PostgreSQL

Sistemas de Base de Dados

2013/2014

Relatório

Grupo 11
Miguel Duque nº 41808
João Fernandes nº 42168
Eduardo Cardigo nº 41790

Indíce

1 Introdução

Este relatório foi desenvolvido no âmbito da disciplina de Sistemas de Bases de Dados do Mestrado Integrado de Engenharia Informática. O trabalho consiste na descrição do sistema de base de dados *PostgreSQL*, um sistema *open source* que está disponível para várias plataformas. Também é de realçar que é um sistema bastante utilizado por grandes empresas como Yahoo, Afilias, Sony Online, Instragram, entre outros.

1.1 História

O PostgreSQL iniciou-se no final dos anos 70 com um projeto chamado *Ingres*, desenvolvido na Universidade de Berkeley, Califórnia. O líder do projeto e um dos pioneiros das bases de dados relacionais, Michael StoneBraker, decidiu sair da universidade e comercializar este sistema em 1982. Mais tarde em 1985 decidiu voltar a universidade e começar um novo projeto chamado post-Ingres, com o objetivo de resolver os problemas que surgiam nos SGBDs contemporâneos. O desenvolvimento do Postgre iniciou-se em 1986 e partilhava vários conceitos dos projetos anteriores. Em 1988 lançaram o seu primeiro protótipo com código completamente novo.

No entanto, só em 1994 é que foi criado um interpretador SQL para substituir a linguagem até então usada (QUEL) e o projeto foi renomeado Postgres95. Também foi nesta altura que o código foi disponibilizado na Web, onde iniciou uma nova vida como *software open source*.

Em 1996, Marc Fournier, Bruce Momjian e Vadim B. Mikheev lançaram a primeira versão externa da Universidade de Berkeley e o projeto foi renomeado para *PostgreSQL*. A primeira versão de *PostgreSQL* foi liberada em Janeiro de 1997. Desde então, um grupo de programadores e voluntários de todo o mundo, através da Internet, têm mantido o *software* e desenvolvido novo funcionalidades.

1.2 Aplicabilidade do sistema

O *PostgreSQL* é um sistema com capacidade suficiente para ser usado por grandes empresas que necessitam de estabilidade, desempenho e portabilidade, já que é uma solução que para além de gratuita, é leve e compatível com várias plataformas para a gestão de bases de dados. O *PostgreSQL* para além de ter as características de um sistema de bases de dados relacional, também tem algumas características de orientação a objetos, como herança e tipos personalizados, pelo que é considerado um sistema objeto-relacional.

2 Armazenamento e file structure

2.1 Sistema de ficheiros

O *PostgreSQL* utiliza o sistema de ficheiros do sistema operativo em que se encontra instalado. Os dados e os meta-dados estão guardados na diretoria PGDATA. Este conjunto de informação determina um *cluster* de base de dados. Dentro desta diretoria temos subdiretorias e alguns ficheiros de configuração, que estão descritos na tabela abaixo.

Item	Descrição
PG_VERSION	Ficheiro que contem a versão do *PostgreSQL*.
Base	Subdiretoria que contém uma subdiretoria para cada BD.
global	Subdiretoria que contém tabelas globais a todo o cluster, como por exemplo pg_database.
pg_clog	Subdiretoria que contém o estado de commit das transações.
pg_multixact	Subdiretoria que contém os dados relativos ao estado das multi-transações.
pg_notify	Subdiretoria que contém o estado de LISTEN/NOTIFY.
pg_serial	Subdiretoria que contém informação sobre transações serializáveis que foram commited.
pg_stat_tmp	Subdiretoria que contém os ficheiros temporários para o subsistema de estatísticas
pg_subtrans	Subdiretoria que contém os dados referentes ao estado das sub-transações
pg_tblspc	Subdiretoria que contem links simbólicos para tablespaces
pg_twophase	Subdiretoria que contém ficheiros de estado para transações preparadas
pg_xlog	Subdiretoria que contém ficheiros WAL(Write Ahead Log)
postmaster.opts	Ficheiro que contém a parametrização com que o servidor foi iniciado
postmaster.pid	Ficheiro de bloqueio que contém o PID do servidor e o ID do segmento de memória partilhada (não disponível depois de se desligar o servidor)

Para além dos ficheiros que já vimos, também podemos encontrar os ficheiros de configuração *postgresql.conf, pg_hba.conf e pg_ident.conf.* No entanto nas versões do PostgreSQL 8.0 e seguintes, é possível guarda-los em sítios diferentes.

Para cada base de dados criada no sistema, vai ser gerada uma subdiretoria dentro da diretoria PGDATA/base, que irá ter o nome do seu OID (Object Identifier) onde são guardados todos os ficheiros relacionados com a mesma.

Cada tabela e índex são guardados em ficheiros separados sendo estes nomeados pelo respetivo número de filenode, que é dado pelo ficheiro *pg_class.reflilenode*. Cada um destes tem ainda um *free space map*, que guarda informação sobre o espaço livre disponível e outro com informação sobre os registos que não foram eliminados (*Visibility Map*).

Quando uma tabela ou índex excede 1 GB de tamanho, é dividido em segmentos com um tamanho definido (por omissão 1GB mas pode reajustar-se usando a opção de configuração with-segsize). O nome do ficheiro do primeiro segmento é igual ao *filenode*, mas os segmentos seguintes são nomeados *filenode.1, filenode.2, etc*. Esta solução evita problemas em plataformas que tem limitações nos tamanhos dos ficheiros.

2.2 Paginação

As páginas são a estrutura de dados utilizada para armazenar as tabelas e índices. Estas páginas têm um tamanho fixo de 8KB. Em uma tabela, todas as páginas são logicamente equivalentes, logo um tuplo pode ser guardado em qualquer página. Em relação aos índices, a primeira página está reservada como *metapage* para guardar a informação de controlo. De seguida temos a estrutura da uma página.

Item	Descrição
PageHeaderData	O seu tamanho é 24 bytes. Contém informação geral sobre a página, incluindo os apontadores disponíveis.
ItemIdData	Vector de pares a apontar para os items actuais. 4 bytes por item.
Free space	O espaço por alocar. Apontadores para novos itens são alocados no início deste espaço, novos itens desde o fim.
Items	Os próprios itens.
Special space	Espaço especial, existente para acessos por índice. Vazio em tabelas normais.

2.3 Partições

O particionamento de informação de uma base de dados serve para dividir uma tabela de grande dimensão em diversas partes, essencialmente para melhorar o desempenho da base de dados. Este mecanismo permite melhorar o tempo de acesso às tabelas e também na atualização de uma parte da tabela presente em uma partição.

O PostgreSQL implementa partições via herança de tabelas, pelo que cada partição é criada como filha de um único pai. Existem duas formas de realizarmos a partição:

- Range Partitioning:

  A tabela é particionada em intervalos definidos por uma coluna chave ou por um conjunto de colunas.
- List Partitioning:

A tabela é particionada listando que valores chave irão aparecer em cada partição.

2.3.1 Implementar partições

Apresentamos de seguida um exemplo que mostra como conseguimos fazer particionamento em vários passos. Em primeiro é necessário criar uma tabela mestre, da qual as partições vão herdar:

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

Em segundo lugar vamos criar várias tabelas filhas que vão herdar da tabela pai. Normalmente estas tabelas não vão adicionar nenhuma coluna ao conjunto herdado da tabela pai:

```
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);
```

O terceiro passo será adicionar *constraints* as tabelas particionadas que permitam decidir que tuplo pertence a cada partição:

```
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);
```

Para cada partição vamos criar um índex sobre a coluna chave da mesma:

```
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);
```

Em último lugar e opcionalmente, criar um trigger que facilite a inserção de dados nas partições. Este trigger em vez de inserir os dados na tabela original, vai introduzir os dados na respectiva partição:

```
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 '2008-
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;
```

2.3.2 Gerir partições

É comum querer alterar ou remover partições que foram criadas. Vamos agora mostrar alguns exemplos de como gerir as nossas partições.

Uma opção simples para remover dados antigos é simplesmente fazer "*drop*" da partição que já não precisamos. Este comando consegue apagar uma grande quantidade de dados rapidamente, já que não apaga individualmente cada tuplo.

```
DROP TABLE measurement_y2006m02;
```

Se o que pretendemos é remover a partição da estrutura de partições, mas manter a tabela, podemos desligar a partição da sua respetiva tabela pai.

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

Também podemos criar uma nova partição para gerir dados que são novos. Simplesmente criamos uma nova partição na estrutura das partições como as partições que foram criadas originalmente.

```
CREATE TABLE measurement_y2008m02 (
    CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' )
) INHERITS (measurement);
```

2.4 Buffer Management

O sistema *PostgreSQL* possui o seu próprio buffer manager e este depende do sistema operativo onde se encontra instalado. A função principal deste buffer é controlar a quantidade e a forma como a informação é guardada em disco. O *PostgreSQL* não escreve diretamente nos blocos do disco, ele utiliza um buffer cache partilhado, que por omissão tem 32MB. Podemos alterar este valor se acedermos ao ficheiro de configuaração *postgresql.conf,* através da variável *shared_buffers.* No mesmo ficheiro também podemos encontrar a variável *temp_buffers* com o valor definido por defeito de 8MB, e este valor representa o tamanho máximo de memória alocada para os buffers temporários que cada base de dados tem para cada sessão. Este valor também pode ser modificado caso o utilizador o queira fazer.

Cada buffer tem associado uma flag que representa se o buffer esta a ser utilizado pelo sistema ou não. Através da função *bufferaloc*, um processo obtém um buffer, e se o mesmo não estiver a ser utilizado por outro processo é-lhe associada a flag *pinned*. Uma vez que deixe de usar este buffer, a flag volta a estar *unpinned.*

O PostgreSQL decide que informação deve carregar em memória e que informação libertar em função do seu estado de atualização e da frequência que a informação que contém é acedida. Esta estratégia é chamada LRU (least-recently-used). Assim sempre que houver informação nova que seja necessária carregar em cache, o sistema vai substituir a informação com o menor número de acessos (realizadas através de clock sweep).

2.5 Clustering

Uma técnica de otimização disponível no sistema *PostgreSQL* é o clustering por tabela, que consiste em reordenar fisicamente uma tabela com base na informação de um índice. Para implementar este mecanismo utilizamos o comando CLUSTER, sendo necessário especificar uma tabela e o respetivo índice. Um exemplo de este comando:

```
CLUSTER [VERBOSE] table_name [ USING index_name ]
```

Depois de uma tabela ter sido clustered, o utilizador poderá ordenar novamente a tabela sem ter que referenciar o índice de novo, fazendo o comando:

```
CLUSTER tablename
```

Ou simplesmente reordenar todas as tabelas que tenham sido previamente clustered através do seguinte comando:

```
CLUSTER
```

3 Indexação e hashing

3. Indexação e Hashing

A indexação é um mecanismo utilizado para otimizar o desempenho de uma base de dados. Permite obter linhas de uma tabela de uma forma mais rápida ao invés de fazer uma pesquisa inteira à tabela. Depois de criado o índice, o sistema manterá o índice atualizado sempre que a tabela seja alterada. Este índice será utilizado pelo sistema sempre que melhore o desempenho dos comandos a executar, o que irá causar um maior *overhead* à base de dados. Por tanto é preciso ter em atenção se o índice está a ser usado apropriadamente ou não. Vamos ver no seguinte exemplo, criando uma tabela previamente:

```
CREATE TABLE table1
{
Id integer,
Name varchar
};
```

Ao fazer uma consulta do tipo:

```
SELECT Name FROM table1 WHERE id = constant
```

Admitimos que sem qualquer tipo de indexação, o sistema seria obrigado a percorrer os tuplos sequencialmente à procura da entrada com o id desejado. A solução passa por criar um índice para a chave primaria, visto que a consulta esta dependente do campo id.

Para a criação de índices, o comando é o seguinte:

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table [ USING method ]

    ( { column | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC |
DESC ] [ NULLS { FIRST | LAST } ] [, ...] )

    [ WITH ( storage_parameter = value [, ... ] ) ]

    [ TABLESPACE tablespace ]

    [ WHERE predicate ]
```

Este código é utilizado para criar um índice com o nome ***name*** na tabela ***table***, que usa o tipo de índice ***method***, que já veremos posteriormente que tipos pode ter. Vamos agora estudar os vários parâmetros deste código:

- UNIQUE: Faz com que o sistema confirme que não existem valores duplicados na tabela já existente e nos novos tuplos adicionados. Caso existam valores duplicados, o sistema irá apaga-los.

- CONCURRENTLY: Se esta opção é utilizada, o sistema *PostgreSQL* vai criar um índice sem ter qualquer atenção a inserções, updates ou deletes concorrentes nessa tabela.

- Method: O tipo de índice a ser utilizado. Podemos escolher entre B-Tree, Hashing, Generalized Search Trees e Generalized Inverted Index. Os comandos são respetivamente *btree*, *hash*, *gist* e *gin*.

- Opclass: Esta classe identifica os operadores a serem utilizados pelo índice na coluna pretendida. Por exemplo o nome int4_ops, no caso de um índice sobre um atributo inteiro de 4 bytes.

- Storage_parameter: Nome de um parâmetro do armazenamento específico do índice escolhido.

- Predicate: É uma expressão que limita o índice.

No caso de querermos alterar um índice, temos que fazer o seguinte comando:

```
ALTER INDEX name RENAME TO new_name

ALTER INDEX name SET TABLESPACE tablespace_name

ALTER INDEX name SET ( storage_parameter = value [, ... ] )

ALTER INDEX name RESET ( storage_parameter [, ... ] )
```

O primeiro comando é utilizado para renomear o índice criado. No segundo comando mudamos o *tablespace* para o que pretendemos. E os dois comandos seguintes são para alterar parâmetros de armazenamento do índice dado.

Caso queiramos apagar um índex da base de dados, teremos que fazer o seguinte comando:

```
DROP INDEX [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Apenas de salientar que o comando IF EXISTS é utilizado para não causar um erro no sistema, de modo a ser mostrada uma notificação caso o índice não exista. E os comandos CASCADE/RESTRICT, o primeiro remove todos os objetos dependentes do índice e o segundo não permite que tal aconteça. Por omissão, é executado o comando RESTRICT.

3.1 Tipos de Índices

Como foi referido anteriormente, o sistema *PostgreSQL* utiliza 4 tipos de índices: B-Tree Hashing, Generalized Search Trees e Generalized Inverted Index. Vamos agora estudar com mais detalhe cada um de estes índices.

3.1.1 B-Tree

As B-Tree é a estrutura utilizada por defeito na criação de um índice se nenhuma estrutura for especificada. Este tipo de árvore representa dados ordenados de maneira a permitir inserções, remoções e consultas de forma eficiente. Este índice lida com operadores =, <, <=, >=, >. Construções equivalentes a combinações destes operadores como IS NULL e BETWEEN IN também podem são suportadas por este índice.

3.1.2 Hash

O índice hash apenas consegue lidar com comparações de igualdade, o que limita o uso deste tipo de índices. O optimizador só irá considerar a utilização deste tipo de índice quando uma coluna indexada é indexada em uma comparação que utilize o operador =. No entanto, este tipo de indexação é relativamente pior que a indexação por B-tree, que também pode ser utilizada em comparações de igualdade. Para além de utilizar mais recursos, o hash não suporta índices de múltiplas colunas.

3.1.3 Generalized Search Tree (GiST)

Os índices GiST não são apenas um índice, mas sim uma interface que pode ser utilizada para implementar vários tipos de árvores como B-trees e R-trees, e também para outros exemplos como para cubos multidimensionais e pesquisa total de texto. Para criar um índice GiST será necessário definir as classes de operadores, incluindo operadores de comparação e estratégias de pesquisa.

3.1.4 Generalized Inverted Index (GIN)

Os índice GIN são capazes de lidar com valores que contém mais do que uma chave, como por exemplo vetores. A semelhança dos índices GiST, os índices GIN suportam varias estratégias de indexação, definidas pelo utilizador, bem como os operadores a usar.

Se compararmos com o GiST, ambos são utilizados para acelerar pesquisas totais de texto, mas a nível de desempenho, vamos ter algumas diferenças. As procuras no índice GIN vão ser mais rápidas que no GiST, já a sua criação vai ser mais lenta. As atualizações nos índices GIN vão ser um pouco mais lentas e o tamanho que ocupa vai ser maior que no GiST.

3.2 Índices Multicoluna

No sistema *PostgreSQL* um índice pode ser definido usando mais do que uma coluna de uma tabela. No entanto esta técnica apenas é suportada pelos tipos B-Tree, GiST e GIN. Por defeito podem ser especificadas até 32 colunas, embora este limite possa ser alterado. Devemos ter algum cuidado no momento de criação de índices deste tipo visto que há situações onde não irá compensar a sua utilização.

No índice B-Tree a eficiência de pesquisas sobre instâncias multicoluna do mesmo é maior quando as condições de pesquisas são sobre as colunas indexadas mais à esquerda do índice.

A eficiência nos índices GiST depende do número de valores distintos da primeira coluna, mesmo que hajam vários valores diferentes nas colunas adicionais.
Já no caso dos índices GIN, a eficiência das pesquisas é a mesma independentemente das colunas, indexadas pelo índice, usadas nas condições das mesmas.

3.3 Multiplos índices

O sistema *PostgreSQL* também suporta a implementação de múltiplos índices numa só tabela para situações em que apenas um índice não é suficiente para tratar da situação. Por exemplo, em uma pesquisa num índice onde os atributos indexados são unidos com AND, um índice serve. Mas caso seja utilizado um OR não se pode utilizar diretamente o índice.

Para combinar múltiplos índices, o sistema pesquisa cada índice necessário e prepara um bitmap em memória que indica quais os tuplos que respeitam a condição. Os bitmaps são depois submetidos a AND e OR conforme a *query*. Para obter os dados, as linhas são visitadas por ordem física porque os bitmaps não conservam a ordenação original dos índices. Caso de o resultado tenha que ser ordenado devido a uma cláusula ORDER BY, será necessário efetuar uma operação de ordenação.

3.4 Inconsistência Temporária

As estruturas no sistema *PostgreSQL* podem encontrar-se num estado insconsistente durante um período de tempo, em particular durante uma transacção.

Para permitir a verificação de restrições em diferido, existe o comando SET CONSTRAINTS. No entanto, durante a criação ou alteração das tabelas, é necessário definir as restrições como DEFERRABLE. Por defeito todas as restrições são NOT DEFERRABLE, o que significa que a sua verificação não pode ser adiada. Se é definida com a cláusula IMMEDIATE então todas as restrições são imediatamente verificadas após cada instrução. Fica aqui a estrutura do comando:

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

3.5 Comparação com Oracle

Em relação à indexação no sistema *PostgreSQL*, é semelhante ao Oracle, no entanto suporta outros tipos de indexação como o GiST e o GIN. O *PostgreSQL* assume que escolhe sempre o melhor método de otimização disponível, o que impede o utilizador de forçar o uso de um índice numa *query*.

4 Processamento e otimização de perguntas

O processamento e otimização de perguntas é um mecanismo fundamental num SGBD, uma vez que é o responsável por processar as querys (perguntas do utilizador) e apresentar os resultados o mais rapidamente possível.
A eficácia deste mecanismo é medida pelo tempo que leva a responder ao utilizador, e por isso há a necessidade de existência de um optimizador.
A função do optimizador é, com base em informação e estatísticas da BD, estimar e comparar os tempos de resposta de todos os planos possíveis, de forma a tentar executar o plano que tiver menor custo.

4.1 Ciclo de vida de uma pergunta

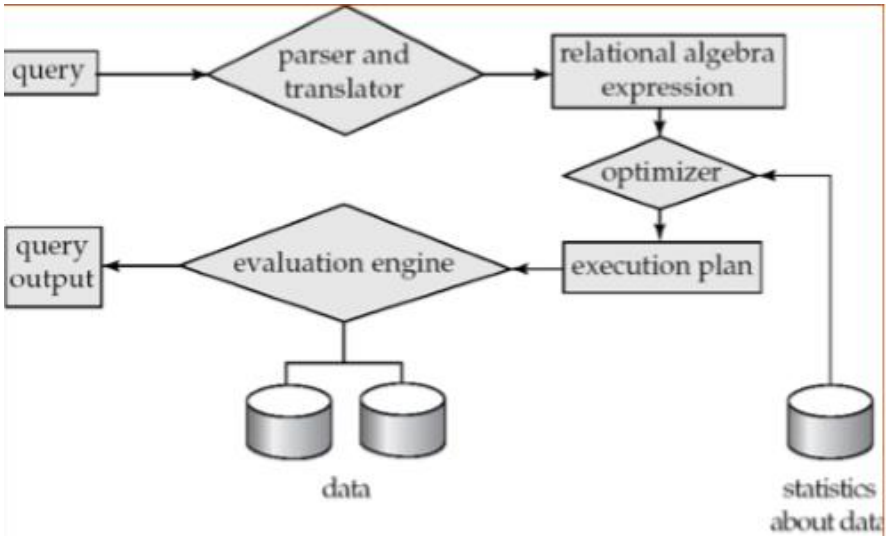

Figura 4.1 – Percurso de uma interrogação ao sistema

Parser - verificação da pergunta ao nível de sintaxe, e criação de uma query tree.

Rewrite system - recebe query tree e verifica se existe alguma regra a aplicar sobre esta.

Planeador/optimizador - constrói o melhor plano de execução

Executor - executa o plano de execução e devolve os tuplos do resultado

4.1.1 Fase de Parser

O parser, que está definido em "gram.y" comeca por receber uma query string (em plain text) que é testada ao nível da sintaxe, utilizando o "scan.l" para criar uma "parser tree". Se a "parser tree" estiver sintaticamente correta, é feita uma verificação semântica (verificar as tabelas, funções e operadores que a *query* necessita) e é criada uma query tree, ou caso contrário, é devolvida uma mensagem de erro.

Query Tree

Uma *Query Tree* é uma representação interna de um comando SQL, em que todos os seus constituintes estão guardados separadamente. As *query trees* criadas pelo servidor podem ser consultadas nos logs se forem activados os parâmetros *debug_print_parse*, *debug_print_rewritten* ou *debug_print_plan*.

Cada *query tree* é constituída pelos seguintes elementos:

- **Command Type -** indicação do comando produzido pela query (SELECT, INSERT, UPDATE ou DELETE)
- **Range Table -** identifica a tabela ou vista sobre a qual a consulta recai, e os nomes das outras partes da *query* (relações existentes na consulta). Por exemplo para uma instrução SELECT são as existentes a seguir à instrução FROM.
- **Result relation -** são os índices da Range Table (relação para onde irão os dados da query). SELECT queries não têm *Result relation* mas para as instruções de INSERT, DELETE e UPDATE, a *result relation* é a tabela/vista onde as alterações são feitas.
- **Target List -** lista que recebe os resultados da consulta efetuada. O conteúdo da Target List para as diferentes operações vai ser:
    1. Para a operação SELECT são as expressões do resultado final.
    2. Para a operação DELETE não produz resultados, no entanto é adicionado o CTID das linhas a serem eliminadas (localização física da linha dentro da tabela) para permitir ao executor encontrar as linhas a serem apagadas.
    3. Para a operação INSERT são as novas linhas a serem adicionadas na relação.
    4. Para a operação de UPDATE são os tuplos a serem atualizados e quais os seus novos valores.
- **Qualification -** corresponde à clausula WHERE, é um booleano que indica se a instrução INSERT DELETE UPDATE ou SELECT deve ser aplicável ao resultado final.
- **Join Tree -** corresponde à estrutura da clausula FROM. Para consultas simples como (SELECT * FROM a, b, c) é apenas a lista dos elementos do FROM porque o join pode ser feito em qualquer ordem. Para consultas que forcem a ordem dos *Joins* (como OUTER JOIN), *a Join Tree* representa a estrutura das expressões de junção. As restrições dos *joins* como clausulas On ou USING são guardadas como *Qualification*
- **Others -** restantes instruções existentes na consulta (como ORDER BY).

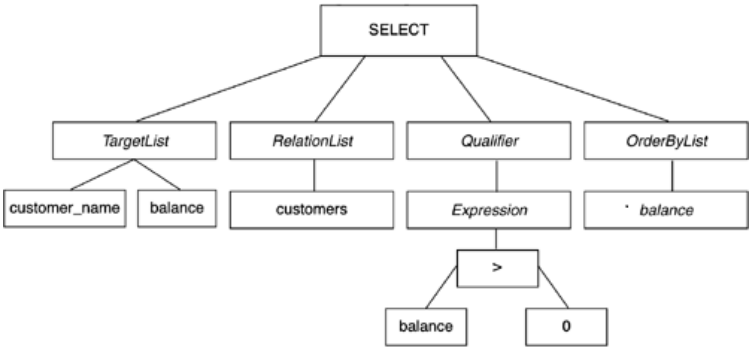

Figura 4.2 – Exemplo de uma Query Tree

4.1.2 Rewrite System

Caso o parser tenha sucesso, a *query tree* gerada é entregue ao *Rewrite System* que, caso existam regras definidas pelo utilizador, cria *query trees* baseadas nessas regras, para serem utilizadas pelo planeador. Para se consultar as regras a serem aplicadas, pode-se consultar o *pg_rewrite* no servidor.

4.1.3 Planeador/optimizador

Nesta fase, o objetivo é encontrar o melhor plano possível (plano com menor custo) para executar a *query*. Como se sabe, existem várias formas de executar uma *query* e obter o mesmo resultado, por isso, caso seja possível, o planeador/optimizador olha para todas as alternativas e estima o seu custo de execução. Caso não seja possível (analisar todas as alternativas consumia demasiado tempo e memória) o *PostgreSQL* utiliza uma ferramenta chamada *Genetic Query Optimizer*. Esta ferramenta é utilizada quando o número de items em FROM ultrapassa um limite, definido pelo *geqo_threshold* que por defeito, tem o valor 12.

Geração de planos possíveis

O planeador/optimizador começa por gerar planos que scannem cada tabela individual da *query*, quer através da utilização dos diferentes índices existentes, quer por scan sequencial. Os planos são gerados das seguintes formas:

- Planos para pesquisas sequenciais
- Para atributos na clausula WHERE que também sejam a key de um índice
- Para as clausulas ORDER BY que tenham índices de ordenação
- Planos para fazer join de duas ou mais relações, para os quais existem 3 estratégias de join (nested loop join, merge join, hash join)

Genetic Query Optimizer

Como se sabe, otimizações de junções são das tarefas mais complicadas a realizar por um SGBD, uma vez que o número de planos possíveis para uma dada consulta cresce exponencialmente para cada nova junção. Por esta razão, o *PostgreSQL* utiliza uma estratégia usada em Inteligência Artificial baseada em algoritmos genéticos de forma a conseguir o melhor plano para um número muito elevado de junções. A implementação do GEQO no *PostgreSQL* tem as seguintes características:

- O uso de um algoritmo genético de estado estacionário (substituição apenas dos indivíduos com menos aptidão na população, não se substitui a geração inteira)

permite a rápida convergência para planos de consulta melhorada. Isto é essencial para a manipulação de consultas em tempo razoável.

- O uso de recombinação das arestas de modo a minimizar a perda de arestas para a solução do "caixeiro-viajante" por meio de um algoritmo genético.
- A mutação é desaconselhada como operador genético, o que leva a que não sejam necessários mecanismos de reparo para gerar circuitos válidos para o problema do caixeiro-viajante.

Geração de Planos Possíveis para o GEQO

O processo de planificação do GEQO usa o código padrão do planeador para gerar planos de pesquisa das relações individuais. De seguida, os planos de junções são desenvolvidos utilizando a abordagem genética.

Cada plano de junção candidato é representado por uma sequência e para cada uma destas sequências de junções, o código do planeador é usado para estimar o custo de execução da pergunta usando essa sequência.

Para cada passo da sequência de junções são considerados os três métodos de junção, estando também disponíveis todos os planos de pesquisa determinados inicialmente para as relações. O custo estimado consiste no menor custo de entre todas estas possibilidades.

Uma sequência de junções com menor custo diz-se "com mais aptidão" que uma com custo mais elevado, sendo descartadas pelo algoritmo genético as sequências candidatas com menor aptidão, e criadas novas candidatas através da combinação de genes de sequências com maior aptidão. A escolha dos genes é feita através da escolha aleatória de porções das sequências. O processo de descartar as piores e combinar os genes das melhores é repetido até que tenha sido considerado um número predefinido de sequências de junções.

Por fim a melhor sequência encontrada durante toda a procura é usada para gerar o plano final.

4.1.4 Executor

O executor ao receber o plano dado pelo planeador, começa por executa-lo recursivamente, processando-o para extrair os tuplos pretendidos. Sempre que um nó do plano é chamado(de baixo para cima), tem de devolver um ou mais tuplos, ou reportar que já terminou. Este mecanismo também é utilizado para avaliar os tipos básicos de Query SQL (SELECT, INSERT, UPDATE E DELETE):

- **INSERT -** cada linha devolvida é inserida
- **DELETE -** devolve apenas os identificadores dos tuplos a serem removidos e remove-os efetivamente.
- **UPDATE -** devolve o valor das colunas a serem atualizadas (estes valores serão usados para criar um novo tuplo) e devolve o identificador dos tuplos para serem removidos.

4.2 Algoritmos utilizados nos vários tipos de operações

4.2.1 Seleção

Sequential Scan

Este é o algoritmo mais básico das querys e por isso é sempre gerado o seu *query plan*. O funcionamento deste algoritmo passa por percorrer todas as linhas de uma tabela, e, por cada linha, verifica se a condição é respeitada. Para ativar ou não o uso deste algoritmo usa-se *enable_seqscan(boolean)*.

```
EXPLAIN SELECT * FROM tenk1;
                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

Index Scan

Este algoritmo baseia-se numa pesquisa sobre um índice. Ao contrário do *Sequential Scan*, este algoritmo não percorre todos os tuplos da tabela se lhe providenciarmos valores de inicio e/ou fim. Todavia, o nº de *seeks* neste algoritmo pode mais elevado do que o *Sequential Scan*, uma vez que os valores são retornados na ordem do índice e não pela ordem sequencial. Para ativar ou desativar usa-se o *enable_indexscan(boolean).*

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.29..8.30 rows=1 width=244)
   Index Cond: (unique1 = 42)
```

Bitmap Index Scan

Este algoritmo utiliza os diferentes índices criados numa dada tabela para otimizar e aumentar a velocidade de pesquisas mais complexas. Para ativar ou desativar usa-se o *enable_bitmapscan(boolean)*.

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 = 'xxx';

                                 QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.04..229.43 rows=1 width=244)
   Recheck Cond: (unique1 < 100)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

4.2.2 Junção

Neste-Loop Join

Este algoritmo consiste em comparar todos os tuplos da primeira relação com todos os da segunda (*scan* à tabela do lado direito por cada tuplo da tabela do lado esquerdo) e caso satisfaça as condições de junção, esses pares de tuplos são retornados. Este algoritmo é bastante simples e pode sempre ser utilizado, no entanto, pode ter custos bastante elevados. O custo torna-se menor nos casos em que ambas as relações cabem em memória ou quando existem índices nos atributos de junção em pelo menos uma das tabelas. Para o planeador utilizar o algoritmo, *fazer enable_nestloop(boolean);*

Merge Join

Este algoritmo só pode ser utilizado se for possível percorrer ambas as relações pela ordem do atributo de junção (usando um índice ou ordenando previamente as relações se necessário), em junções naturais e junções de igualdade. Caso seja possível, as duas relações são percorridas de forma idêntica, quase "paralela", visto que se encontram ordenadas. Um dos pontos interessantes desta abordagem é o facto de percorrer cada tabela uma única vez. Para o planeador utilizar o algoritmo, deve fazer *enable_mergejoin(boolean)*;

Hash Join

Este algoritmo também so pode ser aplicado a junções naturais e junções de igualdade. Este algoritmo consiste em criar partições de ambas as relações "r" e "s" , usando como chave de *hash* os valores do atributos de junção. De seguida efectuar a junção em cada partição "r" e "s" (usando correspondências por *hash keys*). Para o planeador utilizar o algoritmo, fazer *enable_hashjoin(boolean);*

4.2.3 Ordenação

Começamos por distinguir os dois domínios onde pode ser realizada ordenação:

- na "main memory" em que não há necessidade de escrever resultados intermédios em disco (ambas as relações cabem em memória) e por isso pode apenas ser usado o *Quicksort*
- quando o volume de dados obriga a que sejam escritos resultados intermédios em disco. Neste caso o algoritmo a ser usado é o *External Merge Join* que divide o conjunto de entrada em divisões que caibam em memória, ordena cada uma das divisões e, posteriormente, processa a sua junção;

A função *enable_sort(boolean)* activa/desactiva a utilização de ordenação. É impossivel desactivar inteiramente a ordenação, mas desactivando-a faz com que o optimizador considere outras opções.

4.3 Avaliação de Expressões Completas

Materialização

Consiste em guardar os resultados das expressões em relações temporárias da base de dados de forma a poderem ser utilizados em avaliações de operações dos níveis superiores. Planos de execução que usem o *Nested Loop Join* podem utilizar Materialização, em vez de executar várias vezes o plano da relação da direita.
O comando *enable_material (boolean)* activa/desactiva o uso de materialização por parte das queries. É impossível suprimir a materialização por completo, mas ao desativar, previne que sejam inseridos "materialize nodes", exceto em casos que seja necessário maior exatidão.

Pipelining

Consiste na passagem de tuplos para as operações acima durante a execução da operação corrente.

Apesar de dispinibilizar os dois mecanismo, não permite ao utilizador escolher o mecanismo a usar, ficando essa escolha à responsabilidade do planeador/optimizador que sempre que possível, utiliza o *pipelining*.

4.4 Mecanismos para Visualização de Planos

Como já indicámos, é possível visualizar os planos de execução dada uma consulta SQL através do comando *Explain*. A sintaxe do comando é a seguinte:

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement

where option can be one of:

    ANALYZE [ boolean ]
    VERBOSE [ boolean ]
    COSTS [ boolean ]
    BUFFERS [ boolean ]
    TIMING [ boolean ]
    FORMAT { TEXT | XML | JSON | YAML }
```

Analyze – esta opção para além de planear a consulta em causa, executa-a efetivamente. Neste caso o custo real da consulta é disponibilizado;
Verbose – inclui no output do explain a árvore de planeamento construída;
Statement – qualquer querry SQL;

```
EXPLAIN SELECT * FROM tenk1;

                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

Figura 4.3 – Exemplo de um *Query Plan*

Os valores apresentados pelo EXPLAIN são os seguintes (da esquerda para a direita):

- Custo estimado antes do inicio do output. Consiste no tempo gasto antes da fase de devolução de resultados (por exemplo fazer a ordenação prévia de uma tabela)
- Custo estimado para a operação
- Numero estimado de tuplos de resultado para a operação
- Largura estimada em bytes para os tuplos do resultado da operação

4.5 Estatísticas utilizadas pelo Planeador

O *planner* do *PostgreSQL* precisa de estimar o número de linhas que uma determinada consulta irá devolver de forma a conseguir boas estimativas e aproximações para obter o melhor plano de execução. Para efectuar tais análises, é necessário guardar valores estatísticos sobre os dados da base de dados.

As informações que guarda são as seguintes, número de tuplos da tabela e o número de disk blocks que a tabela ocupa em disco, os índices e o espaço que ocupam em disco. Esta informação é guardada na tabela pg_class nas colunas reltuples e relpages e pode ser acedida com a query SELECT * FROM pg_stats;

```
       relname         | relkind | reltuples | relpages
-----------------------+---------+-----------+----------
 tenk1                 | r       |     10000 |      358
 tenk1_hundred         | i       |     10000 |       30
 tenk1_thous_tenthous  | i       |     10000 |       30
 tenk1_unique1         | i       |     10000 |       30
 tenk1_unique2         | i       |     10000 |       30
```

Figura 4.4 – Exemplo de estatísticas utilizadas

Por razões de eficiência, reltuples e relpages não são actualizados instantaneamente, ou seja, é possível existir alturas em que estes estejam ligeiramente desatualizados. Para tal, é possível, através de certos comandos, atualizar estes dados estatísticos, tais como VACUUM, ANALYZE e CREATE INDEX. No caso do VACUUM, esta ferramenta serve para limpar os dados que foram apagados da base de dados mas que continuam em memória. Ou seja, é feita uma limpeza dos dados.

As consultas que utilizam a cláusula WHERE, retornam apenas o conjunto de tuplos que respeitam a cláusula. A informação usada para esta tarefa guarda-se na tabela *pg_statistic*. Esta guarda informação detalhada sobre os dados das tabelas, como a percentagem de nulls, diferentes valores, tamanho médio dos valores de cada coluna etc. Os dados nesta são actualizados também com ANALYSE e VACUUM ANALYZE, e são sempre aproximados, mesmo com updates recentes.

Exemplo do uso de estatísticas por parte do planner

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;

                                 QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=24.06..394.64 rows=1007 width=244)
   Recheck Cond: (unique1 < 1000)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007
width=0)
         Index Cond: (unique1 < 1000)
```

Neste exemplo o planner examina a cláusula WHERE e verifica o valor da *selectivity function* para o operador '<' na tabela pg_operator. O valor da função é guardado na coluna oprrest e, neste caso, a entrada é scalarltsel. A função vai devolver um histograma para *unique1* de pg_*statistics*. O histograma pode ser obtido a partir de uma *query* manual à view *pg_stats*:

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='unique1';

                    histogram_bounds
------------------------------------------------------
 {0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995}
```

O histograma faz uma distribuição linear dos valores no interior de cada *bucket*, por isso a única coisa a fazer é localizar o bucket que satisfaz a nossa condição e retirar desse *bucket* os tuplos com os valores que interessam, mais todos os tuplos dos buckets anteriores.
A função de seleção pode ser calculada da seguinte maneira:

```
selectivity = (1 + (1000 - bucket[2].min)/(bucket[2].max -
bucket[2].min))/num_buckets
            = (1 + (1000 - 993)/(1997 - 993))/10
            = 0.100697
```

O número estimado de linhas pode depois ser calculado com o o produto da função de selecção com a cardinalidade de *tenk1*:

```
rows = rel_cardinality * selectivity
     = 10000 * 0.100697
     = 1007  (rounding off)
```

4.6 Comparação com o Oracle 11g

O *PostgreSQL* investe mais que o *Oracle* no processamento de consultas grandes e por isso apresenta mais algoritmos de optimização que o *Oracle*. O problema de suportar mais algoritmos é o número de combinatórias que surgem quando se pretende fazer uma junção. Por vezes, apesar de encontrar um plano melhor que o escolhido pelo *Oracle*, acaba por demorar mais tempo a chegar a esse plano.

Por outro lado, o *PostgreSQL* permite activar/desactivar mais flags que o Oracle, o que pode permitir a utilizadores mais experientes optimizar as consultas ao máximo.

5 Gestão de transacções e controlo de concorrência

Na maioria das vezes, uma base de dados não está projetada para ser utilizada apenas por uma pessoa, por essa razão torna-se essencial que o SGBD em uso garanta a integridade dos dados no acesso concorrente. Um transação é a manipulação dos dados sobre a base de dados que no fim da operação garante a consistência destes. Para que tal garantia seja possível, as transações de SGBD deve respeitar as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade).

5.1 Transações

As transações no PostgreSQL começam com uma instrução COMMIT, seguidas de um conjunto de operações que vão ser executadas atomicamente, e termina com a instrução END. Apesar de não suportar nested transactions (transações embutidas noutras), suporta SAVEPOINTS com os quais podemos obter o mesmo efeito. Esta instrução pode ser colocada dentro de uma transação permitindo que, caso seja feito rollback, a base de dados é recuperada para o estado antes da instrução SAVEPOINT.

```
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;
```

Figura 5.1 – Exemplo do uso de *SAVEPOINT*. A transação insere os valores 1 e 3.

5.2 Controlo de Concorrência

O PostgreSQL utiliza dois mecanismos para lidar com a consistência dos dados, o MultiVersion Concurrency Control(MVCC) - o mais usado, pois geralmente obtém melhor performance, e o controlo de concorrência baseado em locks.

5.2.1 MultiVersion Concurrency Control(MVCC)

Este é o protocolo usado por omissão no PostgreSQL devido ao facto de se obterem melhores resultados com este protocolo. Internamente, a consistência dos dados é mantida através de do modelo multi-versões, ou seja, ao questionar a base de dados, cada transação vê a versão da base de dados (snapshot) de um dado momento anterior independentemente do estado atual. Desta forma é garantido Isolamento da transação uma vez que estas são impedidas de verem dados inconsistentes, que eventualmente seriam causados por transações concorrentes sobre os mesmos dados. A maior vantagem deste protocolo, é que os seus querie locks de leitura - escrita não são conflituosos, ou seja, leitura não bloqueia escrita, escrita não bloqueia leitura. O PostgreSQL consegue garantir esta propriedade mesmo no nível mais rigoroso de isolamento de transações, através do uso de um nível inovador de Serializable Snapshot Isolation (SSI).

Assim, ao evitar os métodos baseados em locks usados pelos SGBD mais tradicionais, este sistema proporciona um melhor desempenho em ambientes com vários utilizadores.

5.2.2 Locks Explícitos

O PostgreSQL fornece vários níveis de locks a fim de garantir um controlo de concorrência sobre as tabelas existentes na base de dados. Estes modos surgem na necessidade de existirem protocolos de controlo quando existem aplicações que não se adaptem ao comportamento do MVCC.

A maioria dos comandos do PostgreSQL adquirem locks automaticamente para garantir que tabelas referenciadas não são destruídas ou alteradas para um estado incompatível enquanto os comandos executam.

Existem três níveis de locks no PostgreSQL, Table-Level Locks e Row-Level Locks para aplicações que geralmente não necessitam de completa isolação da transação e preferem gerir pontos particulares de conflito, e Advisory Locks que têm mecanismo para obter locks que não estão apenas associados a uma transação. Para examinar a lista dos locks em uso no servidor, pode-se consultar a view pg_locks.

Table-Level Locks

Para se criarem locks ao nível de tabelas, devem-se executar a seguinte instrução:

```
LOCK [ TABLE ] [ ONLY ] name [ * ] [, ...] [ IN lockmode MODE ] [ NOWAIT ]
```

Parâmetros:

- **Name -** Nome da table a fazer lock. Se ONLY for especificado, apenas essa é feito um lock sobre essa tabela, caso contrario, são feitos locks sobre a table e todas as suas descendentes.
- **Lockmode -** especifica que locks é que vão estar em conflito com este novo lock ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
- **NOWAIT -** Indica que o lock não deve esperar que os locks em conflitos sejam libertados. Caso não seja possível adquirir de imediato, a transação é abortada.

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- Do ROLLBACK if record was not returned
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

Figura 5.2 – Exemplo de um *Share Lock* numa tabela

Requested Lock Mode	Current Lock Mode							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

Figura 5.3 – Tabela que especifica os conflitos entre os diferentes tipos de locks

Row-LevelLocks

Para além de locks em tabelas, é possível fazer locks a tuplos. Estes locks podem ser partilhados (shared locks) ou exclusivos (exclusive locks).
Os locks exclusivos são adquiridos automaticamente quando o tuplo é actualizado ou apagado, e são mantidos até ao fim da transação (commit ou rol back). No entanto, estes locks não afectam as consultas, apenas bloqueiam as escritas num mesmo tuplo.
Para obter um lock exclusivo deve ser usado o comando SELECT FOR UPDATE.
Para obter um lock partilhado deve ser usado o comando SELECT FOR SHARE. Este tipo de lock não impede outras transações de obterem locks partilhados sobre o mesmo tuplo, mas impede os locks exclusivos.

Advisory Locks

Este tipo de locks permite ser feito ao nível da aplicação (ao contrario dos outros que apenas permitem ao nível de transação). Como o nome indica, o sitema não forca o seu uso, é da responsabilidade da aplicação o seu bom uso.
Há duas maneiras de adquirir este tipo de locks:
-Ao nível da sessão - locks são mantidos até serem libertados explicitamente ou até a sessão terminar. Ao contrario dos standard locks, os locks ao nível da sessão não seguem as mesmas regras de lock/unlock. Um lock adquirido durante uma transação que mais tarde seja rolled back, irá continuar a ser controlado pela transação. Para cada pedido de lock, tem que haver um pedido de unlock (rollbacks não o fazem automaticamente).
-Locks ao nível de transação - têm um comportamento regular, são automaticamente libertados no fim das transações e não há comando explicito de unlock.

Pedidos ao nível de sessão e transação para o mesmo Advisory Lock bloqueiam-se um ao outro da seguinte maneira, se a sessão que faz o pedido já controla o Advisory lock, os pedidos irão sempre suceder, mesmo que outras sessões estejam à espera do lock. Isto acontece independentemente de quem controlar/pedir o lock estar ao nível de sessão ou transação.

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345;
```

Figura 5.4 – Exemplo de *advisory lock*

Deadlocks

Deadlock é um problema comum a todos os tipos de locks e ocorrem quando duas transações ficam indefinidamente à espera uma da outra. O PostgreSQL tem um mecanismo de deteção de deadlocks que automaticamente aborta uma das transações permitindo que a outra prossiga (a escolha de qual vai ser abortada é difícil de prever).

5.3 Isolamento de Transações

O standard do SQL define quatro níveis de isolamento das transações (serializable, Read uncommitted, read committed, repeatable read) de forma a garantir que os seguintes fenómenos não aconteçam entre diferentes transações:

- Dirty read - Uma transação lê dados escritos por uma transação concorrente que ainda não fez o commit
- Nonreapeatable read – Uma transação relê os dados que anteriormente tinha lido e verifica que foram modificados por uma outra transação;
- Phantom read – Uma transação executa novamente uma consulta anterior- mente executada e verifica que o conjunto de tuplos inicialmente obtidos é diferente devido a uma mordicação feita por outra transação;

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Figura 5.5 – Tabela com o comportamento de cada fenómeno em cada nível de isolamento

Ao nível interno do PostgreSQL só existem três níveis distintos, Read Committed, Repeatable Read e Serializable e o fenómeno Phantom Read não acontece na implementação do Reapetable Read por parte do PostegreSQL.
Para se ver o nível de isolamento de uma transação, usa-se o comando SET TRANSACTION.

5.3.1 Read Commited

Este é o nível de isolamento utilizado por omissão no PostegreSQL. Neste nível de isolamento, as queries SELECT apenas vêm os dados que foram committed antes do inicio da query (vê um Snapshot), e alterações feitas pela própria transação. Os comandos UPDATE, DELETE, SELECT FOR UPDATE e SELECT FOR SHARE têm um comportamento semelhante, no entanto, o tuplo alvo já pode ter sido atualizado.

5.3.2 Repeatable Read

Neste nível de isolamento, as queries apenas vêm os dados que foram committed antes do inicio da transação (vê um Snapshot), e alterações feitas pela própria transação. A diferença deste para o Read Commited, é que este vê snapshots criados antes do inicio da transação, enquanto o Read Commited vê snapshots criados antes do inicio de cada query, ou seja, queries sucessivas de SELECT (Repeatable Read) obtêm sempre o mesmo resultado, não vêm alterações commited depois da transação começar. Aplicações que usem este nível de isolamento têm que estar preparadas para repetir transações devido as falhas de serialização.

5.3.3 Serializable

Este é considerado o nível de isolamento mais estrito. Este nível simula execução serial de todas as transações commited, como se as transações tivessem sido executadas umas depois das outras em vez de forma concorrente. Como no Repeatable Read, alicações que usem este nível de isolamento têm que estar preparadas para repetir transações devido as falhas de serialização. De facto, este nível de isolamento funciona da mesma forma que o Repeatable Read, exceto que monitoriza condições que possam fazer execuções concorrentes comportar-se de forma inconsistente. No entanto esta monotorização não introduz blocks extras em relação ao Repeatable Read.

5.4 Locking e índices

Apesar de PostgreSQL fornecer acesso de leitura/escrita nonblocking a dados da tabela, o acesso de leitura/escrita não é oferecido atualmente para cada método de acesso a índices implementado em PostgreSQL. Os vários tipos de índice são tratados como se segue:

- B-tree, GiST and SP-GiST indexes - Short-term share/exclusive page-level locks são usados em acessos de leitura/escrita. Os locks são libertados imediatamente após cada linha do índice ser fetched ou inserted. Este tipo de índice é o que oferece maior capacidade de concorrência sem deadlocks.
- Hash indexes - Share/exclusive hash-bucket-level locks são usados em acessos de leitura/escrita. Os locks são libertados após o processamento de um bucket. Estes índices oferecem uma melhor capacidade de concorrência, no entanto, os deadlocks são possíveis uma vez que os locks são mantidos durante mais tempo que os anteriores.
- GIN indexes - Short-term share/exclusive page-level locks são usados em acessos de leitura/escrita. Os locks são libertados imediatamente após cada linha do índice ser fetched ou inserted. A inserção de um GIN-indexed value causa que sejam inseridos vários índex key por linha, por isso, GIN tem mais trabalho em cada inserção.

Actualmente, os índices B-tree oferecem a melhor performance para aplicações concorrentes, uma vez que têm mais features que os índices por hash. Os B-tree são os tipos de índices recomendados para aplicações concorrentes que precisam de indexar data escalável. Quando se lida com data não escalável, os B-tree não são os mais uteis, e por isso usam-se os índices GiST, SP-GiST ou GIN.

5.5 Write Ahead Log

O PostgreSQL possuí um método para assegurar a integridade dos dados, o Write Ahead-Log. Com o WAL não é permitido que sejam feitas alterações em disco sem antes serem inseridas num log. Assim, mesmo que ocorra um crash na base de dados antes de todas as alterações terem sido aplicadas, pode-se refazer as alterações através da leitura do log.

5.6 Comparação com o Oracle 11g

O PostgreSQL é semelhante ao Oracle em relação aos tipos de isolamento uma vez que implementa o Read Committed, Repeatable Read e Serializable, apesar de internamente não suportar Read Uncommitted. Ambos utilizam mecanismos de Snapshot Isolation, permitem declarar savepoints e possuem a possibilidade de obter explicitamente locks. Para além disto o PostgreSQL oferece ainda advisory locks.

6 Suporte para bases de dados distribuídas

O PostgreSQL disponibiliza um mecanismo de replicação para suportar alta disponibilidade, balanceamento de carga e replicação de dados. A replicação é baseada no uso de WAL (Write-Ahed Logs) e o sistema, baseado num servidor primário e num ou mais servidores secundários, garante que a qualquer momento, um servidor secundário pode assumir o papel de primário, caso haja uma falha. Caso isto aconteça, o novo servidor principal tem que executar os procedimentos de failover.

6.1 Log-Shipping Standby Servers

Log-Shipping é uma técnica usada para que o servidor secundário seja atualizado com as alterações do servidor principal, através da transferência dos registos do WAL. Os ficheiros do WAL são transferidos um de cada vez em segmentos de 16MB, o que torna a sua transferência na rede fácil e sem grandes custos, independentemente da distancia entre os servidores. A largura de rede necessária depende da transaction rate do servidor principal. A transferência do WAL para os servidores secundários é efetuada de forma assíncrona (WAL é enviado depois da transação ser commited) o que cria uma janela temporal na qual se podem perder dados, no caso de haver uma falha no servidor antes das transações serem concluídas.

6.2 Replicação por Stream

Este tipo de replicação permite que as réplicas estejam o mais atualizadas possível uma vez que o servidor secundário abre uma conexão com o servidor principal para receber as informações por stream, assim que são geradas. Desta forma, não tem que esperar que o ficheiro WAL seja preenchido e por isso diminui o delay entre os servidores.

6.3 Replicação em cascata

Esta funcionalidade permite que os servidores secundários, ao receberem as atualizações, propaguem essa informação para outros servidores de forma a reduzir a carga do servidor principal.

6.4 Replicação síncrona

No PostgreSQL, a replicação por defeito é feita de forma assíncrona e por isso, em caso de falha do servidor principal, pode haver transações que embora tenham terminado com sucesso, acabem por não ser replicadas, o que leva à perda de dados. Com replicação síncrona é garantido que se a transação teve sucesso, então pelo menos uma réplica recebeu a atualização (o sucesso da transação depende se esta foi escrita e guardada nos logs do servidor principal e de um dos servidores secundários). Como seria de esperar, estas transações têm um acréscimo no tempo de resposta, uma vez que tem que se esperar que o servidor secundário execute as operações. A única forma de haver perda de dados é o servidor principal e secundário falharem ao mesmo tempo. É possível definir que transações é que usam replicação síncrona de forma a evitar perda de eficiência.

6.5 Comparação com o Oracle 11g

O suporte de distribuição no PostgreSQL é bastante inferior ao disponível no Oracle. Tal como o Oracle, o PostgreSQL suporta replicação de dados mas apenas numa arquitetura de servidor primário/secundário. No entanto, ao contrario do Oracle, não implementa fragmentação de dados.

7 Outras características do sistema estudado

7.1 Procedural Languages

O *PostegreSQL* permite que sejam criadas outras funções sem ser em SQL ou C. Estas outras linguagens são chamadas de *Procedural Languages* (PLs). Para as funções escritas em linguagem PL, a sua interção é feita através de um *handler* específico da linguagem (servidor não tem mecanismos para interpretar a função internamente). Este *handler* é uma função na linguagem C. As *procedural languages* disponiveis pelo standard *PostegreSQL* são o PL/pgSQL, PL/Tcl, PL/Perl e PL/phyton, no entanto podem ser instaladas outras linguagens pelo utilizador.

PL/pgSQL

- Pode ser usado para criar funções e *triggers*
- Adiciona estruturas de controlo à linguagem SQL
- Pode executar computações complexas
- Herda todos os *user-defined types*, funções e operadores

PL/Tcl

Oferece a maioria das funcionalidades que C oferece, tem-se acesso as bibliotecas de processamento de texto do Tcl e todo o código da função corre no ambiente protegido do interpretador de Tcl.

PL/Perl

Oferece a possibilidade de escrever funções e *triggers* em *perl*. Ao podermos usar o *Perl* temos acesso as expresões regulares e grande biblioteca que existe. Fazer parse de sintrs complexas pode ser mais facil com *Perl* do que as string functions e estruturas de controlo do PL/pgSQL.

PL/Python

Oferece a possibilidade de escrever funções em *Python* o que significa que temos acesso a uma linguagem orientada a objectos que contem uma biblioteca bastante completa.

7.2 Suporte para XML

O *PostgreSQL* suporta o tipo de dados XML e tem varias funções que permitem a sua manipulação, desde a existência de um tipo de dados XML com um conjunto de primitivas de manipulação, executar consultas a elementos XML, mapeamento de esquemas de tabelas, entre outras funcionalidades.

Para criar um valor do tipo XML a partir de uma *string*, é usado o seguinte comando:

```
XMLPARSE ( { DOCUMENT | CONTENT } value)
```

Para realizar o inverso e criar uma *string* desde um documento XML, é utilizado o seguinte comando:

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } value AS

{ character | character varying | text } )
```

O *PostgreSQL* tem suporte para processar XML usando a interrogação XPath, para tal temos a seguinte função:

```
xpath(xpath, xml [, nsarray])
```

7.3 Segurança e Autenticação

O sistema *PostgreSQl* permite a possibilidade de criar uma ou varias bases de dados em um servidor, composta por uma lista de utilizadores que provavelmente terão diferentes direitos de acesso. São oferecidos vários métodos de autenticação, nos quais podemos listar:

- *Trust*
- *Password*
- *GSSAPI*
- *SSPI*
- *Kerberos*
- *Ident-based*
- *LDAP*
- *RADIUS*
- *Certificate*
- *PAM*

A existência deste número elevado de formas de autenticação esta fortemente relacionada com o facto de o sistema ser de código aberto. A configuração da autenticação dos clientes é controlada por um ficheiro de configuração chamado *pg_hba.conf.* Neste ficheiro é possível especificar-se o tipo de ligação (local ou remota), base de dados e utilizador a conectar-se e qual o método de autenticação.

8 Referências

[1] http://www.postgresql.org/files/documentation/pdf/9.3/postgresql-9.3-A4.pdf

[2] http://www.postgresql.org/docs/9.3/static/