qualquer página.

No geral uma página será constituída por 5 partes:

Item	**Descrição**
PageHeaderData	24 bytes que contém a informação geral da página, incluindo apontadores para os espaços vazios.
ItemIdData	Array de (offset, length) pares a apontar para os itens originais. 4 bytes por item
Free space	Espaço por alocar. Apontadores para os novos itens começam a ser alocados neste espaço e o espaço para os respetivos itens é alocado do fim.
Items	Os itens em questão
Specialspace	Método de índice de acesso a dados específicos. Diferentes métodos armazenam diferentes dados. Vazio em tabelas ordinárias.

Registos de tamanho variável e Clustering

O *PostgreSQL*usa páginas de tamanho fixo (usualmente 8kB) e não permite que os tuplos criem múltiplas páginas, no caso do Oracle este permite agrupar as tabelas através de múltiplas páginas. Logo não é possível armazenar grandes valores diretamente. Para poder ultrapassar esta dificuldade, os grandes valores são comprimidos e/ou partir em múltiplas filas físicas. Essa técnica é conhecida por TOAST (The Oversized-Attribute Storage Technique or "the best thing since sliced bread").

Apenas determinados tipos de dados suportam TOAST, os tipos de dados devem ter uma variável tamanho (*varlena*). O TOAST tem por base a associação de uma tabela TOAST com colunas de registos de tamanhos variáveis, ou TOAST-able. Assim os valores *out-of-line* são divididos em pedaços de tamanho TOAST_MAX_CHUNK_SIZE bytes (2000 bytes) e armazenados em diferentes filas na tabela TOAST. Cada tabela TOAST tem colunas *chunk_id*,*chunk_seq* e *chunk_data*. A vantagem de ter um único índice no *chuck_id* e *chunk_seq* é a possibilidade de obter os valores de forma mais rápida.

Um mecanismo que permite o reordenamento nas tabelas com base no índice é denominado de *clustering* e podes otimizar imenso as bases de dados. Porém o PostgreSQL não permite *multitable*clustering, apenas *clustering* de uma tabela, pelo que o seguinte comando é o usado:

```
CLUSTER table_name [USING index_name];
```

3 Indexação e Hashing

Os índices são estruturas auxiliares que permitem obter os dados pretendidos mais rapidamente. A sua utilização é motivada pela necessidade de obter resultados mais rapidamente quando se trata de Bases de Dados muito grandes. Os indíces armazenam dados de uma tabela, organizados pelo(s) atributo(s) especificados pelo utilizador.

Apesar de poderem trazer muitas vantagens, os índices podem também piorar a performance, quando usados incorrectamente. Tanto a criação como as sucessivas actualizações do indíce podem deixar o sistema num estado de o*verhead*. Assim, a criação de um índice só é justificada se o ganho em tempo no auxílio das consultas compensar as próprias actualizações do índice.

O comando usado pelo PostgreSQL para a indexação de valores é o **CREATE INDEX** e é formatado da seguinte maneira:

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON
table [ USING method ]
( { column | ( expression ) } [ opclass ] [ ASC | DESC ]
[ NULLS { FIRST | LAST } ] [, ...] )
[ WITH ( storage_parameter = value [, ... ] ) ]
[ TABLESPACE tablespace ]
```

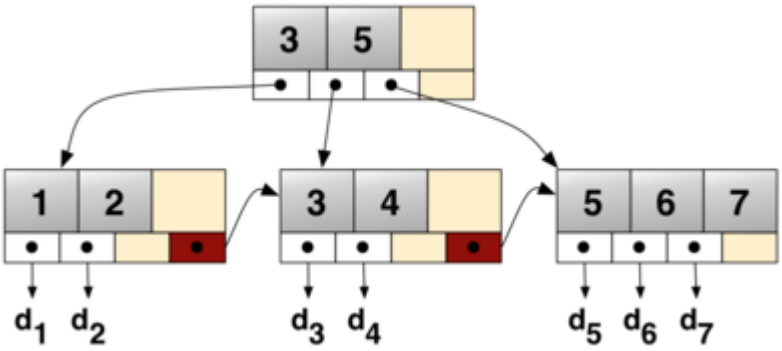
O PostgreSql utliza os seguintes índices:

- **Árvores B+** (apesar de o manual do PostgreSQL mencionar *B-tree index*, o qual difere do *B+tree index*, o utilizado pelo PostgreSQL é o *B+tree*)
- **Hash**
- **Árvores Genéricas de Pesquisa *GiST*** (*Generalized Search Tree*)
- **Índices Genéricos Invertidos *GIN*** (*Generalized Inverted Index*)
- **Bitmaps**

Quando o tipo de índice não é especificado quando o mesmo é criado, o PostgreSQL opta, por defeito, pela criação de uma Árvore B+.

➢ ÁRVORES B+

Este tipo de índices armazenam os dados de uma forma ordenada, forma essa que permite um acesso eficiente aos tuplos pretendidos. Por estar ordenada, permite também apresentar os dados ordenadamente, se essa for a escolha. Nesta estrutura, os dados estão armazenados no nível mais baixo da árvore, nas *folhas*. Neste último nível todos os valores estão ligado entre si, numa Lista Ligada, como apresentado na figura abaixo:



Exemplo de um árvore B+

Estes índices podem ser utilizados em consultas de igualdade ou intervalos, ou seja, consultas que utilizem os operadors =, <, >, <=, >= ou mesmo numa combinação destes operadores, tais como **BETWEEN** e **IN**.

A implementação deste índice em PostgreSQL utiliza o algoritmo *Lehman-Yao*

O comando para a criação de um índice B+ é:

CREATE INDEX *nome* **ON** *tabela* (atributo);

- **HASH**

  Os índices hash apenas suportam consultas que utilizam o operador de igualdade. O hash utilizado é dinâmico e não permite a utilização do rehash tradicional, o que iria permitir a remoção dos buckets em overflow. O rehash consiste na divisão dos buckets quando algum atinge a sua capacidade máxima. A divisão é feita pela ordem em que os buckets se encontram.

  Para a criação de um índice hash recorre-se à cláusula *USING*:

  CREATE INDEX *nome* **ON** *tabela* **USING hash** (atributo);

- **GiST**

  Os índices GiST não são um índice por si só, mas sim uma estrutura de dados onde podem ser implementadas várias Árvores de Índice, possibilitando a indexação de vários tipos de dados.

- **GIN**

  Os índices GIN são, tal como os GiST, estrutra de dados que implementam vários tipos de árvores de índice. Os índices GIN são índices invertidos pois permitem trabalhar com valores que têm mais de uma chave, por exemplo os arrays.

- **BITMAP**

Contrariamente com o que acontece no Oracle, não é permitido ao utilizador criar um índice de Bitmap. No entanto, este tipo de índices é utilizado pelo próprio PostgreSQL internamente, quando a consulta à base da dados envolve múltiplos índices.

No caso da consulta existir a cláusula

WHERE n=10 AND m=15

ou

WHERE n=10 OR m=15

sendo **n** e **m** nomes de atributos de tabelas, o sistema pesquisa cada uma das condições separadamente e prepara um bitmap em memória que indica quais os tuplos que respeitam a condição pesquisada. De seguida, faz o **AND** ou o **OR**, consoante o caso, aos dois bitmaps e fica com o bitmap a indicar os tuplos que satisfazem a condição total da consulta.

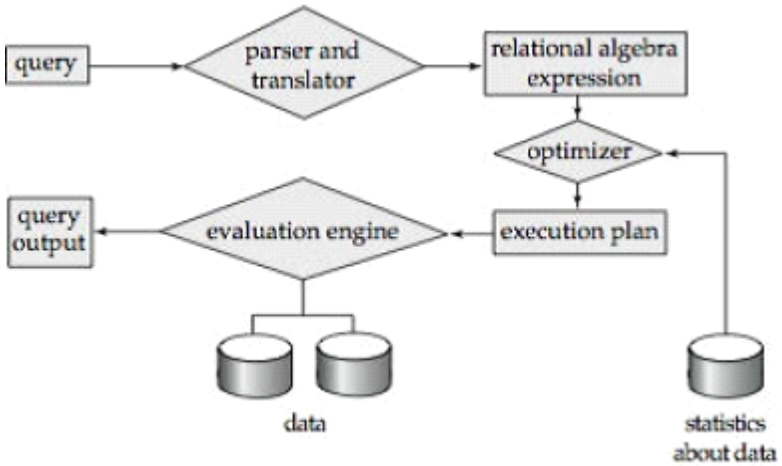
4 Processamento e Optimização de Perguntas

No PostgreSQL, o processamento de perguntas está relacionado com o conjunto de actividades envolvidas na extracção de dados da base de dados. A optimização de perguntas é o processo de selecção do plano de avaliação de perguntas mais eficiente de entre as varias estratégias possíveis para o processamento de uma pergunta.

Sendo o PostgreSQL uma linguagem declarativa de alto nível, o sistema constrói um plano de avaliação de perguntas de forma a minimizar o custo de avaliação de perguntas. Adicionalmente, o PostgreSQL usa a linguagem C como linguagem intermédia.

O percurso de uma query pode ser descrito em cinco passos principais:

- Estabelece-se uma ligação da aplicação para um servidor do PostgreSQL. A aplicação transmite a pergunta e espera por uma resposta por parte deste.
- O parser verifica que a pergunta está correcta (a nível de sintaxe) e cria uma query tree.
- O rewrite system pega nesta árvore criada e verifica se existe alguma regra a aplicar nesta árvore.
- O planificador/optimizador do sistema pega na árvore reescrita e cria um plano de execução que servirá de input para o executador..
- O executor pega nestes dados e executa-os retornando os tuplos na forma representada na árvore. O executor usa o storage system enquanto percorre relações, executando sorts e joins, avaliando qualifications e retornando os resultados pretendidos.



Caminho de uma Query

- **Ligação**
  As ligações ao servidor no PostgreSQL são realizadas utilizando um processo por utilizador, baseado no modelo cliente/servidor, sendo que um processo de utilizador está ligado apenas a um processo de servidor. A atribuição desta relação é realizada por um *master process*, de nome *postgres*, que recebe pedidos por uma porta *TCP/IP* específica e gera um processo de servidor cada vez que recebe um desses pedidos de ligação.

- **Parsing**

  O *parser* numa primeira fase, recebe a pergunta, uma String (em texto ASCII), e verifica a sua validade a nível de sintaxe. Se esta for correcta, gera-se uma *parse tree*. Caso contrário, é retornada uma mensagem de erro. O *lexer* esta definido no ficheiro *scan.l* e é responsável por reconhecer identificadores, SQL *key words*. etc... Para cada *key word* ou identificador encontrados, um *token* é gerado e passado para o *parser*. Este está definido no ficheiro *grammar.y* e consiste num conjunto de regras de gramática e acções que são executadas quando uma regra é accionada. O código das acções é utilizado para construir a árvore. O ficheiro *scan.l* é transformado para o ficheiro fonte *scan.c* utilizando o programa *lex* e o *gram.y* é transformado em *gram.c* Após estas transformações, um compilador normal de C pode ser utilizado para criar o *parser*.

- **Re-write System**
  No processo anterior cria-se uma parse tree usando regras sobre estruturas sintáticas do SQL. Após se utilizar o parser, utiliza-se a árvore gerada e faz-se uma 32 interpretação semântica necessária para saber que tabelas, funções e operadores estão referenciados na pesquisa. A estrutura de dados criada neste passo é a chamada query tree. Uma árvore de pergunta consiste numa representação interna de uma pergunta SQL onde todos os elementos são guardados separadamente. A estrutura destas árvores é a seguinte:

  Tipo de comando - Valor que indica que tipo de comando produziu a árvore de pergunta (SELECT, INSERT, UPDATE, DELETE). Tabela de abrangência - Lista de relações usadas na pergunta. Na instrução SELECT estas são dadas depois da palavra chave FROM.

  Relação resultante - Índice na tabela de abrangência que identifica a relação onde os resultados da consulta vão ser reflectidos. As consultas SELECT não têm uma relação resultante. Para os comandos INSERT, UPDATE e DELETE, a relação resultante é a tabela (ou vista) onde se terão feito as mudanças.

  Lista de alvos - Lista de expressões que definem o resultado da consulta. No caso de um SELECT, correspondem às expressões entre as palavras chave SELECT e FROM. Comandos do tipo DELETE não produzem nenhum resultado. Para comandos INSERT, a lista de alvos descreve as novas linhas que deverão entrar na relação do resultado. Consiste nas expressões na cláusula VALUES ou na cláusula SELECT se o comando for do tipo INSERT. Para os comandos do tipo UPDATE, descreve as novas linhas que deverão substituir as antigas. Todas as entradas da lista de alvos consistem numa expressão. Esta expressão pode ser um valor constante, um apontador para uma coluna de uma das relações da tabela de abrangência, um parâmetro, ou uma árvore de expressões constituída por chamadas de funções, constantes, variáveis, operadores, etc.

  Qualificação - A qualificação de uma consulta é uma expressão cujo resultado é um booleano que indica se a operação (tipo de comando) deve ou não ser executada para o resultado final, corresponde à cláusula WHERE de uma expressão SQL.

  Árvore de junções - A árvore de junção da consulta mostra a estrutura da cláusula FROM. A árvore de junções é apenas uma lista de itens, pois é permitido efectuar a junção em qualquer ordem. No caso de utilizamos OUTER JOINs já temos de efectuar a junção seguindo uma determinada ordem e a

árvore representa a estrutura das expressões de junção. As restrições associadas a determinadas junções (através das expressões ON ou USING) são guardadas como expressões de qualificação associadas a esses nós da árvore de junções. Por vezes também é útil associar a estes nós as expressões da cláusula WHERE como expressões de qualificação. É ainda possível visualizar-se estas árvores no ficheiro de log do servidor alterando os parâmetros de configuração *debug_print_parse*, *debug_print_rewritten*, ou *debug_print_plan*. Há dois processos de parsing que são necessários. O primeiro só faz sentido em caso de transacções, em que simplesmente identifica palavras como BEGIN, ROLLBACK… Estas podem ser imediatamente executadas sem qualquer análise adicional. Após se saber que se está a lidar com uma interrogação podemos iniciar a transacção propriamente dita e aqui é invocado o processo de transformação.

- **Planificador/Optimizador**
  A tarefa do planificador/optimizador é criar um plano de execução óptimo. Uma interrogação SQL pode ser executada de várias formas diferentes, sendo que cada uma tem custos associados, e havendo umas melhores que outras, produzindo-se os mesmos resultados em qualquer um dos casos. Se for possível a nível computacional, o planificador/optimizador examina todas as possibilidades, elegendo a mais vantajosa. Podem haver situações em que verificar todas as possibilidades é muito dispendioso tanto a nível de tempo como de recursos e, por isso, quando o número de joins de uma interrogação ultrapassa um determinado limite, o PostgreSQL utiliza uma ferramenta chamada Genetic Query Optimizer por forma a conseguir criar um plano em tempo útil.
  Os planos criados estão directamente relacionados com os índices disponíveis. Se uma interrogação tiver uma restrição de atributo, e este atributo coincidir com o atributo chave de um índice, é criado um plano de execução para esse índice. Se houve mais índices e/ou mais restrições de atributos, são criados vários planos de execução para cada índice. Criam-se também planos para índices quando as interrogações têm também cláusulas de ordenação (ORDER BY). Um exemplo disto é o caso dos índices B-Tree. Se houver uma pesquisa de ordenação, basta chegar às folhas da árvore e percorre-las. Neste caso, a utilização do índice é útil. Independentemente dos índices utilizados, é sempre criado um índice para pesquisa sequencial, o Seq Scan. Este é sempre criado em qualquer situação. Se a interrogação requer junção de duas ou mais tabelas, planos para a junção de tabelas também são utilizados. Há três planos de junção possíveis (hash join, merge join, nested loop join), relativa à implementação das operações básicas. Quando uma interrogação envolve mais do que duas tabelas, o resultado tem de ser construído a partir de uma árvore de junção de vários passos. O planificador examina as diferentes possibilidades de junção e escolhe a mais barata. Se o número de tabelas intervenientes numa junção for pequeno, então é possível assumir-se que todas as combinações de junção são analisadas. Caso a interrogação implique a utilização de mais tabelas que o limite considerado, as sequências de junção são determinadas por heurística, utilizando o Genetic Query Optimizer. O optimizador realiza uma pesquisa quase exaustiva de todas a combinações possíveis, sendo que quanto maior fosse a pesquisa, mais tempo e mais recursos seriam gastos para fazer esta pesquisa exaustiva. Desta forma, decidiu-se que o ideal para estes casos grandes, era utilizar algoritmos genéticos para ajudar a resolver a situação, sendo que, nem sempre estes retornam a melhor solução possível, mas retornam-na num espaço de tempo muito mais reduzido.

- **Executor**
  O executor pega no plano dado pelo planeador e recursivamente processa-o para extrair os tuplos pretendidos. Aqui tem-se essencialmente um mecanismo de demand-pull pipelining. Sempre que um nó do plano é chamado, este tem de devolver um ou mais tuplos, ou reportar que já terminou.

Algoritmos de Selecção:

Sequential scan - Este é o algoritmo de pesquisa mais básico. É sempre gerado um query plan para este algoritmo. Pode não ser o melhor muitas das vezes, mas é sempre gerado. Tal como o nome indica, o modo de funcionamento do Sequencial Scan passa por percorrer todas as linhas de uma tabela, e, por cada linha, verifica se a condição é respeitada. Se a pesquisa for efectuada numa chave primária, quando o algoritmo encontra o tuplo desejado, este pode parar imediatamente.
Index scan - O index scan baseia-se numa pesquisa sobre um índice. Se for dado um valor inicial de pesquisa, o index scan apenas começará nesse valor, e se for dado um valor final, o index scan parará nesse valor. Relativamente ao sequential scan, apresenta algumas vantagens e desvantagens. A pesquisa sequencial percorre todas as entradas de uma tabela, enquanto uma pesquisa por índice apenas percorre alguns valores (não os percorre todos). Porém, a pesquisa sequencial retorna os valores por ordem de entrada na tabela, e a pesquisa por índice retorna os valores de acordo com a ordem no índice (o que poderá querer dizer que se terão de realizar mais seeks).
Bitmap index scan - O bitmap scan utiliza os diferentes índices criados numa dada tabela para optimizar e aumentar a velocidade de pesquisas mais complexas. Dá uso da indexação. É de notar que poder-se-ão realizar bastantes mais seeks, por isso, este apenas deve ser utilizado com um pequeno conjunto de valores.

Algoritmos de Ordenação:

Sort - Após uma interrogação ter gerado um conjunto de resposta (output table), esta pode opcionalmente ser ordenada. Se tal não for pedido, a tabela será retornada tal como está. Para se ter algo ordenado, é preciso explicitar o comando ORDER BY na interrogação. Há duas formas de ordenação possíveis, ordenação em memória e ordenação em disco. Estas opções são escolhidas de acordo com um parâmetro denominado: work_mem(integer), chamado previamente de sort_mem. Isto determina o espaço de memória gasto para se realizar um algoritmo de ordenação. Se o conjunto de resultados tiver um tamanho inferior a work_mem, ordenação em memória será feita através do algoritmo quicksort. Caso contrário, o conjunto
de entrada será dividido, ordenando cada parte, procedendo-se depois a uma junção dos vários componentes, utilizando o external merge sort.

Algoritmos de Junção:

Nested Loop Join - Este join é feito através da análise de todos os tuplos da relação direita (do join) por cada tuplo da relação esquerda. Esta é uma estratégia de fácil implementação, mas pode ser bastante cara. No entanto, pode-se ter algumas vantagens usando esta técnica quando o uso de índices diminui o número de tuplos a percorrer.

Merge Join - Antes de começar a ser feita a junção entre as duas relações, estas são ordenadas pelos atributos que as une. Depois de ordenadas, as relações serão percorridas paralelamente e são criados os tuplos resultante das junções dos atributos definidos na

query. Este join é interessante porque cada relação apenas tem de ser percorrida uma vez, sendo a ordenação feita através de algoritmos de ordenação ou através de um percurso dado por um índice que ordena os tuplos pelo atributo definido no join.

Hash Join - Inicialmente a relação da direita é carregada para um hashtable em que a chave corresponde aos atributos de junção do "join", depois disto a relação da direita é percorrida e os seus atributos de ligação no "join" são utilizados para encontrar tuplos com o mesmo valor na hashtable anteriormente construída de onde vão saindo os tuplos do resultado.

Outros Algoritmos:

Aggregate - as funções de agregação computam um único valor de um conjunto de valores de input. Estas funções são chamadas quando inseridas instruções do estilo: AVG(), COUNT(), MAX(), MIN(), etc. O operador hashaggregate também é utilizado em expressões com Group By.

Unique - Este operador é utilizado para remover duplicados, que necessita de estar ordenado por coluna, sendo estas únicas. É esta a forma como o operador DISTINCT está definido.

Append - Este operador é utilizado sempre que é detectada uma cláusula de união. É necessária a intervenção de duas tabelas. São ambas ordenadas, e depois ao juntar vão-se eliminando os duplicados.

Materialização - Consiste em analisar as expressões (de baixo para cima) guardando o seu resultado em tabelas temporárias guardadas na base de dados. Os resultados de cada operação intermédia são armazenados e utilizados em avaliações de operações nos níveis superiores.

Pipelining - Consiste na passagem de tuplos para as operações "pai", mesmo enquanto uma operação esteja a ser executada.

Paralelização - Consiste em executar várias tarefas paralelamente.

Relativamente ao sistema de gestão de bases de dados PostgreSQL, podemos ver que nem todas estas propriedades são utilizadas. A materialização é utilizada para algumas operações de sub-selecção. O planificador/optimizador por vezes decide que é menos caro materializar uma sub-selecção do que repetir o trabalho em níveis superiores da árvore. É também
utilizado em algumas operações de merge-join. O pipelining também é utilizado, mais concretamente, demand-pull pipelining. A paralelização é óptima para o algoritmo hash join, mas, o PostgreSQL não a implementa.

Estatísticas:

Como seria de esperar, o planificador tem de estimar o número de linhas devolvidas por uma pergunta de forma a conseguir fazer boas estimativas e aproximações. Assim, surge a necessidade de termos dados estatísticos sobre dados do sistema, de maneira a que esta questão seja resolvida. Um dos componentes das estatísticas é o número de entradas em cada tabela e em cada índice, bem como o número de blocos de disco ocupados por cada tabela (com dimensão BLOCKSIZE, que, em geral, é 8K por tabela) e índices. A informação usada para esta tarefa guarda-se na tabela pg_statistics. Esta guarda informação detalhada sobre os dados das tabelas. Os dados nesta são actualizados também com ANALYSE e VACUUM ANALYZE, e são sempre aproximados, mesmo com updates recentes. Para além dos dados relativos ao conteúdo de tabelas, guardam-se também dados estatísticos sobre os valores de expressões de índices.

5 Gestão de Transacções e Controlo de Concorrência

Transações

Nos sistemas de bases de dados, *PostgreSQL* inclusive, as transações são um conceito fundamental, uma vez que estas permitem agrupar e executar múltiplas operações como uma só, logo basta uma das operações falhar para a transação não ser executada.

Em *PostgreSQL*, a transação é criada rodeando os comandos de SQL da transação

```
BEGIN;
  UPDATE accounts SET balance=balance-100.00
  WHERE name='Alice'
  …
COMMIT;
```

com os comandos BEGIN e COMMIT:

Se por alguma eventualidade for necessário anular a transação executada, pode-se usar o comando ROLLBACK em vez do COMMIT e todos as alterações serão canceladas.

No caso das *nestedtransactions* o *PostgreSQL* não as suporta, porém em alternativa podemos utilizar marcas dentro das transações denominadas de *savepoint* que permitem executar ROLLBACK's até ao *savepoint* específico e desfazer assim as alterações feitas:

```
BEGIN;
  INSERT INTO table1 VALUES(1);
  SAVEPOINT my_savepoint;
  INSERT INTO table1 VALUES(2);
  ROLLBACK TO SAVEPOINT my_savepoint;
  INSERT INTO table1 VALUES(3);
COMMIT;
```

Devido aos ROLLBACK's de cada transação estas por norma não devem demorar muito tempo, mas não há um limite de duração.

Isolamento

O PostgreSQL ao contrário dos outros sistemas que usam *locks*, consegue manter os dados consistentes através de um modelo multiversão. Isto permite que quando uma *query* corre na base de dados cada transição veja uma *snapshot* dos dados já com algum tempo, ignorando os dados atuais. Isto permite proteger a transição de dados de dados inconsistentes devido a atualizações de outras transições sem necessidade de bloquear as várias *queries*.

O sistema de VACUUM já referido que trata da limpeza é necessário uma vez que este método exige armazenar um maior número de dados.

Diferentes níveis de isolamento

Uma vez que o *PostgreSQL* segue o *standard*SQL este utiliza quatro níveis de isolamento, onde o *Serializable* é o mais inflexível sendo definido num parágrafo standard. Os restantes três níveis são definidos em termos de fenómenos, resultante de um interação entre transições concorrentes. Os fenómenos que não são permitidos em vários níveis são:

Dirtyread

A transição lê os dados escritos por uma transição concorrente *uncommitted*.

Nonrepeatableread

A transição relê dados que já tinha lido previamente e descobre que os dados foram *alterados por outra transição.*

Phantomread

A transição reexecuta uma *query* que retorna um conjunto de filas que satisfazem a condição de pesquisa e descobre que o conjunto que satisfaz a condição foi alterada devido a outra transição *committed*.

Internamente existem apenas três níveis distintos de isolação, que correspondem aos níveis *ReadCommitted*, *RepeatableRead* e o *Serializable*. Quando se chama o nível *ReadUncommitted* o realmente utlizado é o *ReadCommitted*.

READ COMMITTED

O nível ReadCommitted é o *default* no *PostgreSQL*, quando a transição usa este nível de isolação, a *query* SELECT vê apenas os dados *committed* antes desta começar, nunca os *uncommitted* ou alterados por transição, sendo igual ainda para os comandos UPDATE, DELETE, SELECT FOR UPDATE e SELECT FOR SHARE.

REPEATABLE READ

Este nível apenas vê os dados *committed* antes da transição, não vê os dados *uncommitted* ou alterados durante a execução da transição pelas transições concorrentes. Assemelha-se imenso ao READ COMMITTED onde este vê os dados antes da *query* dentro da transição e o *repeatableread* vê antes da transição em si. Logo uma sucessiva execução de comandos SELECT numa única transição irá devolver os mesmos dados.

SERIALIZABLE

O nível Seriazable fornece a isolação de transição mais rígida. Este nível emula a execução das transições em serie para todas as transições *committed*, isto é, como se tivessem sido executadas uma a seguir à outra em vez de forma concorrente.

Locks e Deadlocks

O PostgreSQL permite contudo também o uso de locks para controlar o acesso concorrente aos dados bloqueando o acesso a estes quando estão a ser utilizadas, para o mesmo efeito temos tipos diferentes de locks. Por norma os locks só são libertados no final da transição.Estes podem ser divididos em dois grandes níveis de granularidade, tabela ou tuplo.

Um dos problemas do uso de locks é que este irá muito provavelmente originar deadlocks quando duas ou mais transições estão a prender dados pelos quais outras transições aguardam acesso.

Uma das maneiras de evitar os deadlocks é através de timeouts que consiste em contar o tempo de uma transação e caso esse tempo exceda o PostgreSQL consegue detectarodeadlock mantendo um grafo no qual periodicamente vai procurando através de ciclos se existe algum deadlock e se alguma outra transição precisa dos dados retidos. Caso seja encontrado esse deadlock é executado um rollback numa das transações que contribuíram para o deadlock.

Consistência

Aquando de uma transação, existe uma forma, isto é, a melhor, de se garantir a consistência da base de dados. Usando Serializable como nível de isolamento, visto que no caso de haver algum tipo de violação da integridade, uma das transações é abortada de imediato. Já foi demonstrado que os outros tipos de isolamento podem dar origem a leituras inconsistentes. Ainda assim, o utilizador pode configurar uma transação a fim de verificar a integridade da base de dados no fim da mesma transação (DEFERRED) ou até entre operações (IMMEDIATE), a partir do comando:

```
SET CONSTRAINTS { ALL | name [ , …]}{DEFERRED|IMMEDIATE}
```

6 Suporte para Bases de Dados Distribuídas

O PostgreSQL não tem suporte para bases de dados distribuídas. No entanto, existem soluções externas relacionadas com o assunto que podem ser utilizadas, tais como o ***dblink***, que é um módulo que permite a conexão a outras bases de dados em PostgreSQL numa sessão local.

7 Outras características do PostgreSQL

- XML

O PostgreSQL suporta a utilização do XML. É possível efectuar consultas a uma base de dados e retornar os dados em formato XML. Também suporta o tipo de dados XML nas próprias tabelas. A vantagem em armazenar XML num campo de texto é que verifica a correxta formatação dos valores introduzidos e existem funcções para realizar operações sobre o tipo XML, tais como:

xmlcomment
Cria um comentário XML **<!- text ->**
xmlelement
cria um elemento com os atributos e conteúdo especificados.
Por exemplo:
xmlelement(name xpto, xmlattributes(current_date as date),'conteudo')
cria o elemento **<xpto date="2014-05-31">conteudo</xpto>**

- Linguagens Procedimentais

O PostgreSQL tem sistema de *triggers* e *stored procedures*. A definição dos procedures é realizada através da linguagem PL/pgSQL. Estão ainda disponíveis na distribuição base as linguagens PL/TCL, PL/Perl e PL/Python. No entanto, podem ser utilizadas outras linguagens procedamentais não incluídas na distribuição base.

Os próprios utilizadores podem definir uma linguagem a utilizar no sistema, sendo possível então programar triggers.

- Vários Utilizadores e Segurança

O sistema de autenticação utilizado pelo PostgreSQL é baseado em perfis. A cada utilizador está associado um perfil que indica as permissões que o utilizador tem sobre a Base de Dados.

A configuração da autenticação dos clientes é controlada por um ficheiro de configuração denominado por **pg_hba.conf .** Nesse ficheiro é possível especificar qual o método de autenticação e parâmetros para o tipo de ligação, base de dados e utilizador a conectar-se.

- Ferramentas Associadas

O PostgreSQL fornece uma aplicação de administração,o ***pgAdmin*.**

A interface gráfica da aplicação suporta todos os recursos do PostgreSQL. Permite gerir as tabelas, gerir restrições e tem à disposição uma consola SQL. Permite também navegar graficamente nas tabelas da Base de Dados.