departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Sistemas de Bases de Dados 2013-14

Estudo do Sistema de Gestão de Bases de Dados PostgreSQL

Trabalho realizado por:

Albert Linde, nº 42180

Marta Lidon, nº41996

Ricardo Monteiro, nº 41810

G04

Professor: José Alferes

Índice

1. Introdução

No contexto da cadeira de Sistema de Base de Dados foi-nos proposto que estudássemos um sistema de gestão bases de dados relacional. Após alguma pesquisa sobre os sistemas existentes, a documentação que possuem e a utilização do sistema em si, acabámos por escolher o sistema de base de dados PostgreSQL, versão 9.3.

PostgreSQL é um sistema de gestão de bases de dados objecto-relacional *open source* com ênfase na extensibilidade e conformidade com os *standards*. É baseado na versão 4.2 do POSTGRES desenvolvido na Universidade de Berkeley. O POSTGRES foi um sistema pioneiro em vários conceitos que apenas apareceram em alguns sistemas de bases de dados comercias muito mais tarde.

Um dos ancestrais do PostgreSQL foi o projeto Ingres que foi lançado na Universidade de Berkeley, nos Estados Unidos da América em 1977. Depois desse projecto, Michael Stonebraker iniciou o projecto POSTGRES (Post-Ingres) com o objectivo de implementar novas funcionalidades, como possibilidade de definir tipos de dados e relações. O projecto terminou após quatro versões. Mais tarde, Andrew Yu e Jolly Chen, dois estudantes de Berkeley, substituíram a linguagem de interrogação QUEL do POSTGRES para uma linguagem SQL, criando o Postgres95. Em 1996, o sistema foi renomeado para PostGreSQL. Em 1997 foi lançada a versão 6.0 e desde essa altura que o sistema é desenvolvido pela comunidade *open source*.

O PostgreSQL é um sistema de gestão bases de dados fiável, que implementa quase todas as funcionalidades existentes, corre em vários sistemas operativos e é de fácil utilização. Por isso, é bastante usado a nível pessoal e também a nível empresarial.

Durante este relatório iremos abordar vários aspectos do PostgreSQL como o armazenamento e *file structure*, indexação e hashing, processamento e optimização de perguntas, gestão de transações e controlo de concorrência, suporte para bases de dados distribuídas e outras características como XML, *triggers* e tipo de dados. Durante os diversos capítulos iremos fazendo a comparação com o SGBD estudado durante as aulas, Oracle.

2. Armazenamento e file structure

2.1. Database File Layout

O PostgreSQL armazena os dados em ficheiros, no sistema de ficheiros geridos pelo sistema operativo. Neste sentido, todas as coleções de bases de dados são guardadas numa área em disco, numa directoria designada por PGDATA. Esta directoria contém diversas subdirectorias assim como ficheiros de controlo e configuração do *cluster* (ver tabela 1). Para cada base de dados no *cluster* há uma sub-diretoria dentro PGDATA/base, cujo nome é um identificador único. A lista de correspondências de bases de dados/identificador pode ser consultada no ficheiro pg_database.

Em comparação com o PostgreSQL, o Oracle não conta com o sistema de ficheiros do sistema operativo subjacente.

Tabela 1- Conteúdo da diretoria PGDATA

Item	Descrição
`PG_VERSION`	Ficheiro que contém a versão do sistema.
`base`	Sub-directoria que contém várias directorias, uma por cada base de dados.
`global`	Sub-directoria que contém tabelas transversais do *cluster*.
`pg_clog`	Sub-directoria que contém o estado das transações *commited*.
`pg_multixact`	Sub-directoria que contém o estado de multitransacções.
`pg_notify`	Sub-directoria que contém os estados dos dados *LISTEN/NOTIFY*.
`pg_serial`	Sub-directoria que contém informação sobre as transações *commited serializable*.
`pg_snapshots`	Sub-directoria que contém os *snapshots* exportados.
`pg_stat_temp`	Sub-directoria que contém ficheiros temporários para o subsistema estatístico.

`pg_subtrans`	Sub-directoria que contém o estado dos dados das subtransacções.
`pg_tblspc`	Sub-directoria que contém ligações para *tablespaces*.
`pg_twophase`	Sub-directoria que contém o estado ficheiros para transações preparada. (*Two phase commited*).
`pg_xlog`	Sub-directoria que contém ficheiros WAL(*Write Ahead Log*).
`postmaster.opts`	Ficheiro que contém o registo dos argumentos passados na linha de comandos quando o servidor foi lançado da última vez.
`postmaster.pid`	Ficheiro que contém o identificador do processo do *postmaster* corrente.

Cada tabela e índice são guardados num ficheiro em separado. Associado a cada tabela existe o *free space map* e *visibility map*. O ficheiro *free space map* é responsável por manter informação sobre o espaço disponível na tabela e o *visibility map* por manter a informação sobre as páginas que têm os tuplos visíveis a todas transacções activas.

Quando uma tabela (ou índice) ultrapassa 1 GB de tamanho, então é dividida em segmentos de 1 GB, com o nome nome_ficheiro.1, nome_ficheiro.2, nome_ficheiro.3, e assim sucessivamente. Por outro lado, também os *free space maps* e os *visibility maps* podem ser segmentados. Neste sentido, o PostgreSQL faz uso do sistema de ficheiros do próprio sistema operativo.

Para além disso, uma tabela, onde existam colunas com entradas de grandes dimensões, terá uma tabela TOAST associada, na qual serão guardados estes valores.

2.2. The Oversized-Attribute Storage Technique (TOAST)

O PostgreSQL usa páginas de tamanho fixo (geralmente 8 kB) e não permite que tuplos estejam em duas páginas diferentes. Por esta razão, não é possível armazenar atributos de grandes dimensões directamente. Para ultrapassar esta limitação, o TOAST comprime e/ou divide um tuplo em múltiplas linhas. Apenas alguns tipos de dados suportam TOAST, uma vez que não há necessidade de haver uma tabela destas para tipos de dados que não produzam valores de grande dimensão. Assim, o tipo de dados para suportar TOAST terá que ter uma representação de tamanho variável.

Esta técnica é usada quando um tuplo é superior a ¼ do tamanho do bloco. O tamanho da linha é reduzido, por compressão dos campos. Se esta compressão for ainda insuficiente, os dados são armazenados numa outra tabela, sendo guardada uma referência para a tabela original. Fazem-se estas técnicas iterativamente, até se obter o resultado desejado.

2.3. Organização da informação

Cada tabela e índice é armazenado como um *array* de páginas (ver tabela 2) de tamanho fixo (normalmente 8 kB). Estas são organizadas num ficheiro de *heap*, que utiliza *slotted-pages* (Figura 1). Numa tabela, todas as páginas são logicamente equivalentes, portanto um dado tuplo pode ser armazenado em qualquer página. Por outro lado, nos índices, a primeira página é por norma reservada como *metapage* onde está a informação de controlo. Para além disso, pode haver diferentes tipos de página dentro de um índice, dependendo do método de acesso ao índice.

O PostgreSQL não suporta *multitable clustering*, contrariamente ao Oracle.

Tabela 2- Esquema geral da página

Item	Descrição
`PageHeaderData`	Contém informação geral sobre a página, incluindo os apontadores dos espaços livres.
`ItemIdData`	*Array* de *pares* (*offset*, *length*) que aponta para os items actuais (4 bytes por cada item).
`Free space`	O espaço não utilizado.
`Items`	Os items actuais.
`Special Space`	Dados específicos do método de acesso do índice.

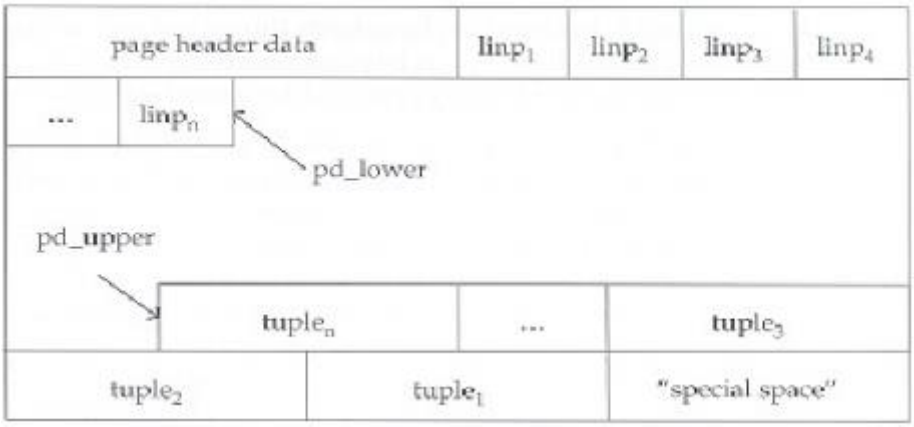

Figura 1 - Slotted-page

2.4. Buffer management

Como referido o PostgreSQL utiliza o sistema de ficheiros oferecido pelo sistema operativo. O sistema operativo já tem um *buffer manager*, que tem como função controlar a quantidade e a forma de como a informação está armazenada. O *buffer* manager do PostgreSQL é responsável por decidir que páginas vai trazer para a memória e que páginas serão libertadas através da política de LRU (*Least Recently Used*). Assim, o PostgreSQL usa *buffers* que acabam por servir como cache uma vez que os usa para fazer a transição dos dados entre o disco rígido e o *backend* do sistema.

O *buffer manager* do PostgreSQL é responsável por manter em cache os blocos mais usados recentemente e actualizar dinamicamente os blocos mais frequentemente usados. Quando é necessário guardar um bloco, que não se encontra num *buffer*, usa-se o algoritmo *clock sweep*. Este algoritmo vê a cache como uma lista circular, percorrendo-a à procura à procura de blocos disponíveis, para que guardar o bloco. Um bloco está disponível quando não está a ser referenciado por um processo. Em comparação com o PostgreSQL, o Oracle tem o seu próprio *buffer management*, com politicas de privacidade complexas.

2.5. Mecanismo de Partições

O particionamento é a capacidade de partir uma tabela de grandes dimensões em várias de menor tamanho que a original. Tanto o PostgreSQL como o Oracle suportam particionamento. No PostgreSQL para se criar uma partição deve-se criar uma tabela que é descendente de uma outra (que está usualmente vazia) existindo com o intuito de representar o conjunto de informação guardado nas tabelas que dela são descendentes. Para além disso, o utilizador deve criar índices respectivos a cada uma das subtabelas e *triggers* para que instruções como inserir tuplos, estes sejam inseridos na respectiva partição. Abaixo encontram-se os comandos de criação da tabela principal, criação de uma tabela para cada partição e criação dos respectivos índices e *triggers*.

1. Criar a tabela principal.

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

2. Criar uma tabela para cada partição. Cada tabela tem restrições para que não existam tuplos repetidos nas várias tabelas.

```
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);
```

3. Criar índices para o atributo de partição.

```
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);
```

4. Criar *trigger* para a inserção nas várias tabelas.

```
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;
```

3. Indexação e hashing

Os índices de uma base de dados servem para diminuir o tempo de pesquisa de um determinado tuplo. No entanto, os índices nem sempre melhoram o desempenho. Por vezes, o *overhead* criado pela utilização dos índices não compensa e é melhor fazer uma pesquisa completa na tabela. Por isso, a utilização dos índices tem de ser feita com algum cuidado.

Numa base de dados sem índice para a tabela X, a pesquisa de um id nessa tabela teria que percorrer toda a tabela até encontrar esse id. Quando existe um índice nessa tabela para esse atributo, o sistema pode tirar partido desse índice para obter os tuplos desejados.

Quando um índice é criado, o sistema da base de dados encarrega-se de manter esse índice actualizado. O sistema usa o índice quando julga que é melhor que percorrer a tabela sequencialmente. Para o sistema conseguir determinar a melhor opção, deve ser utilizado regularmente o comando ANALYZE para actualizar estatísticas, desta forma o sistema poderá fazer melhores decisões.

A criação de índices em tabelas grande demora algum tempo. O PostgreSQL permite fazer leituras à tabela durante a criação do índice. Por omissão, as escritas na tabela ficam suspensas até que a criação do índice termine. No entanto, o PostgreSQL permite que existam escritas na tabela em paralelo com a criação do índice.

Após a criação do índice, existe o *overhead* de mantê-lo actualizado quando existem modificações na tabela.

3.1. Tipos de Índices

O PostgreSQL implementa vários tipos de índices, B-Tree, Hash, GiST, SP-GiST e GIN. Quando se cria um índice sem especificar qual o tipo do índice, é criado um B-Tree porque é um índice que é o melhor para a maioria dos casos.

Os índices B-Tree são estruturas de dados em forma de árvore n-ária onde os dados são mantidos ordenados. No PostgreSQL os índices B-Tree podem ser usados para obter tuplos onde são utilizados os seguintes operadores sobre uma coluna específica:

<
<=
=
>
>=

Também podem ser usados para obter tuplos de forma ordenada, apesar de nem sempre ser mais rápido que utilizar um *full-scan* e ordenar. Este índice implementa o algoritmo concorrente de Lehman-Yao.

Índices Hash apenas podem ser usados para comparações de equidade e não suporta pesquisas com a condição IS NULL. O uso de índices Hash é ponderado pelo *query planner* quando existe uma comparação de igualdade para uma coluna indexada. O PostgreSQL implementa o algoritmo *Linear Hashing* de Litwin. No PostgreSQL existe um problema com este tipo de índices, após um *crash* da base de dados deve ser usado o comando `REINDEX` se houve modificações que não foram escritas. Desta forma, o uso deste índice é desaconselhado.

Índices GiST (*Generalized Search Tree*) não são um tipo de índice concreto, são uma infraestrutura onde podem ser implementadas várias estratégias de indexação. Os operadores possíveis nos índices GiST dependem da estratégia de indexação (*operator class*). A distribuição *standard* do PostgreSQL contém *operator classes* para vários tipos de dados geométricos bidimensionais.

SP-GiST são, assim como os GiST, uma infraestrutura que suporta vários tipos de pesquisas. Permitem a implementação de vários tipos de estruturas de dados não balanceadas, como árvores *k-d* e *radix trees*.

GIN (*Generalized Inverted Index*) são índices especiais optimizados para itens complexos e *queries* que pretendem valores de elementos contidos nos itens. Por exemplo, os itens são documentos e as *queries* são pesquisas por palavras especificas nos documentos.

Os índices GiST, SP-GiST e GIN não estão disponíveis no Oracle. O Oracle tem índices *bitmap* que não podem ser criados em PostgreSQL.

3.2. Índices com múltiplos atributos

Os índices do tipo B-Tree e GiST podem ser criados para mais que uma coluna de uma tabela (múltiplos atributos), até um máximo de 32 colunas.

Um índice com múltiplos atributos pode ser utilizado para uma *query* onde esteja envolvido um subconjunto dos atributos contidos no índice. A utilização destes índices é mais eficiente quando as restrições existentes na *query* são para os atributos mais à esquerda do índice. Com uma restrição de igualdade nos atributos mais à esquerda e outra restrição (sem ser de igualdade) no atributo seguinte aos que foram restringidos por igualdades, diminui-se o range do índice que é percorrido. Restrições em atributos mais à direita aos anteriormente apresentados apenas reduzem acessos à tabela, as pesquisas no índice mantêm-se com o mesmo range.

Este tipo de índices deve ser usado apenas quando mesmo necessários, porque na maioria dos casos um índice com um único atributo é suficiente para obter melhores performances.

3.3. Múltiplos índices

Quando uma *query* contém uma conjunção ou disjunção de atributos da mesma tabela o PostgreSQL consegue combinar vários índices existentes e/ou fazer várias pesquisas no mesmo índice. Por exemplo uma *query* com a condição `WHERE x=42 OR x=47 OR x=53 OR x=99` pode ser dividida em quatro pesquisas separados no índice existente para o atributo `x`. Outro exemplo é para *queries* como `WHERE x = 5 AND y = 6`, se existir um índice para o atributo `x` e outro para o atributo `y`, uma possível solução é usar os índices existentes para cada atributo e no final fazer a intersecção dos dois conjuntos.

Para combinar múltiplos índices, o sistema percorre todos os índices necessários e gera um *bitmap* em memória contendo a localização dos tuplos na tabela que verificam a condição da *query*. Em seguida, aplica a conjunção ou disjunção, consoante a *query*, e vai buscar os tuplos à tabela.

O PostgreSQL também permite gerar vários índices para o mesmo conjunto de atributos.

3.4. Índices parciais

No PostgreSQL é possível criar um índice parcial. Este índice contêm apontadores apenas para uma parte da tabela. Por exemplo, uma tabela com encomendas facturadas e não facturadas, onde estas são uma pequena parte da tabela, é possível criar um índice apenas das encomendas não facturadas. Se a maioria dos acessos à tabela forem apenas para as encomendas não facturadas, não há a necessidade de incluir as encomendas facturadas no índice. Desta forma melhora-se a performance aos acessos das encomendas não facturadas.

3.5. Organização de ficheiros

A organização das tabelas em PostgreSQL não é *clustered*, isto é, os tuplos das tabelas não estão fisicamente ordenados de acordo com os índices. No entanto, existe o comando `CLUSTER` capaz de ordenar fisicamente uma tabela utilizando um índice existente. Esta operação é única e não mantém a tabela ordenada, se houver alterações a tabela não se mantém fisicamente ordenada.

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

CLUSTER [VERBOSE]
```

Este comando ordena fisicamente a tabela ***table_name*** de acordo com o índice ***index_name***. Depois de utilizar este comando numa tabela, o PostgreSQL lembra-se que índice foi usado para cada tabela e nas próximas vezes basta dar o nome da tabela neste comando. Usando apenas o comando `CLUSTER` sem nome dar nome de tabelas, o PostgreSQL ordena todas as tabelas que anteriormente foram ordenadas.

No Oracle existem tabelas designadas de *Index-Organized Tables* que mantêm os tuplos ordenados fisicamente. O comando para a ordenação das tabelas pode ser dado na criação das tabelas ou depois de criadas pode-se alterar a ordenação da tabela.

3.6. Comando CREATE INDEX

```
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 ]
```

`CREATE INDEX` constrói um índice para a(s) coluna(s) específica(s) da relação de uma tabela ou *materialized view*. Em seguida são explicados os parâmetros deste comando.

`UNIQUE`

Quando o índice é criado o sistema verifica os valores duplicados na tabela já existentes e quando são adicionados novos valores. Tentativas de inserção ou actualização em dados que resulta em duplicados geram um erro. Este tipo de índices podem ser usados para reforçar a unicidade de valores numa colunas ou de combinações de valores de várias colunas.

`CONCURRENTLY`

Com esta opção, o índice é criado sem fazer *locks* na tabela, assim é possível outras transacções fazerem escritas na tabela. Sem esta opção a criação do índice apenas permite leituras à tabela. Existem algumas advertências ao utilizar esta opção.

name

O nome do índice a ser criar. Se o nome estiver omitido, o PostgreSQL escolhe um nome adequado baseando-se nos nomes da tabela e da coluna.

table_name

O nome da tabela a ser indexada.

method

O tipo de índice a criar, as várias opções são: `btree`, `hash`, `gist`, `spgist` e `gin`. O tipo por omissão é `btree`.

column_name

O nome da coluna da tabela.

expression

Uma expressão baseada em uma ou mais colunas da tabela. A expressão deve ser escrita entre parenteses.

opclass

O nome do *operator class*, pode ser um dos seguintes:

`ASC`

Especifica ordenação ascendente. Esta é a opção por omissão.

`DESC`

Especifica ordenação descendente.

`NULLS FIRST`

Os valores nulos antes dos não nulos. Quando escolhida a opção DESC, esta é a opção por omissão.

`NULLS LAST`

Valores nulos após valores não nulos. Esta é a opção por omissão quando DESC não é escolhida.

predicate

A restrição para criar um índice parcial.

4. Processamento e optimização de perguntas

O processamento de perguntas consiste nas várias actividades envolvidas na obtenção de dados da base de dados. Estas actividades incluem a tradução das *queries* na linguagem de alto nível, neste caso SQL, para expressões que podem ser usadas pelo nível físico do sistema de ficheiros, transformações para optimização de *queries* e avaliação de *queries*.

A optimização de *queries* é o processo de escolher o plano de execução da *query* mais eficiente de entre todas as possíveis formas de execução da *query*, especialmente se a *query* for complexa.

No PostgreSQL, uma *query* tem de passar por cinco fases até a obtenção de resultados.

1. Uma aplicação tem de estabelecer uma conecção com o servidor PostgreSQL. A aplicação envia a *query* para o servidor e espera pelos resultados.
2. A fase de *parser* (*parser stage*) verifica se a sintaxe da *query* está correcta e cria uma *query* tree.
3. A fase de reescrita analisa o plano de execução e verifica se existe alguma regra a aplicar à *query tree*.
4. Uma modificação realizada pelo sistema de reescrita é para as *queries* que acedem a *views*. A modificação consiste em alterar a *query* de modo a aceder à tabela respectiva da *view*.
5. O optimizador de *queries* transforma a *query tree* num plano de execução.

O plano a executar é o melhor plano de entre vários planos gerados. O optimizador cria todas as combinações de planos possíveis para a *query tree*. Em seguida estima o custo das execuções de cada plano e escolhe o menor.

O executor processa recursivamente o plano dado pelo optimizador para extrair os tuplos, usando um mecanismo *demand-pull pipelining*.

4.1. Fase do parser

4.1.1. Parser

O *parser* recebe a *query* em formato *string* e faz uma validação de sintaxe. O *parser* está definido num ficheiro designado por *gram.y* e consiste num conjunto de regras gramaticais e acções. Se a sintaxe estiver correcta, cria uma *parse tree*, caso contrário retorna um erro.

O *lexer* é responsável por reconhecimento de identificadores, SQL *key words*, entre outros. Para cada *key word* ou identificador é gerado um *token* e devolvido ao *parser*.

Nesta fase, o *parser* cria uma *parse tree* usando regras sobre a estrutura sintática do SQL.

4.1.2. Processo de transformação

Depois da fase de *parsing*, o processo de transformação utiliza o resultado da última fase, a *parser tree*, e faz a interpretação da semântica necessária para compreender quais as tabelas, funções e operadores são referenciados na *query*. Este processo cria uma *query tree*.

Uma *query tree* consiste numa representação interna de *query* SQL em que cada elemento é guardado separadamente. As *query trees* podem ser vistas activando os parâmetros `debug_print_parse`, `debug_print_rewritten`, ou `debug_print_plan` no ficheiro de configuração do PostgreSQL ("postgresql.conf").

As *query trees* são constituídas pelos seguintes elementos.

- **Tipo de comando**

  Valor que indica o comando (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) produzido pela *query tree*.

- **Lista de relações**

  Lista de relações que são usadas na *query*. No comando `SELECT`, são as relações que estão depois da palavra `FROM`.

- **Relação resultante**

  Apontador para a tabela onde irão surgir modificações pelos comandos `INSERT`, `UPDATE` e `DELETE`. O comando `SELECT` não tem relação resultante.

- **Lista de alvos**

  Lista de expressões que definem o resultado de uma *query*. Para o comando `SELECT`, são as expressões entre as palavras `SELECT` e `FROM`. Se for utilizado o caracter '*', o *parser* substitui-o pelo nome de todas as colunas.

  O comando `DELETE` não produz nenhum resultado, então não tem lista de alvos.

  No comando `INSERT`, esta lista descreve os tuplos a serem inseridos na relação resultante.

  No comando `UPDATE`, esta lista descreve os tuplos novos que devem substituir os antigos.

- **Árvore de junções**

  Estrutura ordenada das junções da cláusula `FROM`. As restrições associadas a junções também são colocadas nesta árvore.

A Figura 2- Exemplo de Query treemostra uma *query tree* para a seguinte *query*:

```
SELECT name, surname FROM zion_users WHERE id_user > 1 ORDER BY id_user;
```

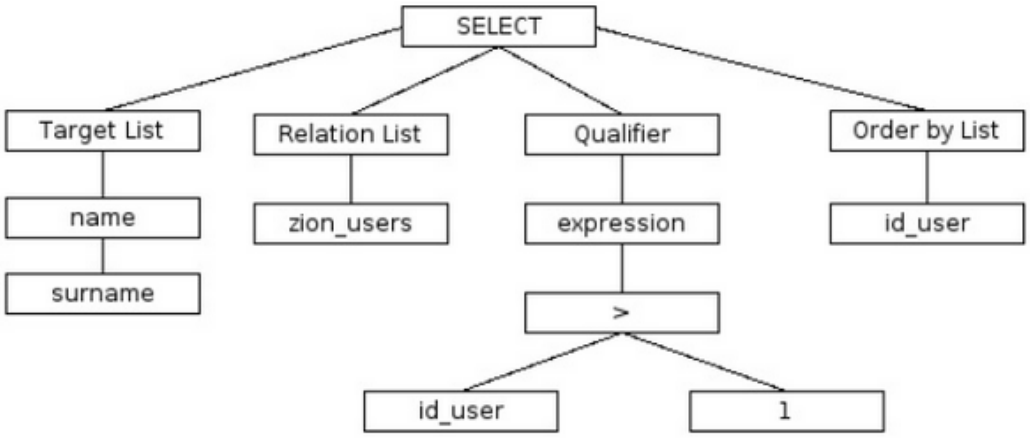

Figura 2- Exemplo de Query tree

O PostgreSQL não traduz o SQL para álgebra relacional como o Oracle. Em vez disso, usa as *queries trees*.

4.2. Fase de planeamento e optimização

Esta fase tem o objectivo de determinar o melhor plano de execução para as *queries*. É utilizado o resultado da fase anterior e são examinados todos os planos de execução possíveis para essa *query*. Após análise de todos os planos, o plano que é considerado o de menor custo é escolhido.

A análise de todos os planos de execução é feita apenas para junções até 12 relações (`geqo_threshold`, valor por omissão, no entanto pode-se alterar). Para junções com mais de 12 relações, o *overhead* criado por esta análise é tão grande que já não compensa escolher o melhor plano. Nesse caso, é utilizado um optimizador com funções genéticas para escolher um plano que pode não ser o melhor de todos os possíveis, mas é o melhor de entre os gerados.

4.2.1. Geração de planos possíveis

O optimizador começa por gerar planos que começam por percorrer relações individuais, usadas na *query*. Existe sempre a possibilidade de fazer um scan sequencial à tabela, então estes planos são sempre criados. Se existir um índice definido para a coluna da relação a restringir pela *query*, então também é gerado um plano que utiliza esse índice. Para além disso, são gerados planos para índices que tenham a ordenação pedida na *query*, ou uma ordenação que possa ser útil para um *merge join*.

Para *queries* que façam junções de duas ou mais relações, são criados todos os diferentes planos possíveis para junção das relações. Para *queries* com mais de duas junções, são criados todos os plano possíveis em forma de árvore de junções, onde em cada nó da árvore existe a junção de duas relações.

Depois de gerados todos os planos, é escolhido o de menor custo para executar. Estes custos são calculados com base em estatísticas. O plano final é constituído por scans, sequenciais ou utilizando índices, de cada relação, operações de junção e mais algumas operações necessárias, como por exemplo, ordenações e/ou agregações. Também podem selecionar apenas alguns tuplos de uma relação e fazer a projecção desses tuplos. Desta forma, reduz-se o tamanho das relações com o objectivo de tornar as junções mais rápidas. As selecções são feitas para as condições especificadas na cláusula `WHERE` da *query*.

O plano escolhido pelo optimizador é entregue ao executor.

4.3. Executor

Nesta fase, o executor recebe o plano criado pelo planificador/optimizador e processa-o recursivamente. Este processamento é basicamente um mecanismo de *demand-pull pipelining*. Sempre que um nó do plano é executado, deve devolver um tuplo ou informa que não há mais tuplos.

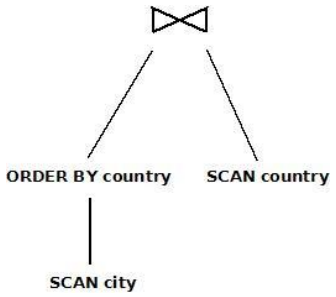

Figura 3- Plano de execução (não exacto)

Considerando o plano apresentado na figura 3, para fazer a junção de dois tuplos, é necessário obter esses tuplos de cada subplano. Para isso, o executor processa os subplanos recursivamente, começando com o subplano da esquerda. O nó seguinte do subplano esquerdo é uma ordenação, e novamente o executor processa o subplano seguinte. O subplano da ordenação é um scan sequencial, a execução deste nó faz o executor ler um tuplo da tabela e devolvê-lo para o nó de cima. O nó da ordenação chama o nó de scan até que seja devolvido o valor NULL, indicando que não existem mais tuplos para devolver. De seguida, é feita a ordenação e o primeiro tuplo é devolvido para o nó da junção. Quando o primeiro tuplo é recebido pelo nó da junção, este pede o tuplo do subplano da direita e aplica o algoritmo da junção.

4.4. EXPLAIN – plano de execução

O comando `EXPLAIN` mostra o plano de execução que o *planner* do PostgreSQL cria para o uma *query*. São mostrados os algoritmos de scan e junção, os atributos a serem projectados em cada tabela e os custos estimados de cada operação.

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

EXPLAIN [ ANALYZE ] [ VERBOSE ] statement



option pode ser:

    ANALYZE [ boolean ]

    VERBOSE [ boolean ]

    COSTS [ boolean ]

    BUFFERS [ boolean ]

    FORMAT { TEXT | XML | JSON | YAML }
```

`ANALYZE`

Esta cláusula faz com que o sistema execute o ***statement*** e mostre os custos reais. Por omissão, não é executado.

`VERBOSE`

Mostra a lista de *outputs* de cada nó do plano. Por omissão, não é executado.

`COSTS`

Informação das estimativas dos custos e custos totais para cada nó do plano de execução.

statement

Uma *query* de qualquer tipo, `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `VALUES`, `EXECUTE`, `DECLARE`, ou `CREATE TABLE AS`.

4.5. Algoritmos de selecção

4.5.1. Seq Scan

Este algoritmo consiste em percorrer toda a relação, linha a linha, e para cada tuplo testar a condição, caso exista. Se for uma pesquisa pro chave primária, o scan termina assim que

encontra o tuplo correspondente. Apesar de na maioria dos casos este não ser o melhor algoritmo, é sempre gerado um *query plan* com este algoritmo.

4.5.2. Index Scan

O *index scan* consiste em fazer uma pesquisa na relação utilizando um índice, retornando os tuplos pela ordem do índice. Este algoritmo pode ter restrições de início e/ou fim, deste modo não precisa percorrer a relação toda. A desvantagem em relação a este algoritmo é ter de realizar mais *seeks* que o *Seq Scan*, porque a tabela não está ordenada de acordo com o índice.

4.5.3. Bitmap Scan

Este algoritmo utiliza múltiplos índices de uma relação para determinar os tuplos que satisfazem a condição da *query* e criar um índice *bitmap* (tópico 3.3.). Os acessos aos tuplos são feitos pela ordem a que estão guardados fisicamente no disco, desta forma perde-se a ordenação dos índices.

4.6. Algoritmos de junção

O PostgreSQL implementa e utiliza os algoritmos de junção *nested loop-join*, merge-join e *hash join*. O utilizador pode controlar quais os algoritmos de junção utilizados através das seguintes funções.

```
enable_nestloop (boolean)

enable_mergejoin (boolean)

enable_hashjoin (boolean)
```

4.6.1. Nested Loop Join

Este é o algoritmo mais simples de junção, no entanto também é muito lento. Para cada tuplo da relação da esquerda, a relação da direita é percorrida e testa todos os tuplos com o tuplo da relação da esquerda. Se alguma relação couber toda em memória, então essa deve ser a relação da direita, caso contrário a relação menor deve ser a da esquerda.

Se existir um índice na relação da direita possível de utilizar, pode-se melhorar bastante o desempenho. O valor do tuplo da esquerda serve como chave para pesquisar no índice da relação da direita, evitando a varredura nessa relação.

4.6.2. Merge Join

Cada relação é ordenada pelos atributos a fazer a junção. Em seguida, as duas relações são percorridas em paralelo e os tuplos correspondentes são combinados. Podem ser utilizados, caso existam, índices dos atributos a fazer a junção, assim poupa-se o custo da ordenação. Este método de junção é melhor que o anterior, *nested loop join*, porque cada relação é percorrida apenas uma vez.

4.6.3. Hash Join

A relação da direita é percorrida e colocada numa tabela de dispersão, usando como chave o atributo a fazer junção. Em seguida, a tabela da esquerda é percorrida e os atributos de junção são utilizados como chave para localizar os tuplos correspondentes de junção.

4.7. Algoritmo de Ordenação

A ordenação de pequenas quantidades de dados é feita em memória. Quando não é possível fazer a ordenação em memória são utilizados ficheiros temporários e um algoritmo *external sort* normalizado.

O algoritmo *external sort* tem duas fases. Na primeira fase, são criadas várias runs ordenadas. Nesta fase é aplicado o seguinte algoritmo.

```
i = 0;
repeat
    read M blocks of the relation, or the rest of the relation,
        whichever is smaller;
    sort the in-memory part of the relation;
    write the sorted data to run file R_i;
    i = i + 1;
until the end of the relation
```

Na segunda fase, as várias *runs* são fundidas aplicando o algoritmo seguinte.

```
read one block of each of the N files R_i into a buffer block in memory;
repeat
    choose the first tuple (in sort order) among all buffer blocks;
    write the tuple to the output, and delete it from the buffer block;
    if the buffer block of any run R_i is empty and not end-of-file(R_i)
        then read the next block of R_i into the buffer block;
until all input buffer blocks are empty
```

4.8. Avaliação de expressões

Para avaliação de expressões complexas os PostgreSQL implementa *pipelining* e materialização.

O *pipelining* consiste em devolver os resultados para a operação de cima à medida que vão sendo produzidos.

A materialização avalia uma operação de cada vez, começando pelos níveis mais baixos. Os resultados intermédios são guardados temporariamente em disco para serem lidos pelas operações dos níveis superiores.

O PostgreSQL não implementa qualquer tipo paralelização para operações (*intraoperation parallelism* nem *interoperation parallelism*).

4.9. Views

No PostgreSQL as *views* são implementadas usando o sistema de regras. Para criar uma *view* utiliza-se o comando `CREATE VIEW`.

Por exemplo:

```
CREATE VIEW view_name AS SELECT * FROM table_name
```

Internamente, para criar esta *view* o sistema faz os seguintes comandos:

```
CREATE TABLE view_name

CREATE RULE "_RETURN" AS ON SELECT TO view_name DO INSTEAD

SELECT * FROM table_name;
```

Isto tem alguns efeitos secundários. A informação da *view* é exactamente a mesma que a da tabela e para o *parser* não há diferenças entre uma tabela e uma *view*.

4.9.1. Materialized Views

As *materialized views* utilizam o mesmo sistema que as *views*, mas preservam os resultados como se fossem uma tabela. Para criar uma *materialized view* utiliza-se o comando:

```
CREATE MATERIALIZED VIEW view_name AS SELECT * FROM table_name
```

Quando uma *query* referencia uma *materialized view*, os dados são devolvidos directamente da *materialized view*. No entanto, estes dados podem não estar actualizados. Para isso utiliza-se o comando:

```
REFRESH MATERIALIZED VIEW view_name
```

4.10. Estatísticas

O PostgreSQL cria um plano para cada *query* que recebe. Para um bom desempenho é essencial escolher o melhor plano de acordo com a estrutura da *query* e as propriedades dos dados. Para

puder escolher um bom plano, o sistema tem de estimar o número de linhas devolvidas por uma *query*. Com este objectivo, o sistema mantém guardados vários dados estatísticos sobre a base de dados.

Um componente estatístico é o número total de tuplos que existe em cada tabela e índice e o número de blocos que cada tabela e índice ocupam. Esta informação é guardada nas colunas `reltuples` e `relpages` da tabela `pg_class`. É possível fazer *queries* a esta tabela da seguinte forma:

```
SELECT relname, relkind, reltuples, relpages

FROM pg_class

WHERE relname LIKE relation_name;
```

As informações estatísticas não estão sempre correctas, para actualizar as estatísticas pode-se usar o comando `VACUUM`, `ANALYZE` e `CREATE INDEX`.

Como muitas das *queries* realizadas ás bases de dados não pretendem todos os tuplos de uma tabela, restringindo os dados pretendidos com a cláusula `WHERE`, o sistema tem de estimar quantos tuplos serão realmente utilizados. A informação utilizada para isso é guardada na tabela `pg_statistics`. As estatísticas desta tabela são actualizadas com os comandos `VACUUM` e `ANALYZE`. Mesmo quando actualizada, os valores guardados são sempre aproximações dos valores reais.

O PostgreSQL permite configurar a recolha de estatísticas no ficheiro “postgresql.conf”.

4.10.1. ANALYZE

Este comando colecciona estatísticas sobre a base de dados e guarda-as na tabela `pg_statistics`.

```
ANALYZE [VERBOSE] [ table_name [( column_name [, ...] )] ]
```

Sem parâmetros, este comando examina todas as tabelas; com o nome da tabela, examina apenas essa tabela. Também pode examinar apenas alguma(s) coluna(s) de uma tabela. O parâmetro `VERBOSE` faz com que o processo mostre mensagens de progresso, indicando que tabela está a ser processada e as suas estatísticas.

5. Gestão de transacções e controlo de concorrência

Este capítulo descreve como o PostgreSQL lida com transacções e a execução em paralelo delas. O uso de transacções e de locks é muito parecido ao seu uso no Oracle. Os dois sistemas funcionam com possibilidade de escolher entre vários modos de isolamento, deixando fazer para além disso lock explicito às tabelas e linhas das mesmas. As primitivas básicas de transacções são as seguintes:

```
BEGIN | BEGIN TRANSACTION
```

Começo de uma transacção.

```
SAVEPOINT savepoint_name;

--/--

ROLLBACK TO savepoint_name;
```

Uso de *savepoints*, fornece um melhor controlo de execução.

```
END TRANSACTION | COMMIT | ROLLBACK
```

Finalização, sendo esta *commit* ou *end transaction* (*save changes*) ou *rollback* (*undo changes*).

Os comandos de inicialização e finalização normalmente estão implícitos na execução de comandos, executando `BEGIN` antes do primeiro e `COMMIT` ao finalizar a sessão.

O PostgreSQL mantém internamente a consistência de dados usando um sistema multiversão (*Multiversion Concurrency Control*, MVCC). Isto significa que cada transacção vê uma cópia (*snapshot*) dos dados. Isto tem como efeito transacções concorrentes não verem um estado inconsistente, ou seja, existe *transaction isolation*. Assim minimiza a contenção criada por *locks* porque *locks* feitos para leitura não concorrem com *locks* de escrita e, assim, *locks* obtidos para escrita não esperam por *locks* de leitura e vice-versa.

Também existe suporte para *locks* a nível de tabela ou linha mas a documentação explicita que usar MVCC em geral tem um performance superior.

O SQL define três fenómenos que definem os vários níveis de isolamento:

Dirty Read - transacção lê dados escritos por uma transacção concorrente que não fez *commit*.

Nonrepeatable Read - transacção relê um campo de dados lido anteriormente e encontra dados modificados por outra transacção que fez *commit* entre as leituras.

Phantom Read - transacção reexecuta uma *query* com condições de pesquisa efectuada anteriormente e encontra o conjunto de dados mudado devido a uma transacção ter feito *commit*.

Os níveis de isolamento definidos no SQL são quatro: *Read Uncommitted*, *Read Committed*, *Repeatable R*ead e *Serializable*. No PostgreSQL podemos pedir as quarto mas só existem de facto três: *Read Committed*, *Repeatable Read* e *Serializable*. Os *reads committed* e *uncomitted* estão agrupados num só. As versões SQL destes modos definem o mínimo de consistência e os modos em PostgreSQL são ainda mais restritivos.

Assim, sobre os aspectos internos de isolamento do PostgreSQL, temos a seguinte tabela:

Tabela 3- Aspectos do isolamento

Nível isolamento	**Dirty Read**	**Nonrepeatable Read**	**Phantom Read**
Read (un)committed	Não é possível	Possível	Possível
Repeatable Read	Não é possível	Não é possível	Não é possível
Serializable	Não é possível	Não é possível	Não é possível

O nível de isolamento de uma transacção pode ser definido usando a instrução `SET TRANSACTION`:

```
SET TRANSACTION transaction_mode [, ...]
SET TRANSACTION SNAPSHOT snapshot_id
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]

where transaction_mode is one of:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

Para obter um ***snapshot_id*** pode-se usar, dentro de uma transacção, `SELECT pg_export_snapshot()`.

5.1. Read Commited

O nível de isolamento por defeito é *Read Committed*. Quando uma transacção usa este nível de isolamento cada *query* só observa dados que foram *committed* antes da *query* ser executada. Nunca encontra dados que não foram *committed* a menos os criados pela própria transacção. Outra característica é que duas *queries* iguais seguidas podem dar resultados diferentes se outra transacção entretanto fizer *commit*. Ou seja, quando uma *query* é executada, a transacção vê um *snapshot* da base de dados.

Ao fazer um `UPDATE`, `DELETE`, `SELECT FOR UPDATE`, ou `SELECT FOR SHARE`, estes actuam da maneira descrita anteriormente excepto se entretanto outra transacção actualizou a mesma linha. Neste caso fica à espera pela primeira transacção que faça *commit* ou *rollback*. Se o primeiro fizer *commit* as condições da *query* são confirmados no segundo e usa a versão mais recente da linha.

Este modo de isolamento é adequado para muita aplicações e é rápido e fácil de usar, mas pode não ser suficiente para todos os casos, assim temos os pontos seguintes.

5.2. Repeatable Read

Repeatable read vê somente dados *committed* antes da transacção ter começado, ou seja, nunca vê dados *uncommitted* ou mudanças efectuadas por transacções concorrentes. Isto difere do nível de isolamento anterior em que este vê um *snapshot* da base de dados antes de começar a transacção, enquanto *Read Commited* apanha um *snapshot* antes de cada *query*. Assim duas *queries* seguidas devolvem sempre os mesmos dados (dado que a própria transacção não actualizou os dados). Um problema aparece quando tentamos modificar dados que foram modificados por outra transacção onde temos de esperar pela execução da outra. Se a outra fizer *rollback* continuamos normalmente mas se esta fizer *commit* então temos que fazer *rollback* da nossa. O PostgreSQL neste caso devolve o seguinte erro:

```
ERROR:  could not serialize access due to concurrent update
```

Isto acontece, no nível de isolamento *repeatable read* impedindo transacções de modificar ou bloquear dados modificados por outras depois de começar. Quando este erro é encontrado a transacção têm de ser abortada e recomeçada do início. Desta segunda vez são encontradas as modificações da outra transacção. Assim chegamos à conclusão que somente transacções que fazem actualizações aos dados podem ser abortadas e transacções *read-only* nunca têm problemas de serialização.

5.3. Serializable

O nível de isolamento *serializable* é o nível mais restrito de isolamento. Este nível garante uma execução concorrente das transacções tal que elas parecem ter executado em série.

Este nível funciona tal como o *Repeatable Read* excepto que monitoriza condições em que transacções concorrentes poderiam correr de forma inconsistente com a serialização delas. Isto tem como efeito que as transacções podem abortar. Esta monitorização adicional tem um pequeno *overhead* e transacções que falham têm código de erro especial chamado *serialization failure*.

É importante somente considerar dados lidos como reais quando se fizer o *commit*, mesmo nas transacções *read-only*. Isto porque a meio da execução uma transacção ainda pode futuramente abortar. Os dados lidos numa transacção *read-only* com *deferrable* podem ser usados imediatamente, porque estas garantem a usar um *snapshot* livre dos problemas causados nos outros casos.

O sistema de *locks* usado nas transações são *predicate locking*, que não causam nenhum bloqueio por só verificarem dependências nas escritas. Se provocariam uma serialização incorrecta é abortada a transacção que primeiro encontra a violação. Ao contrário para garantir a consistência de dados no modo *Read Committed* ou *Repeatable Read* teríamos de fazer *lock* mesmo à tabela inteira.

O uso de transacções serializáveis pode simplificar o desenvolvimento de aplicações. A monitorização de dependências de leitura/escrita tem um custo associado assim como transacções que violem as regras de serialização têm de ser abortadas (com *rollback*) e reiniciadas. Mesmo assim em muitos casos é melhor usar serialização do que usar os outros modos e tentar garantir alguma forma de consistência com *locks* explícitos.

Para um desempenho melhor devem ser considerados as seguintes opções:

- Declarar transacções `READ ONLY` quando possível.
- Controlar o número de conecções activas. Isto pode ser especialmente importante num sistema disputado usando transacções *Serializable*.
- Não usar mais condições de integridade que necessárias.
- Eliminar *locks* explícitos assim que possível.
- Um scan sequencial numa tabela precisa sempre dum *predicate lock* ao nível da relação. Isto pode resultar num grande número de falhas de serialização. É boa prática encorajar o uso de *index scan* reduzindo *random_page_cost* ou incrementando *cpu_tuple_cost*.

Existe ainda a possibilidade de verificar a consistência só no final de uma transacção, isto pode ser feito com a seguinte instrução:

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

Esta instrução muda o comportamento de verificação das restrições de integridade na transacção corrente. Quando posto a `IMMEDIATE` as restrições são verificadas no fim de cada *query*, quando posto a `DEFERRED` só são verificados quando a transacção fizer *commit*.

Quando se cria uma restrição é dada uma das seguintes características: `DEFERRABLE INITIALLY DEFERRED`, `DEFERRABLE INITIALLY IMMEDIATE`, ou `NOT DEFERRABLE`.

A terceira tem sempre o modo `IMMEDIATE` e a instrução `SET CONSTRAINTS` não afecta restrições criadas deste modo. Os outros dois começam no modo indicado, mas o comportamento pode ser modificado.

5.4. Locks explícitos.

O PostgreSQL oferece vários modos de *locking* para controlo de concorrência a acesso a dados nas tabelas. Estes *locks* servem para dar controlo adicional onde o MVCC não fornece o comportamento necessário. A considerar é que muitos comandos PostgreSQL automaticamente adquirem *locks* nos modos certos para ter a certeza que tabelas não são removidas ou modificadas de maneira incompatível com o comando. Para examinar *locks* actualmente activos na base de dados pode-se usar a visualização do sistema aos *pg_locks* (`SELECT * FROM pg_locks`).

5.5. Locks a nível de tabela.

A lista apresentada de seguida apresenta um conjunto possível de modos de *lock* e o contexto em que são adquiridos automaticamente. Todos estes *locks* fazem *lock* a nível de tabela mesmo os com nome '*row*'. Modos de *lock* que não têm conflito podem ser obtidos concorrentemente, enquanto que os com conflito ficam bloqueados. Estes *locks* também podem ser adquiridos com o comando:

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

where lockmode is one of:

    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
    | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

`ACCESS SHARE`

Conflito com `ACCESS EXCLUSIVE`.

O comando `SELECT` adquire um *lock* deste modo nas tabelas referenciadas, em geral, qualquer *query* que somente faz *reads* numa tabela adquire este modo de *lock*.

`ROW SHARE`

Conflito com `EXCLUSIVE` e `ACCESS EXCLUSIVE`.

Comandos `SELECT FOR UPDATE` e `SELECT FOR SHARE` adquirem este *lock* nas tabelas referenciadas.

`ROW EXCLUSIVE`

Conflito com `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE`, e `ACCESS EXCLUSIVE`.

Comandos `UPDATE`, `DELETE` e `INSERT` adquirem este modo de *lock* na tabela seleccionada. Em geral este modo é obtido quando o comando modifica dados numa tabela.

`SHARE UPDATE EXCLUSIVE`

Conflito com `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE`, e `ACCESS EXCLUSIVE`.

Adquiridos por VACUUM (sem FULL), ANALYZE, CREATE INDEX CONCURRENTLY, e algumas formas de ALTER TABLE.

SHARE

Conflito com ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, e ACCESS EXCLUSIVE.

Este modo protege contra mudanças efectuadas concorrentemente. Adquiridos automaticamente por CREATE INDEX.

SHARE ROW EXCLUSIVE

Conflito com ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, e ACCESS EXCLUSIVE.

Não obtido automaticamente. Protege contra mudanças concorrentes aos dados, só uma sessão pode ter este *lock* num determinado momento.

EXCLUSIVE

Conflito com ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, e ACCESS EXCLUSIVE.

Este modo só aceita concorrência com ACCESS SHARE *lock*, ou seja, só leituras da tabela podem ser feitas em paralelo. Este modo nunca é obtido automaticamente.

ACCESS EXCLUSIVE

Conflito com todos os modos de *lock* (ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, e ACCESS EXCLUSIVE).

Garante que nenhuma outra transacção pode aceder à tabela de alguma maneira. Adquirido pelos comandos ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, e VACUUM FULL.

É também o *lock* por defeito para LOCK TABLE que não dizer modo de *lock*.

A notar que só ACCESS EXCLUSIVE bloqueia um comando SELECT.

Quando obtido, um *lock* é mantido até o fim da transacção. O PostgreSQL não tem um comando para libertar *locks* existentes, estes são libertos ao finalizar a transacção. Quando se usa *savepoints* e se obtiver um *lock* após um *savepoint* e fazer *rollback*, o *lock* obtido depois do *savepoint* é libertado.

5.6. Locks a nível de linha

Para além dos *locks* ao nível de tabela, existem *locks* ao nível de linha. Estes *locks* podem ser `EXCLUSIVE` ou `SHARED`. Um *lock* exclusive deste nível é automaticamente adquirido quando uma linha é modificada ou eliminada. O *lock* é mantido até um *commit* ou *rollback*, tal como nos *locks* ao nível de tabela. Estes *locks* só bloqueiam transacções que escrevem na mesma linha, ou seja, não afectam *queries* aos dados.

Para obter explicitamente um *lock* exclusivo tem de ser efectuado um `SELECT FOR UPDATE`. A notar que quando um *lock* exclusivo é obtido, podemos modificar a linha várias vezes sem haver conflitos.

Para obter um *lock* no modo *shared* podemos efectuar o comando `SELECT FOR SHARE`. Este *lock* não impede outras transacções de obter um *lock* do mesmo tipo mas nenhuma pode editar os dados ou obter um *lock* exclusive.

O PostgreSQL não guarda informação sobre linhas modificadas em memória, portanto não há limite nas linhas em *lock* a cada momento. Assim um *lock* pode causar uma escrita em disco.

Em adição aos *locks* de tabela e linha, *locks* ao nível de página são usados para controlar escrita/leitura no acesso à tabela de páginas no *shared buffer pool*. Estes *locks* são libertados imediatamente após uma linha ser obtida ou modificada e normalmente não são de importância aos desenvolvedores de aplicações.

5.7. Deadlocks

O uso explícito de *locks* pode aumentar a ocorrência de *deadlocks*, onde duas ou mais transacções mantêm um *lock* que o outro precisa. Por exemplo tendo uma tabela T e duas transacções t1 e t2, se t1 obtém um *lock* exclusivo à primeira linha e o t2 à segunda, se depois t1 tentar obter *lock* à segunda e t2 à primeira então nenhuma consegue prosseguir. O PostgreSQL automaticamente detecta situações de *deadlock* e resolve abortando uma das transacções dando à outra a possibilidade de continuar. Não há determinismo em qual aborta neste caso.

A melhor defesa contra *deadlocks* em geral é garantir que todas as aplicações obtém os *locks* numa ordem consistente. No exemplo descrito acima não aconteceria porque os dois tentariam primeiro obter o *lock* à primeira linha, um ia obter o lock e o outro ficario logo à espera.

Enquanto não é encontrado uma situação de *deadlocks* uma transacção fica à espera durante tempo indefinido atéconseguir obter os *locks* que precisa. Isto significa que é boa prática tentar obter *locks* durante pouco tempo, isto é, libertar assim que possível.

5.8. Advisory Locks

O PostgreSQL ainda fornece uma maneira de criar *locks* com sentido prático ao nível aplicacional. Estes são chamados *Advisory locks* e cabe ao programador da aplicação usa-las correctamente. Podem ser úteis para definir estratégias de *lock* que não são garantidas pelo modelo MVCC.

Existem duas maneiras de obter estes *locks*: ao nível de sessão ou ao nível de transacção. Quando adquirido ao nível de sessão, estes *locks* só são libertados quando a sessão é terminada. Ou seja, um *lock* destes obtido durante uma transacção é mantido mesmo se a mesma faz *commit* ou aborta. Estes *locks* são reentrantes e podem ser libertados explicitamente.

Locks ao nível de transacção, ao contrário das anteriores, são mais parecidos aos *locks* normais, isto é, são libertados no fim da transacção. Ao fazer um *lock* dentro duma transacção quando o *lock* já é mantido pela sessão, o *lock* é dado mesmo se houverem outras sessões à espera de obter o *lock*.

A semântica para estes *locks* é:

- `pg_advisory_lock(key)`
- `pg_advisory_lock_shared(key)`
- `pg_advisory_unlock(key)`
- `pg_advisory_unlock_shared(key)`
- `pg_advisory_unlock_all()`

Também podem ser usados com `pg_try` que devolve `TRUE` ou `FALSE` se conseguiu obter o *lock* ou não, não bloqueando.

Tal como nos outros *locks*, estes podem ser visualizados na *view* `pg_locks`.

Os *locks* são mantidos numa pool de memória partilhada cujo tamanho é definido pelas variáveis de configuração `max_locks_per_transaction` e `max_connections`. Tem que se exercer cuidado ao ponto de não encher esta memória, senão o servidor fica impossibilitado de dar *locks* de todo. Isto impõe um limite ao número de *locks* dados pelo servidor, dependendo da configuração.

6. Suporte para bases de dados distribuídas

O PostgreSQL, a partir da versão 9, inclui replicação baseado em enviar as mudanças feitas (*write-ahead logs*) para sistemas replicados *slave* de forma assíncrona. Esta versão também introduz a possibilidade de correr *read-only queries* nestes sistemas replicados, onde versões anteriores só deixavam fazer isso depois de o promover a ser o novo master. Assim podemos, de forma eficiente, partilhar o tráfego de leituras entre vários nós.

O Oracle tem, ao invés do PostgreSQL, supporte para replicação multi-master de bases de dados homogéneas usando o protocolo two-phase commit. Também suporta replicação master-slave criando snapshots.

A partir da versão 9.1, o PostgreSQL, também inclui replicação síncrona, isto é, para cada escrita o master espera até que pelo menos um dos nós *slave* escreva os dados no seu log. A definição de assíncrono ou síncrono pode ser especificado ao nível do servidor, base de dados, utilizador, sessão ou transacção. Isto permite que se possa ter um melhor controlo de desempenho quando algumas restrições ou *queries* podem ser mais flexíveis.

O sistema não suporta outras funcionalidades de replicação, como por exemplo a escrita em vários servidores. Para tal é necessário o uso de programas externos. Alguns destes e as suas funcionalidades podem ser encontrados na seguinte tabela:

Tabela 4- Suporte para replicação

Program	**Replication Method**	**Sync**	**Connection Pooling**	**Load Balancing**	**Query Partitioning**
pgpool-I	Statement-Based Middleware	Synchronous	Yes	Yes	No
Pgpool-II	Statement-Based Middleware	Synchronous	Yes	Yes	Yes
slony	Master-Slave	Asynchronous	No	No	No
Bucardo	Master-Master, Master-Slave	Asynchronous	No	No	No
Londiste	Master-Slave	Asynchronous	No	No	No

7. Outras características

7.1. XML

O PostgreSQL implementa um tipo de dados XML. Este tipo de dados faz validações da estrutura dos dados inseridos e tem funções específicas. Para criar um valor do tipo `xml` o PostgreSQL tem a seguinte função.

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

Pode ser usada como nos seguintes exemplos.

```
XMLPARSE (DOCUMENT '<?xml
version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>')

XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

Este tipo de dados não fornece comparação entre elementos porque não existe um algoritmo de comparação de dados XML bem definido. Desta forma não é possível fazer *queries* que compare uma coluna `xml` com um valor de pesquisa. Também não é possível criar índices para as colunas `xml`.

O PostgreSQL implementa funções de interrogação *XPath*, em seguida são apresentadas algumas dessas funções.

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

A função `XPATH` processa valores do tipo de dados `xml` e avalia expressões *XPATH 1.0* devolvendo um vector com valores XML.

```
XMLEXISTS(text PASSING [BY REF] xml [BY REF])
```

A função `XMLEXISTS` devolve `TRUE` se a expressão *XPATH* no primeiro argumento devolve algum elemento, e `FALSE` caso contrário.

7.2. Triggers

Na distribuição *standard* do PostgreSQL existem quatro linguagens procedimentais disponíveis para criar *triggers* e funções, *PL/pgSQL*, *PL/Tcl*, *PL/Perl* e *PL/Python*. Existem outras linguagens procedimentais disponíveis que requerem instalação por parte do utilizador.

O servidor da base de dados não tem conhecimento de como interpretar uma função escrita em linguagem procedimental. Por isso, passa a linguagem procedimental para uma função em C responsável por todo o processamento, fazer *parsing*, análise de sintaxe, execução, etc.

Um *trigger* pode ser criado com o comando `CREATE TRIGGER` da seguinte forma.

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [
OR ... ] }
    ON table
    [ FROM referenced_table_name ]
    { NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY
DEFERRED } }
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments )

where event can be one of:

    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

7.3. Tipos de dados

O PostgreSQL tem vários tipos de dados nativos disponíveis. Além disso, os utilizadores podem criar novos tipos de dados com o comando `CREATE TYPE`. A Tabela 5 - Tipos de dados apresenta os tipos de dados, na coluna designada de *Aliases* estão os nomes internos, usado pelo PostgreSQL, dos tipos de dados. Qualquer um dos dois nomes podem ser usados.

Tabela 5 - Tipos de dados

Name	Aliases	Description
bigint	int8	signed eight-byte integer
bigserial	serial8	autoincrementing eight-byte integer
bit [(*n*)]		fixed-length bit string
bit varying [(*n*)]	varbit	variable-length bit string
boolean	bool	logical Boolean (true/false)
box		rectangular box on a plane
bytea		binary data ("byte array")
character [(*n*)]	char [(*n*)]	fixed-length character string
character varying [(*n*)]	varchar [(*n*)]	variable-length character string
cidr		IPv4 or IPv6 network address
circle		circle on a plane
date		calendar date (year, month, day)
double precision	float8	double precision floating-point number (8 bytes)
inet		IPv4 or IPv6 host address
integer	int, int4	signed four-byte integer
interval [*fields*] [(*p*)]		time span
json		JSON data
line		infinite line on a plane
lseg		line segment on a plane
macaddr		MAC (Media Access Control) address
money		currency amount
numeric [(*p*, *s*)]	decimal [(*p*, *s*)]	exact numeric of selectable precision
path		geometric path on a plane
point		geometric point on a plane
polygon		closed geometric path on a plane
real	float4	single precision floating-point number (4 bytes)
smallint	int2	signed two-byte integer
smallserial	serial2	autoincrementing two-byte integer
serial	serial4	autoincrementing four-byte integer
text		variable-length character string
time [(*p*)] [without time		time of day (no time zone)

zone]		
time [(*p*)] with time zone	timetz	time of day, including time zone
timestamp [(*p*)] [without time zone]		date and time (no time zone)
timestamp [(*p*)] with time zone	timestamptz	date and time, including time zone
tsquery		text search query
tsvector		text search document
txid_snapshot		user-level transaction ID snapshot
uuid		universally unique identifier
xml		XML data

8. Referências

PostgreSQL 9.3.4 Documentation.

Oracle Documentation, *docs.oracle.com*.

W. Bridge, A.Joshi, M. Keihl, T. Lahiri, J.Loaiza, N.Macnaughton, "The Oracle Universal Server Buffer Manager," Oracle Corporation, 1997.

Abraham Silberschatz, Henry F. Korth, S. Sudarshan, "Database System Concepts," 6th edition, 2011.

https://wiki.postgresql.org/wiki/Main_Page

http://www.slideshare.net/Sammy17/chapter26-postgresql

http://www.cubrid.org/blog/dev-platform/postgresql-at-a-glance