Departamento de Informática – FCT – UNL

Relatório de Sistemas de Bases de Dados

PostgreSQL – v. 9.3 – Grupo 24

Realizado por:

Nuno Oliveira (nº42163)

Ricardo Monteiro (nº42070)

Rodrigo Carvalho (nº41818)

Professor:

José Alferes

1 de Junho de 2014

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Introdução

Este documento foi desenvolvido no âmbito da cadeira de Sistemas de Bases de Dados. Pretende-se com o mesmo a apresentação, de forma sumária, do funcionamento do sistema PostgreSQL através da análise dos seus constituíntes, com base no que foi leccionado durante as aulas da referida cadeira.

O documento está estruturado da seguinte forma: de seguida é feita uma breve introdução histórica do PostgreSQL prosseguindo com a análise mais detalhada do sistema. Essa parte inicia com uma análise ao sistema de armazenamento e de estrutura de ficheiros, seguido pelos modos de indexação e de *hashing*. Depois é estudado o ocessamento e optimização das *queries*, seguido pela gestão das transacções e controlo de concorrência. A secção seguinte apresenta o suporte que o PostgreSQL oferece para bases de dados distribuídas, terminando o documento com outras características relevantes do sistema em análise.

1.1 Introdução histórica do sistema

O Sistema de Gestão de Bases de Dados *PostgreSQL* é derivado de um projecto chamado *POSTGRES* que foi desenvolvido na Universidade da California em *Berkley* por uma equipa liderada pelo professor Michael Stonebraker. A implementação deste projecto começou em 1986 e só em 1988 foi apresentada uma primeira demonstração na conferência *ACM-SIGMOD*. Desde esse momento o *Postgres* tem vindo a ser utilizado para implementar algumas aplicações de produção e pesquisa, tais como sistemas de análise de dados financeiros, bases de dados de informações médicas, sistemas de informação geográfica, entre outros. Com o crescimento da comunidade de utilizadores, a manutenção do projecto tomava grande parte do tempo que podia ser usado para investigação associado à base de dados, por isso o projecto *Postgres* acabou na versão 4.2.

Em 1994, Andrew Yu e Jolly Chen, dois estudantes do laboratório de Stonebraker, substituiram a linguagem de *query* do *Postgres* (chamada *Postquel*) por um subconjunto extendido de *SQL*. E assim nasceu o *Postgres95*.

Em 1996, o *Postgres95* foi divulgado pela internet e passou a ser *open-source*. Também neste ano se deu o renomeamento do sistema para *PostgreSQL*, para reflectir a relação entre o original *POSTGRES* e as versões mais recentes que já contêm capacidade de *SQL*.

Em 1997 foi lançado a primeira versão do *PostgreSQL* e desde então o projecto é mantido por um grupo de *developers* de bases de dados e voluntários.

Actualmente o *PostgreSQL* vai na versão estável 9.3.4 (e na versão *beta* 1 de 9.4) e é considerado um sistema robusto, eficiente e avançado, com forte presença na indústria, como por exemplo, no *Reddit*, *Instagram*, *Yahoo!* e *MySpace*.

2 Armazenamento e *File Structure*

Nesta secção será abordada a forma como a estrutura de ficheiros e o seu armazenamento é feito no sistema de gestão da base de dados, bem como a sua comparação com o sistema *Oracle 11g*. Também vai ser explicado o mecanismo do *VACUUM*.

2.1 *Buffer Management*

O *PostgreSQL* implementa um gestor de *buffers* que tem como principal função o controlar a forma como a informação está guardada em disco e a sua dimensão. Também é responsável por manter em *cache* os blocos escritos e lidos, de forma a minimizar as chamadas ao sistema, acessos ao disco e processos morosos.

O *buffer* é um vector composto por várias entradas, em que cada entrada aponta para um bloco de dados com tamanho de 8KB e possui uma etiqueta que identifica o ficheiro ao qual essa entrada está a fazer *buffering* e qual o bloco do ficheiro que ele contém. O *buffer* também é composto por uma série de *flags* que identificam o estado da informação de cada bloco: *pinned* que indica que o bloco está a ser usado por um processo e que este só fica livre quando o processo acabar a sua execução e *dirty* que indica que a informação já foi modificada desde que esta foi lida do disco (útil para se saber que páginas necessitam de ser actualizadas). Nestas entradas é usado um contador para se poder estimar a popularidade do acesso de certas páginas.

Uma desvantagem da implementação de uma gestão de *buffers* própria é a possibilidade de haver conflito entre o sistema de gestão dos *buffers* do sistema operativo onde o sistema de gestão de base de dados está a ser executado.

Em comparação com o *Oracle 11g*, este organiza os *buffers* em duas listas: a *write list* que guarda os *buffers dirty* e a *LRU list* que guarda os *buffers* livre e *pinned*; ao contrário do *PostgreSQL* que usa quatro listas separadas para controlar as páginas de *cache* usadas mais recentemente e com mais frequência e também para optimizar dinamicamente a sua substituição com base no *work load*. Por fim, no *Oracle* são usadas páginas de 4kb, enquanto que no *PostgreSQL* são usadas páginas de 8kb.

2.1.1 Alocação de *buffers*

Quando um processo precisa de um *buffer* é chamada a operação `BufferAloc` que aloca o ficheiro ou bloco pedido. Caso esse bloco já exista em *cache*, este é *pinned* e de seguida é devolvido o bloco, caso contrário, é encontrado um novo *buffer* para guardar esses dados. Se não houver nenhum *buffer* vazio, a operação selecciona um *buffer* para evitar gastar espaço na criação de um novo. Por cada acesso que se

faz a uma página, o seu contador é incrementado por uma unidade, sendo esta a única forma de se poder incrementar o contador de uma página.

2.1.2 Despejo de um *buffer*

A primeira implementação feita para resolver o problema da decisão da remoção de uma entrada da *cache* foi a LRU (*Least Recently Used*), que consistia na remoção da entrada que tinha sido usada há mais tempo desde a sua última utilização. Esta estratégia (como não tem memória) levantou alguns problemas, como por exemplo, se uma página for acedida muitas vezes no passado e não depois não o for durante algum tempo é indistinguível de uma que foi acedida só uma vez ao mesmo tempo.

Para melhorar estes aspectos, e desde a versão 8.0 do *PostgreSQL*, é usado o contador criado pela operação `BufferAloc` e assim consegue-se ordenar as páginas pela sua ordem de popularidade. Assim a estratégia basicamente é a seguinte: é feito um *scan* circular à *cache*, qualquer página que tenha contador acima de zero ou que esteja *pinned* está a salvo do despejo, todas as páginas cujo o contador esteja a zero são despejadas e em todas as páginas que não foram despejadas é decrementado o seu contador em uma unidade. Ou seja, se uma página tiver o contador com o valor igual a cinco, esta "sobrevive"cinco passos até ser possível despejá-la.

2.1.3 Tipos de *buffer*

O *PostgreSQL* apresenta dois tipos de *buffer* que podem ser utilizados pelo sistema: os *Shared Buffers* (`shared_buffers`) que são usados para memória partilhada entre o Sistema Operativo e o Sistema de Gestão de base de dados e os *Temporary Buffers* (`temp_buffers`) que são usados para o acesso a tabelas temporárias da base de dados.

2.2 Sistema de ficheiros

O *PostgreSQL* não implementa o seu próprio *file system*, baseando-se no *file system* do sistema operativo onde está alojado. Toda a informação física usada pelo *cluster* de base de dados (tais como ficheiros de dados e de configuração) esta arquivadaa na directoria chamada *PGDATA*, sendo que o caminho para essa directoria em sistemas operativos baseados em *Unix* é `/var/lib/pgsql/data`. É permitido na mesma máquina existirem múltiplos *clusters* controlados por instâncias de servidor diferentes.

A directoria *PGDATA* contém várias subdirectorias e ficheiros de controle (além dos ficheiros de configuração do *cluster*), tais como:

Item	Descrição
`PG_VERSION`	Ficheiro que contém o número da versão do *PostgreSQL*
`base`	Subdirectoria que contém as subdirectorias por cada base de dados
`global`	Subdirectoria que contém tabelas que pertencem a todo o *cluster*
`pg_clog`	Subdirectoria que contém informação sobre estado de commit de transacções
`pg_multixact`	Subdirectoria que contém informação sobre o estado das multi-transacções
`pg_notify`	Subdirectoria que contém a informação sobre o estado *LISTEN/NOTIFY*
`pg_serial`	Subdirectoria que contém a informação sobre as transacções *committed serializable*
`pg_stat_tmp`	Subdirectoria que contém ficheiros temporários para o subsistema de estatísticas
`pg_subtrans`	Subdirectoria que contém informação sobre o estado de substransacções
`pg_tblspc`	Subdirectoria que contém *symbolic links* para *tablespaces*
`pg_twophase`	Subdirectoria que contém ficheiros de estado para transacções preparadas
`pg_xlog`	Subdirectoria que contém ficheiros WAL (*Write Ahead Log*)
`postmaster.opts`	Ficheiro que guarda as opções de linhas de código que o servidor tinha na última vez que foi inicializado
`postmaster.pid`	Ficheiro que guarda o PID do processo do *postmaster* corrente, o caminho da directoria dos dados do *cluster*, *timestamp* do inicio do *postmaster*, entre outros dados
`pg_snapshots`	Subdirectoria que guarda os *snapshots* exportados

Tabela 1: *Conteúdo do PGDATA*

Cada tabela e cada *index* é guardado em ficheiros diferentes, são nomeados pelo número *filenode* da tabela ou do *index* (estes valores podem ser consultados em `pg_class.relfilenode`) e possuem um *free space map* que guarda a informação sobre o espaço livre disponível na relação. As tabelas possuem um *visibility map* que serve para rastrear as páginas que são conhecidas por não ter tuplos "mortos".

As tabelas são guardadas em *heap files*, sendo que estes têm a forma de *slotted pages* (da mesma forma que o *Oracle 11g*). Cada registo guardado na *heap* tem um identificador.

Por omissão, cada segmento do *PostgreSQL* tem um tamanho de 1GB, ou seja, quando a dimensão de uma tabela ou *index* ultrapassa este valor, estes são divididos em diferentes segmentos de 1gb cada. O tamanho do segmento por omissão pode ser configurado através da operação `-with-segsize` em tempo de compilação.

Uma tabela que tenha colunas com entradas potencialmente grandes, vai ter uma tabela *TOAST* associada, que é usada para armazenamento *out-of-line* de valores que são grandes demais para manter na

tabela original.

Os conceitos de *TOAST*, *Free Space Map* e *Visibility Map* vão ser detalhados nas secções seguintes

2.2.1 TOAST

O *PostgreSQL* como usa um tamanho fixo de páginas e como não permite que os tuplos se estendam por várias páginas, não permite guardar directamente valores grandes. Para ultrapassar esta lacuna, esses valores grandes são compressos ou partidos em várias linhas. Isto é o mecanismo é conhecido como *TOAST*. Esta técnica só pode ser aplicada a tabelas que contenham valores de tamanho variável (*varlena*).

Ao longo do funcionamento da base de dados, o *TOAST* apodera-se dos dois primeiros *bits* da palavra de comprimento *varlena* para codificar o estado e as condições em que os dados estão guardados, usando-os como valores lógicos. Por exemplo, se ambos os *bits* tiverem o valor zero, quer dizer que os dados estão *un-TOASTed*; quando o primeiro *bit* está activo, então os dados seguintes representam um apontador para outra zona de memória onde se poderá encontrar o valor pretendido; quando o segundo *bit* está activo, pode-se concluir que o valor foi compresso e tem de ser descomprimido antes de ser usado.

Existem quatro estratégias diferentes para armazenar colunas *TOAST-able*:

- `PLAIN` - que previne tanto a compressão quer o armazenamento *out-of-line*. Esta é a única estratégia para colunas de tipos de dados *non-TOAST-able*;
- `EXTENDED` - permite a compressão e o armazenamento *out-of-line*. Esta é a estratégia por omissão para maioria dos tipos de dados *TOAST-able*. Primeiro tenta-se a compressão e depois a decomposição se a linha continuar grande;
- `EXTERNAL` - permite o armazenamento *out-of-line* mas não a compressão. Esta estratégia optimiza as operações de *substring* porque essas optimizações estão melhoradas para capturar só as partes do valor que está decomposto e quando não está compresso;
- `MAIN` - permte a compressão e a decomposição. Nesta estratégia só é feita a decomposição em casos extremos;

Cada tipo de dados *TOAST-able* tem uma estratégia por omissão, mas a estratégia pode ser alterada com o comando `ALTER TABLE SET STORAGE`.

2.2.2 Free Space Map

Cada relação de *Heap* ou de *index* (exceptuando os *hash index*), tem um *Free Space Map* (FSM) que mantém o controlo do espaço disponível da respectiva relação.

Esta estrutura está organizada como uma árvore de vários *map* em que as páginas da *bottom level* guardam o espaço disponível em cada página de *head* ou de *index*, usando um *byte* para representar cada página. Os níveis superiores simplesmente agregam informação dos níveis mais baixos.

Cada página FSM é uma árvore binária armazenada num vector com um *byte* por nó. Cada nó folha representa uma *heap page* ou uma página FSM de um nível inferior. Em cada nó que não seja folha, o maior valor dos filhos é guardado. O valor máximo dos nós folha é depois guardado na raiz.

2.2.3 Visibility Map

Cada relação de *heap* tem um *Visibility Map* para manter o controlo de quais as páginas que contêm apenas os tuplos que são visíveis para todas as transacções activas. Está armazenado juntamente com os dados principais da relação num relação *fork*, nomeada pelo nome da relação principal mais o sufixo *_vm*.

O *Visibility Map* armazena um *bit* por *heap page*. Um *bit* activo significa que todos os tuplos daquela página são visíveis para todas as transacções. Esta informação também pode ser usada para *scans* de *index* para responder a perguntas usando só o tuplo do *index*.

2.3 Layout das páginas da base de dados

Cada tabela e cada *index* é armazenado num vector de páginas de tamanho fixo de 8kb (este valor pode ser alterado em tempo de compilação).

Como todas as páginas são logicamente equivalentes numa tabela, uma linha da tabela pode ser armazenada em qualquer página.

Nos índices, a primeira página é reservada para armazenar uma *metapage* que guarda informação de controlo. Dentro do índice pode haver vários tipos de páginas dependendo do tipo de acesso.

Cada página deste sistema de base de dados apresenta o seguinte conteúdo:

Item	Descrição
`PageHeaderData`	Contém a informação da página, incluindo apontadores para os espaços livres. Tem comprimento de 24 *bytes*.
`ItemIdData`	Vector de pares `<offset,length>` que apontam para os Items. Tem comprimento de 4 *bytes* por cada item.
`Free space`	Espaço não alocado. Os novos apontadores de items são alocados a partir do ínicio desta área e os items a partir do fim.
`Items`	Os items da página.
`Special Space`	Espaço específico para os métodos de acesso de indices. São usados diferentes métodos para diferentes tipos de dados. Em tabelas normais este espaço está vazio.

Tabela 2: *Conteúdo de uma página*

2.4 Clustering

O *Clustering* é uma forma de ordenamento físico de tabelas de forma a melhorar o tempo de acesso ao disco. Pode ser definido como a junção de dados em espaços contínuos das componentes de armazenamento, ou seja, os dados mais requisitados são armazenados na mesma região do disco. Assim o sistema melhora a sua *performance*.

No PostgreSQL esta forma pode ser implementada usando o método `CLUSTER` que faz com que seja feito um*cluster* a uma tabela usando um ficheiro de *index*. Isto faz com que a tabela seja reordenada pelo índice indicado no método. É uma operação efectuada uma única vez, ou seja, quando uma tabela é actualizada após a execução do método, as mudanças já não vão ser *clustered*.

Para se poder fazer *cluster* a uma tabela, faz-se da seguinte forma: `CLUSTER [VERBOSE] table_name [USING index_name]`, onde `VERBOSE` é uma *flag* que requere a apresentação de dados para *debugging*, `table_name` é o nome da tabela a ser *clustered* e `index_name` o nome do índice usado.

Para melhorar o desempenho de perguntas nas quais existe uma junção de tabelas usa-se o conceito de *multitable clustering*. Essa funcionalidade não está implementada no PostgreSQL, ao contrário do Oracle 11g.

2.5 Partições

O particionamento refere-se à divisão lógica de uma tabela de grandes dimensões em pedaços mais pequenos. Este mecanismo trouxe as seguintes vantagens:

- A *performance* de *queries* pode ser melhorada nos casos em que a maior parte das linhas mais requisitadas de uma tabela estão presentes numa só partição ou num número baixo de partições.
- Quando as consultas ou as actualizações acedem a uma grande percentagem de dados numa única partição, o desempenho pode ser melhorado aproveitando o *scan* sequencial dessa partição em vez de usar um índice ou e a acessos aleatórios espalhados por toda a tabela.
- Para carregamentos e eliminações em massa pode ser feito pela criação e remoção de partições.
- Os dados raramente usados podem ser migrados para armazenamentos físicos mais baratos e mais lentos.

O PostgreSQL suporta particionamento por via de herança de tabelas, ou seja, cria-se uma tabela com os atributos da tabela "mãe"e replicam-se os dados dessa tabela para a tabela "filho".

O Oracle 11g também suporta particionamento e possibilita a criação de vários tipos de particionamentos: *range partitioning* (que funciona através de bandas de dados, por exemplo partições com datas desde Janeiro até Junho de 2014), *hash partitioning* (um algoritmo de *hash* que decide em qual partição um tuplo deve estar), *list partitioning* (a distribuição dos dados é definida por uma lista de valores), entre outros tipos.

2.6 VACUUM

Numa execução normal do PostgreSQL, quando são feitos remoções ou actualizações de tuplos, esses não são removidos fisicamente das suas tabelas. Esses tuplos ficam assim classificados como "tuplos mortos". A operação *VACUUM* trata de recuperar o espaço ocupado por esses tuplos. Assim é necessário que esta operação seja feita periodicamente e, especialmente, em tabelas que são frequentemente actualizadas. No fim de contas, pode-se encarar esta operação como o *garbage-collector* do PostgreSQL.

Esta operação tem várias formas de execução, como o processamento de todas as tabelas da base de dados corrente ou só de uma tabela (escolhida através do parâmetro da operação), bem como a junção da operação de análise para se poder actualizar as estatísticas da base de dados após a execução da operação.

3 Indexação e *Hashing*

Nesta secção será abordado o funcionamento da indexação e do *hashing* do sistema, nomeadamente aqueles que são suportados, e ainda a sua comparação com o *Oracle 11g*.

3.1 Criação de índices

Em bases de dados, índices são um modo de melhoramento da sua performance, pois permitem a pesquisa, análise e obtenção dos tuplos pretendidos na *query* realizada ao sistema. Esta melhoria da performance, principalmente no aumento da rapidez da obtenção dos referidos tuplos, é um factor fulcral quando estão em causa bases de dados de grandes dimensões. Caso não existissem ou não se usassem índices, a pesquisa das bases de dados teria que ser total, isto é, ter-se-ia que percorrer todas as tabelas de forma sequencial, acedendo a cada tuplo e avaliando, para cada um, se está de acordo com a *query* ou não. Este método é claramente ineficiente se se pensar, por exemplo, numa tabela de movimentos de um supermercado, onde existirão certamente milhões de tuplos, e onde o percorrer dessa tabela para obtenção dos tuplos pretendidos iria ser altamente moroso.

Assim, através dos comandos

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table_name [ USING method ]
(  column_name | ( expression )  [ COLLATE collation ] [ opclass ]
[ ASC | DESC ] [ NULLS  FIRST | LAST  ] [, ...]  )
[ WITH ( storage_parameter = value [, ...  ]  )  ]
[ TABLESPACE tablespace_name ]
[ WHERE predicate ]
```

é, então, criando um novo índice; e através do comando `DROP INDEX index;` é removido o índice `index` passado como argumento.

Após a sua criação, o sistema encarrega-se de o manter actualizado à medida que a tabela sobre a qual o índice está implementado for sendo modificada, não sendo necessária intervenção alguma. Ao manter o índice actualizado a cada modificação, no momento em que é necessário utilizar esse índice o sistema sabe que o plano que formar será o mais eficiente (de acordo com o índice em causa).

No entanto, como em tudo, existem desvantagens na utilização de índices. Estes causam a inserção de *overhead* no sistema, na sua criação, manutenção e durante a sua utilização. Por isto, é necessária alguma prudência na sua utilização, assim como a remoção caso o(s) índice(s) não seja(m) utilizado(s) com muita

frequência por parte do sistema. Uma outra desvantagem é a não permissão de utilização forçada de um índice criado por um utilizador numa *query* específica. Assume-se assim que o sistema calcula o melhor plano de execução e usa o índice que mais lhe convier.

3.2 Estruturas suportadas

O PostgreSQL suporta, obviamente, a criação de índices, nomeadamente os índices B+, *Hash*, GiST, SP-GiST e GIN. Os índices referidos são conhecidos pelo sistema como sendo índices secundários, ou seja, o índice está separado da tabela à qual pertence. De referir que o PostgreSQL não suporta índices *bitmap* persistentes.

3.2.1 Árvores B+

Este é o tipo de índice que é criado por omissão através do comando `CREATE INDEX index_name ON table (column)` . Estas árvores são baseadas na implementação das *B-link trees* de Lehman-Yao, e apresentam muito bons resultados sob *queries* com operações em que os dados podem ser ordenados de algum modo. Essas são operações de comparação, nomeadamente `<, <=, =, >=, >, BETWEEN, IN` e ainda condições do tipo `IS NULL` e `IS NOT NULL` , em que o *query planner* irá utilizar a árvore B+, especialmente quando um coluna indexada está na comparação. É ainda possível usar, por parte do optimizador, os operadores `LIKE` e caso o padrão pretendido seja uma constante e esteja acoplado no início de uma *string* como, por exemplo, `column LIKE 'foo%'`.

Comparando com o sistema Oracle 11g, ambos suportam índices B+.

3.2.2 Hash

Neste sistema, os índices de *hash* são uma implementação do *linear hashing* de Litwin's, e são criados recorrendo ao comando `CREATE INDEX index_name ON table USING hash (column)` . Apenas lidam com comparações simples de igualdade, isto é, o optimizador de *queries* só irá considerar a utilização de um destes índices quando uma coluna indexada esteja implicada numa comparação que contenha o operador de igualdade `=` . Comparados com os B+, são maiores e, consequentemente, implicam custos de manutenção bastante superiores. Uma outra desvantagem é que estes índices, por não serem *WAL-logged*, não permitem recuperação de falhas, o que implica problemas em replicações (discutidas no capítulo 6). Uma terceira desvantagem é o facto de em virtude do *hash* ser dinâmico, não existe a função de *rehashing*. Postas estas adversidades, tornam-se claras as vantagens na utilização dos índices B+.

Em comparação com o Oracle 11g, este não possui suporte para este tipo de índices.

3.2.3 GiST

Generalized Search Tree, árvore de procura generalizada, é um tipo de árvore equilibrada que serve de base à implementação de outras estruturas, como por exemplo a B+. Portanto, o GiST não é um tipo de índice mas sim uma estrutura fornecida pelo PostgreSQL que permite a implementação de vários índices. Os operadores permitidos pelo GiST são

```
<<, &<, &>, >>, <<|, &<|, |>&, |>>, @>, <@, ~=, &&
```

Estes índices são também utilizados na optimização de pesquisas *"nearest neighbours"*, por exemplo, encontrar os 10 locais mais próximos de um ponto dado:

```
SELECT * FROM places ORDER BY location <-> point '(101,406)' LIMIT 10;
```

Este tipo de índice tem perdas, isto é, pode produzir falsos resultados, o que leva a que seja necessário verificar a linha da tabela e eliminar esses resultados errados. Felizmente o PostgreSQL trata deste processo de forma automática e sempre que necessário. No entanto, estas perdas causam diminuição da performance resultante da obtecção dos falsos resultados da tabela, sendo esta a sua maior desvantagem. De referir que caso sejam utilizadas *queries* standard, não existem perdas, pelo que a diminuição da performance depende apenas do número de palavras únicas.

No caso do sistema Oracle, este não possui suporte para este tipo de índices.

3.2.4 SP-GiST

Este tipo de índice, à semelhança do GiST, fornece uma estrutura de suporte a vários tipos de pesquisas, o que permite diversos tipos de estruturas. Os operadores suportados pelo PostgreSQL para este tipo de índice são

```
<<, >>, ~=, <@, <\^, >\^
```

De igual modo ao GiST, o Oracle 11g também não suporta este tipo de índice.

3.2.5 GIN

Este é um índice invertido que é utilizado em índices que contêm mais do que uma chave, como por exemplo os *arrays*. É muito utilizado para fazer pesquisa em texto, onde a palavra dada funciona como *key* e a sua localização é o valor.

À semelhança do GiST, o GIN também suporta diferentes estratégias de indexação definidas pelo utilizador, tendo ainda os seus próprios operadores, nomeadamente `<@, @>, =, &&` . A criação de um índice GIN é dada por

```
CREATE INDEX index_name ON table USEING gin (column);
```

A utilização destes índices traz vantagens e devantagens, neste caso em comparação com o GiST, seguidamente descritas. Uma vantagem é o facto de a pesquisa realizada pelo GIN ser três vezes mais rápida que o GiST. No entanto, existe a desvantagem de o GIN demorar três vezes mais a criar índices, e de ser duas a três vezes maior que os GiST. No campo das actualizações, os índices GIN são mais lentos a actualizar. Em suma, os índices GIN são melhores para dados estáticos, visto que a pesquisa é mais rápida; por outro lado, os GiST são melhores para dados dinâmicos, uma vez que os seus índices são actualizados mais rapidamente. Note-se que o tempo de construção do índice pode ser melhorado se o parâmetro `maintenance_work_mem` for aumentado, o que não é possível de ser feito no GiST.

No caso do Oracle 11g, este tipo de índices não é suportado.

3.3 Índices multicoluna

Um índice pode ser definido em mais de uma coluna de uma tabela onde, neste caso, apenas os índices do tipo B+, GiST e GIN permitem multicoluna, até 32 colunas. Este tipo de índice pode ser criado através do comando

```
CREATE INDEX test_mm_idx ON test (max, min); .
```

Estes índices apenas se tornam vantajosos quando as *queries* referenciam as colunas pela mesma ordem do índice, pelo que nem sempre é preferível a utilização destes índices versus os unicoluna.

Um índice B+ pode ser utilizado em condições de pesquisa que envolvem um subconjunto das colunas desse índice, no entanto, torna-se mais eficiente quando existem restrições nas colunas mais à esquerda. Aqui a ideia subjacente é a utilização de restrições de igualdade nas referidas colunas mais à esquerda juntamente com outras restrições de desigualdade na primeira coluna que não tenha uma restrição de igualdade, o que limita a porção do índice que é percorrido, traduzindo-se numa boa performance. As restrições das colunas à direita são verificadas no próprio índice, o que resulta numa redução de acessos à tabela (mas não impõem limites na porção de índice que é percorrida), novamente traduzindo-se num bom desempenho. Como exemplo, dado um índice sobre as colunas `(a, b, c)` e uma condição `WHERE a = 5 AND b >= 42 AND c < 77` , seria necessário percorrer o índice desde a primeira entrada com `a = 5` e `b = 42` até à última entrada com `a = 5` , ignorando as entradas com `c >= 77` (embora sejam na mesma analisadas).

No caso de um índice multicoluna do tipo GIN, este pode ser utilizado com qualquer tipo de condições que relacionem subconjuntos das colunas do índice. Em oposição aos índices B+ e GiST, o custo da pesquisa é o mesmo independentemente do número de colunas indexadas utilizadas pelas condições especificadas na *query*.

No caso do sistema Oracle, os índices multicoluna também são suportados, aplicando-se as mesmas

condicionantes.

3.4 Combinação de múltiplos índices

O PostgreSQL permite combinar índices múltiplos (ou vários usos do mesmo índice), de modo a evitar os casos em que não é possível implementar recorrendo apenas a um índice. No caso de a *query* ser baseada no operador `OR`, é necessário ter em atenção que não é de todo vantajosa a utilização directa de índices: pode ser utilizada a unificação. Para realizar a combinação, o sistema percorre cada um dos índices necessários e cria, em memória, um índice *bitmap* que indica a localização dos tuplos que verificam as condições da *query*. É então realizada a operação de `AND` ou `OR`, consoante o necessário pela *query*, sendo depois extrídos os tuplos. Em virtude de os tuplos estarem num *bitmap*, a ordem de saída é de acordo com a estrutura física dos dados, pelo que é ainda necessária uma nova ordenação (caso se esteja na presença da cláusula `ORDER BY`) para que o resultado esteja ordenado de acordo com os índices iniciais. Esta necessidade de uma ordenação extra é a causadora do aumento do tempo de execução, pelo que o planeador do sistema irá decidir sobre qual dos tipos de índice optar, sendo que na maioria dos casos este opta pela utilização de índices únicos, mesmo que outros índices estejam disponíveis.

3.5 Índices e ordenação

Um índice, além de devolver os resultados ordenados, ordem esta 'por si' escolhida, permite ainda a utilização da cláusula `ORDER BY`, em que o sistema faz uma análise ao índice, no caso de ordenação por coluna indexada, ou análise à tabela e realiza a ordenação dada pela referida cláusula. Assim, permite-se que a cláusula seja respeitada não sendo necessário uma ordenação extra. É o caso dos índices B+, os únicos cujo resultado é ordenado, e suportados pelo PostgreSQL, em que o resultado é apresentado por ordem crescente e os valores *null* são dipostos no fim.

No geral, os índices apenas são melhores que uma ordenação explícita para tabelas de tamanho relativamente reduzido, uma vez que para as de tamanhos grandes o número de acessos ao disco no caso da ordenação explícita é consideravelmente inferior à utilização de índices. Existe ainda um caso especial, a cláusula `ORDER BY` combinada com `LIMIT n`, em que a ordenação explícita trata apenas nos `n` primeiros tuplos; na existência de um índice que corresponda à cláusula, os ditos `n` primeiros tuplos podem ser prontamente retornadas, não sendo necessário uma análise.

Para organizar a ordenação dos índices B+, é possível utilizar as opções `ASC, DESC, NULLS FIRST, NULLS LAST`, como por exemplo

```
CREATE INDEX test1 ON test2 (info NULLS FIRST);
CREATE INDEX test3 ON test3 (id DESC NULLS LAST);
```

3.6 Índices únicos

Este tipo de índice é utilizado para que haja a garantia de unicidade dum determinado valor presente numa coluna ou de valores combinados pertencentes a mais que uma coluna. Esta garantia é apenas dada pelos índices B+, aqueles que podem ser decladados como únicos. No seguimento das restrições da linguagem SQL, no PostgreSQL a comparação de valores `NULL` não é considerada como sendo igual. Este sistema cria, de forma automática, um índice único quando um restrição única ou uma chave primária é definida para uma tabela específica. Um índice único é criado a partir dos comandos

`CREATE UNIQUE INDEX index ON table (column [, ...]);`.

No entanto, o modo mais correcto a adoptar para adicionar restrições a colunas que já são únicas é através da alteração da tabela. O comando `ALTER TABLE index ADD CONSTRAINT` permite alterar a tabela, adicionando-lhe uma restrição.

3.7 Índices parciais

Este é um índice que é contruído sobre um subconjunto de uma tabela, definido por uma expressão condicional, que contém apenas as entradas para os tuplos da tabela que satisfazem a referida expressão. Uma forte razão para a criação e utilização de índices parciais é o facto de tentar evitar o mais possível a existência de valores comuns, uma vez que qualquer *query* que os pesquise não irá utilizar um índice. Portanto, é desnecessário manter esses tuplos num índice, o que se traduzirá numa redução do tamanho do índice e no aumento da velocidade das *queries* que usem esse índice e das operações de actualização. A criação destes índices é feita recorrendo, por exemplo, aos comandos

`CREATE UNIQUE INDEX index ON table (column) WHERE conditional_expression;`

O sistema Oracle 11g não possui suporte para os índices parciais.

3.8 Índices de *Hash*

Este tipo de índices apenas suporta operações de igualdade, pelo que só serão escolhidos na presença de *queries* do tipo `SELECT attr FROM table WHERE col = val`. A sua criação é através de

`CREATE INDEX index ON table USING hash (column)`.

As referidas operações efectuadas nestes índices não são *WAL-logged*, pelo que existe a possibilidade de ocorrência de erros e inconsistências quando, por exemplo, é sofrida uma falha de dados por escrever.

Uma vez que apenas suporta as igualdades numa coluna, e devido ao facto do *hash* ser dinâmico, este tipo de índice é menos vantajoso que os B+, pelo que se devem evitar.

No Oracle, é suportado o índice de *hash*.

3.9 Índices e organização de ficheiros

Na criação de índices não há a definição da organização de ficheiros, apenas o nível de ocupação do índice, isto é, o modo como o PostgreSQLune os componentes do índice para que o acesso sequencial seja mais rápido.

3.10 Inconsistência temporária

Uma vez que existe concorrência no PostgreSQL, o acesso a estruturas de índice é necessário de ser controlado. O mecanismo `LOCK` permite controlar o acesso às estruturas, nomeadamente aquelas em que existe a noção de concorrência. Estes factos permitem que existam inconsistências, situação recorrente nas transacções (em mais detalhe no capítulo 5), uma vez que entre o início e o fim da transacção os dados são alterados. O PostgreSQL permite diferir a verificação das restrições, para que sejam verificadas ou no fim da transacção, isto é, após o `commit;`, ou imediatamente após cada instrução dada. O comando que permite criar este deferimento é

```
SET CONSTRAINTS ALL | name [...]    DEFERRED | IMMEDIATE .
```

No sistema em análise, apenas as restrições do tipo `UNIQUE, PRIMARY KEY, REFERENCES` e `EXCLUDE` podem utilizar este comando.

No caso do Oracle 11g, também é possível definir quando uma verificação pode ser feita, o que permite que exista a inconsistência atrás referida.

4 Processamento e Optimização de *queries*

Quando o PostgreSQL recebe uma *query*, são executados alguns passos para que esta seja analisada, executada e os resultados devolvidos. A análise transforma a *query* numa representação interna, passando por várias transformações resultando num plano que é utilizado pelo executor. Assim, será apresentado o percurso que uma *query* percorre desde que um utilizador do sistema a insere até à devolução dos resultados, seguido por uma breve descrição dos algoritmos que o PostgreSQL implementa (selecção, ordenação e junção) e a utilização de estatísticas por parte do sistema.

4.1 Caminho de uma *query*

Quando uma *query* é executada por um utilizador, existem alguns passos que têm de acontecer de modo a que o PostgreSQL consiga devolver os resultados esperados. O primeiro é a ligação ao servidor do PostgreSQL em que a aplicação local, após a conexão, envia a *query* para o servidor e fica a aguardar pela resposta; de seguida é realizado um *parsing* à *query* e é criada a árvore da *query*; é então feita uma reescrita, em que são pesquisadas regras na árvore anteriormente criada e estas são substituídas pelo valor real; a *query* reescrita é analisada pelo planificador e este cria o plano de execução optimizado da *query*; por fim, este plano é executado pelo executor e os resultados obtidos são devolvidos. As secções seguintes explicam em mais detalhe os passos referidos. A **Figura 1** mostra a visão geral do processo de execução de uma *query*, e a **Figura 2** exemplifica a sequência dos passos de execução de uma *query* no PostgreSQL.

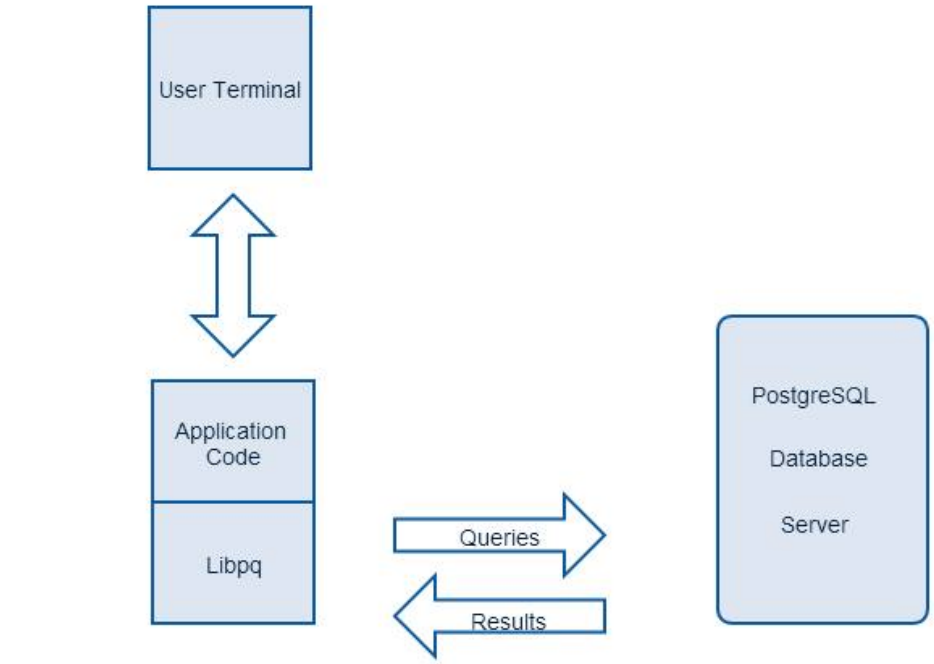

Figura 1: *Esquema geral do funcionamento do PostgreSQL.*

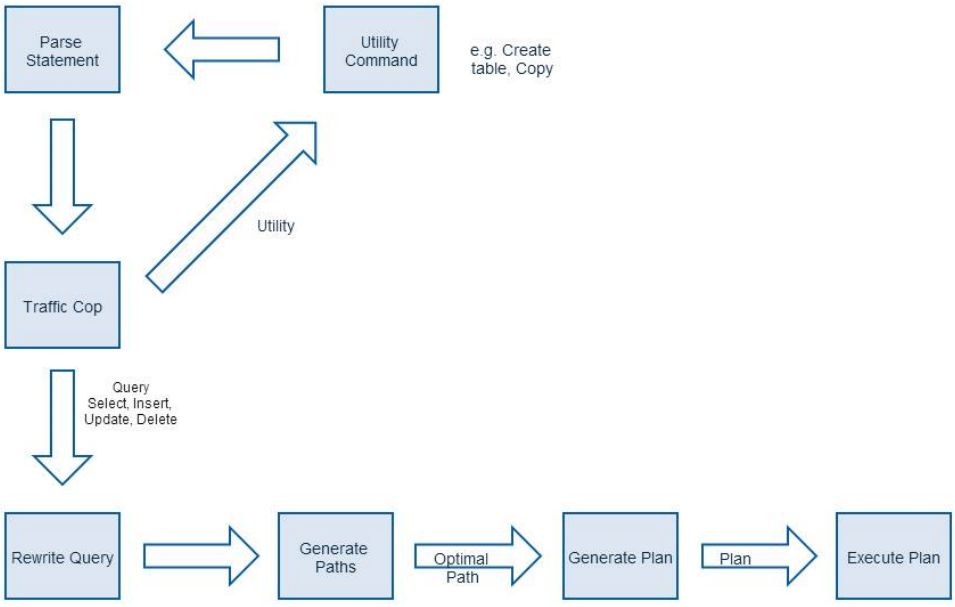

Figura 2: *Fases de execução de uma query.*

4.1.1 Ligação ao servidor do PostgreSQL

Aqui é utilizado um modelo cliente-servidor, com um processo-cliente ligado a um só processo-servidor. Esta ligação é estabelecida através do protocolo TCP/IP. Para a conexão ser realizada, o processo do lado do cliente apenas necessita de conhecer o protocolo do PostgreSQL sendo usual a utilização da biblioteca `libpq` da linguagem C. Uma vez estabelecida a ligação, o cliente envia a *query* para o servidor e este prossegue com os restantes passos.

4.1.2 *Parser*

Este passo na análise de uma *query* é responsável por verificar a sintaxe, construir a *parse tree* e analisar a semântica das tabelas, criando no fim a *query tree*.

Quando o *parser* recebe a *query* verifica se esta está sintaticamente correcta. Se estiver, é então criada a *parse tree*; se não, é retornado um erro. Este é responsável por identificar e executar as regras de gramática e consequentes acções para cada regra existente. De seguida entra em acção o *lexer*, responsável pelo reconhecimento das palavras reservadas do SQL, que retorna cada um dos comandos reconhecidos ao *parser*.

Após a criação da *parse tree* é necessário realizar a análise semântica. Esta análise vai permitir compreender quais as tabelas e operadores que estão referenciados pela *query*. São aqui interpretadas todas as regras, tabelas, junções e operadores pertencentes à *query*. No fim, é construída a *query tree*, semelhante à *parse tree* mas onde é adicionada informação sobre tipos de dados de colunas e resultados de expressões.

Como exemplo, a *query*

```
SELECT customer_name, balance FROM customers WHERE balance > 0 ORDER BY balance
```

irá prduzir a *parse tree*, presente na **Figura 3.**

4.1.3 Reescrita

Nesta fase são aplicadas as regras que o utilizador definiu. Um exemplo de reescrita é a criação de uma vista: caso existam, o seu valor é substituído pelo valor real na árvore . Quando uma vista é criada, é também criada uma regra própria que é accionada para actualizar as referências para as tabelas. No entanto, visto que a física não é materializada, torna-se obrigatória a reescrita da *query* a cada acesso, trazendo como vantagem um melhor desempenho visto não ser necessária a actualização dessas vistas.

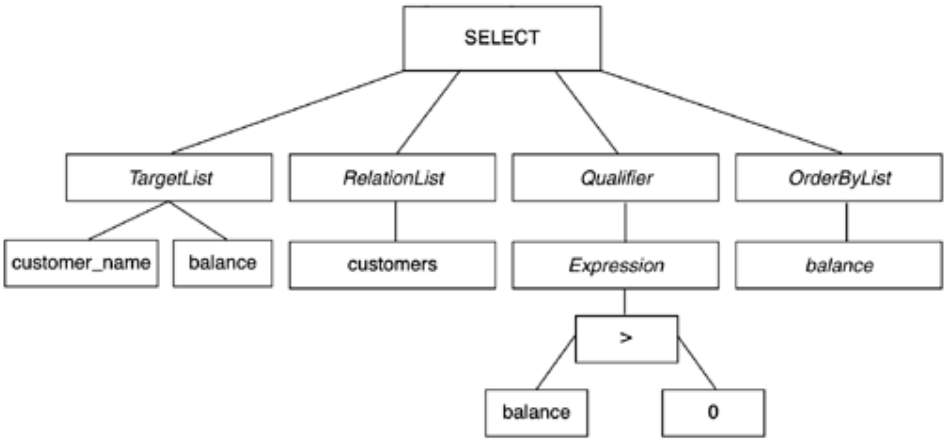

Figura 3: *Parse tree criada pelo parser.*

4.1.4 Planificador

É neste passo que o PostgreSQL determina qual(is) o(s) plano(s) de execução possíveis e, de entre estes, opta por aquele cujo custo seja inferior. Não existe um só plano para uma dada *query*, mesmo sendo óptimo. Existe ainda a possibilidade de serem feitas simplificações à medida que o optimizador avalia o plano, uma vez que o plano é percorrido de baixo para cima. Para escolher qual o plano a executar, o optimizador analisa cada um deles escolhendo o melhor no fim (isto, claro, se o custo da análise não for superior ao custo de executar um plano que não seja o melhor). Se existir a necessidade de realizar junções, existem três possíveis estratégias, *nested loop join, merge-join* e *hash-join*. Estas estão mais detalhadas na secção seguinte.

Caso necessário, é então utilizado o *Genetic Query Optimizer*, que permite resolver o problema do custo de tempo e de memória consequente da criação de uma ordem de junção. Este algoritmo utiliza o planeador existente no PostgreSQL para a geração de planos de scan, sendo depois os planos de execução criados a partir de um algoritmo genético. Ocorrem algumas iterações, em que a cada uma são descartados os planos menos eficientes e criados novos para serem avaliados quanto à sua eficácia. Quando for encontrado o melhor plano, este é utilizado na geração do plano de execução final. No término deste passo, o optimizador envia o plano escolhido ao executor.

4.1.5 Executor

Neste último passo do caminho de execução de uma *query*, o plano cujo custo é inferior é entregue ao executor para que seja possível retornar os dados pedidos na *query*. Este avalia as *queries* do tipo `SELECT`, `INSERT`, `UPDATE` e `DELETE`. Ao melhor plano é aplicado o mecanismo de *demand-pull pipeline*, isto é, o nó da árvore a ser executado deve devolver um tuplo ou informar que não possui mais tuplos ao seu nó-pai.

4.2 Algoritmos de scan

4.2.1 Sequential scan

Este algoritmo de selecção assenta na procura linear numa tabela, isto é, desde o primeiro até ao último bloco.

4.2.2 Index scan

Permite realizar pesquisa em índices, quer estes sejam primários ou secundários, requerendo um atributo de junção. À medida que percorre os índices, devolve os tuplos que verificam a condição. Este varrimento ocorre nos casos em que existem operações de igualdade e desigualdade, ou seja, $=, <, <=, >$ e $>=$, embora o PostgreSQL não o utilize muito devido à possibilidade de operações de I/O em demasia e desnecessárias (nestes casos o algoritmo *sequetial scan* é preferível).

4.2.3 Bitmap index scan

Este algoritmo necessita de uma estrutura auxiliar, um *bitmap* que se trata de um conjunto de bits representante dos tuplos que possuem um determinado valor na estrutura. A principal vantagem na sua utilização é o facto de ser bastante eficiente nas operações de `COUNT` e de `AVG` uma vez que estas apenas necessitam de uma contagem do número de bits correspondentes ao referido valor, situação que não se verifica nos casos em que atributos têm muitos valores.

4.3 Algoritmos de ordenação

4.3.1 Quicksort

Este algoritmo de ordenação é o preferencial quando a relação cabe na memória, ou seja, não são necessários acessos ao disco (que são mais lentos e, por isso, de evitar). A cada execução, o algoritmo selecciona um tuplo e ordena os restantes de modo a que os tuplos inferiores a si estejam do seu lado esquerdo, portanto, menores.

4.3.2 External merge-sort

No caso de a relação ser de tamanho superior à memória e, consequentemente, o algoritmo *Quicksort* não puder ser utilizado, é este que é executado. É composto por duas fases, em que na primeira o algoritmo parte a tabela em partes que caibam em memória, ordena-os e escreve-os num ficheiro temporário; e na segunda fase, o algoritmo combina esses ficheiros temporários num.

4.4 Algoritmos de junção

4.4.1 Nested loop-join (block)

Este algoritmo de junção consiste na análise de cada um dos tuplos na relação de dentro, *inner relation*, para cada tuplo da relação de fora, *outter relation*. Este funcionamento é praticamente igual ao de dois ciclos `for`, um dentro do outro, das linguagens de programação. Se pelo menos uma das relações a serem analisadas couber em memória, essa deve ser a *inner relation* pois assim é reduzido o número de scans a essa relação, o que se torna numa melhoria de performance.

4.4.2 Indexed nested loop-join

Esta variante, que executa pesquisa sobre os índices, apenas é utilizada se o método de junção for equi-join ou natural join e se existir um índice no atributo de junção da relação *inner*. No caso de existirem índices nos atributos de junçãoo de ambas as relações, deve-se utilizar a relação com menos tuplos como a *outter relation*.

4.4.3 Merge-join

Este algoritmo ordena as relações pelo seu atributo de junção e executa uma análise às tabelas para verificar que as condições de junção são mantidas. É bastante eficiente uma vez que cada tuplo é analisado apenas uma só vez e, consequentemente, cada bloco.

4.4.4 Hash-join híbrido

Este algoritmo é usualmente utilizado quando existe bastante memória, embora a relação não caiba necessariamente toda lá. As relações são partidas mas não são escritas em disco, de modo a reduzir o número de acessos de I/O. São então gerados os tuplos de output ao mesmo tempo que os fragmentos das relações são lidos.

4.5 Mecanismos para expressões complexas

Nesta secção são detalhados os mecanismos que o PostgreSQL fornece para que as *queries* mais complexas sejam trabalhadas. Esses mecanismos mais relevantes são a materialização, o pipelining e por fim, a paralelização.

4.5.1 Materialização

Este mecanismo é utilizado sempre que o algoritmo *merge-join* for usado pois este necessita de percorrer os dados do *join* para a ordenação, o que só é possível através da materialização, isto é, criação de vistas. Esta decisão de criação é feita pelo optimizador, da fase de planificação da execução de uma *query*. As vistas criadas, com resultados intermédios, são gravadas em memmória ou no disco, no caso de não caber na memória, sendo utilizadas pelas iterações seuintes. A utilização deste mecanismo providencia mais eficiência pois não é necessário percorrer várias vezes os resultados que vão surgindo.

4.5.2 Pipelining

O sistema PostgreSQL sempre que possível, utiliza pipelining, uma forma dos operadores produxirem tuplos um de cada vez, à medida que são necessários. São depois enviados aos nós inferiores, processados e, caso necessário, reenviados para o ascendente.

Este mecanismo é utilizado pelo executor através de um modelo de iteração que percorre a *query tree*. Este modelo, no caso particular do PostgreSQL possui uma função extra de **rescan** que realiza um reset a um dado plano, inserindo-lhe outros parâmetros. No entanto, nem sempre é possível utilizar o *pipelining* uma vez que existem operações que necessitam de aceder à totalidade dos dados desde o início, como é o caso da ordenação que precisa de conhecer os dados para os poder ordenar da forma mais eficiente possível.

4.5.3 Paralelização

No PostgreSQL tanto do lado do cliente como do servidor existe a noção de paralelismo, isto porque o cliente pode estabelecer mais do que uma conexão com o servidor em modo paralelo, assim como o servidor pois suporta em paralelo os pedidos que recebe. O algoritmo de *Hash-join* pode ser executado em modo paralelo, em que passam a existir vários processadores a realizar simultaneamente diferentes *hashes*.

4.6 Estatísticas e visualização de planos de *queries*

No PostgreSQL e de modo muito semelhante no Oracle 11g são realizados cálculos de custos, sempre baseados em estatísticas. Estes sistemas permitem ainda que o utilizador consulte planos de execução, através do comando `EXPLAIN`, e também realizar uma análise estatística à base de dados, utilizando para tal o comando `ANALYZE`. Estes comandos são seguidamente detalhados.

4.6.1 Analyze

Esta funcionalidade está acessível através do comando

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

São recolhidas estatísticas da base de dados e estas são guardadas internamente, para o planificador as poder utilizar para o auxiliar na elaboração do plano de execução mais eficiente. Caso seja omitido o parâmetro do comando, é feita uma análise completa a todas as tabelas. Embora exista esta possibilidade, o sistema encarrega-se de realizar análises regulares à base de dados, de modo a garantir que quando cria um plano de execuçao os dados que consulta estão sempre actualizados. Esta situação é comum tanto ao PostgreSQL como ao Oracle 11g. Uma vez que é apenas feita uma leitura a tabela que se quer analisar, é possível que a análise corra em paralelo com outras operações, o que não causa diminuição da performance do sistema.

4.6.2 Explain

Esta funcionalidade permite que o utilizador tenha acesso ao plano de execução de forma visível, isto é, apresentado na consola do sistema. Através do comando

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

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

é então possível ter acesso ao plano de execução criado pelo planificador. O plano de execução apresentado é composto pelo número de tuplos e acessos previstos ao disco, a cardinalidade das operações e quais os algoritmos de junção e pesquisa que irão ser usados.

Este comando associado à opção `ANALYZE` faz com que a *query* seja executada, pelo que irá ter custos de execução adicionados ao total retornado. De notar que se estiverem associadas às operações `INSERT, UPDATE, DELERE, CREATE TABLE AS` ou `EXECUTE`, estas vão criar efeitos secundários que ficarão presentes. Para evitar este problema devem utilizar-se os comandos

`BEGIN; EXPLAIN ANALYZE ...; ROLLBACK;`.

4.6.3 Exemplo de Explain

Dada a *query* `EXPLAIN SELECT * FROM foo;` cuja tabela contém apenas um atributo (inteiro) e 10.000 tuplos, o resultado seria

`Seq Scan on foo (cost=0.00..155.00 rows=10000 width=4) (1 row)`.

4.7 Parametrizações possíveis

O PostgreSQL oferece um sistema de parametrização especialmente utilizado quando o plano de execução escolhido pelo optimizador não é o óptimo. É então utilizada esta solução que força que o optimizador a escolher um plano diferente. Esta estratégia está em seguimento com o já referido comando `ANALYZE` pois ambas pretendem que seja utilizado o plano mais eficiente. As possibilidades de parametrização são as seguintes (sempre aplicadas ao planificador da execução da *query*). Apenas são apresentadas as consideradas mais relevantes à análise relizada.

4.7.1 Método planificador

Nestas parametrizações importa referir que o seu valor por omissão é sempre activo (on).

- **enable_bitmapscan (boolean):** Permite activar ou desactivar a utilização de *bitmaps*.
- **enable_hashjoin (boolean):** Activa ou desactiva a utilização do algoritmo *hash-join*.
- **enable_indexscan (boolean):** Activa ou desactiva o uso de pesquisas nos índices.
- **enable_indexonlyscan (boolean):** Permite a activação ou desactivação de *index-only-scan*.
- **enable_material (boolean):** Permite activar ou desactivar a utilização da materialização dos resultados.
- **enable_mergejoin (boolean):** Activa ou desactiva a utilização dos algoritmos *merge-join*.
- **enable_nestloop (boolean):** Activa ou desactiva a utilização dos *nested-loop*. Não sendo possível a sua desactivação total, força a que o planificador não o utilize se existirem outros disponíveis.
- **enable_seqscan (boolean):** Permite activar ou desactivar o uso de scans sequenciais. Novamente, não sendo possível desactivar totalmente, força o planificador a usar outros métodos caso existam.
- **enable_sort (boolean):** Activa ou desactiva a utilização de ordenação. Força o planificador a utilizar outros métodos caso disponíveis.

4.7.2 Custos do planificador

- **seq_page_cost (floating point):** Altera o valor estimado do planificador relativamente ao custo de acesso sequencial ao disco. Por omissão este valor é 1.0.
- **random_page_cost (floating point):** Altera o custo estimado de acesso não sequencial ao disco. Por defeito o valor é de 4.0. Ao reduzir este valor face ao parâmetro `seq_page_cost`, fará o sistema

preferir scans aos índices; seo valor for aumentado, tornará os scans aos índices aparentemente mais caros.

- **cpu_tuple_cost (floating point):** Modifica o valor estimado do custo de processar cada tuplo. O valor por defeito é de 0.01.

- **cpu_index_tuple_cost (floating point):** Permite alterar o custo estimado de processar um operador ou uma função executada durante uma *query*. O valor por omissão é de 0.0025.

4.7.3 Optimizador de *queries* genéticas

- **geqo (boolean):** Permite actiar ou desactivar a optimização genética. Por omissão está activo.

- **geqo_threshold (integer):** Utiliza a optimização genética para planificar *queries* com tantos itens envolvidos como o número de tabelas da cláusula `FROM`. O valor por defeito é 12. Esta parametrização oferece um controlo de mais baixo nível do GEQO - *Genetic Query Optimizer*.

4.7.4 Outros

- **default_statistics_target (integer):** Altera o valor *target* por omissão que não possuam um *target* específico da coluna da tabela. Valores menores aumentam o tempo necessário para realizar o comando `ANALYZE`, no entanto podem melhorar a qualidade das estimativas.

5 Gestão de transacções e controlo de concorrência

No PostgreSQL uma transação é a realização de um conjunto de comandos de uma só vez. Estas transações necessitam de ser geridas assim como controladas por questões de concorrência já que os sistemas de bases de dados permitem o acesso de vários utilizadores em simultâneo. É nestes aspectos que o seguinte capitulo se vai focar, sendo estes: transações de uma forma geral, o seu aninhamento e duração, níveis e protocolos de isolamento, níveis de granularidade, consistência, atomacidade e durabilidade.

5.1 Transações

Em PostgreSQL as transações utilizam uma série de comandos principais de forma a especificar as mesmas. O `BEGIN` e o `END` servem respectivamente para indicar o inicío e fim de uma transição, sendo que o `BEGIN` é implicito se não for incluído na transação. O `COMMIT` (tal como o `BEGIN` é declarado implicitamente) grava permamentemente as alterações feitas até aquele ponto, graças a isto as transações concorrentes passam a ter acesso ao que foi alterado. Antes de se executar um `COMMIT` podem ocorrer erros, na transação ou no sistema, quando isto acontece pode-se recorrer ao comando `ROLLBACK` que reverte todas as operações feitas na mesma transação garantindo assim a segurança e integridade dos dados. Devido ao PostgreSQL não suportar *nested transactions*, é utilizado um sistema equivalente que funciona com base em checkpoints. O comando `SAVEPOINT` marca um ponto de controlo ao qual se pode retornar com o comando `ROLLBACKTO` no caso de algo correr mal. Ao voltar ao ponto marcado pelo `SAVEPOINT` todas as alterações feitas até então são revertidas. Isto permite não ter de fazer um ROLLBACK total, o que dá muito jeito em transações grandes. Pode-se acrescentar que nas bases de dados Oracle 11g é utilizado um sistema semelhante. Por fim, o PostgreSQL para além dos comandos de `SAVEPOINT` e `ROLLBACKTO` que ajudam na execução de transações de longa duração não há outro suporte especial para as mesmas.

5.2 Isolamento

O isolamento no PostgreSQL é realizado através de *multiverson concurrency control* (MVCC) de forma a ter certeza que não existem conflitos entre transações concorrentes. Este mecanismo cria uma *snapshot* da base dados sempre que for realizada uma transação garantindo que os recursos se mantêm consistentes. É possivel criar vários níveis de isolamento com várias versões do mesmo recurso, tal como o nome indica. No entanto isto leva a uma grande ocupação de espaço no sistema o que leva a uma constante limpeza de recursos que não estejam visíveis a nenhuma operação, sendo por vezes invocado o comando *vacuum* (já mencionado anteriormente) implicitamente ou pelo utilizador caso o deseje. Embora este seja o mecanismo usado pelo PostgreSQL também existe suporte para utilização de *locks*, que não é tão vantajoso uma vez

que pode levar a conflitos e *deadlocks*.

5.2.1 Níveis de isolamento

O PostgreSQL apresenta quatro níveis de isolamento, os mesmos do standard SQL, estes são: *Read Commited, Read Uncommited, Repeatable Read e Serializable*. No entanto quando é seleccionado o *Read Uncommited* o que é utilizado na realidade é o *Read Commited*. O utilizador pode indicar o nível de isolamento que pretende das seguintes maneiras:

```
SET TRANSACTION  SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED  READ WRITE | READ ONLY [ NOT ] DEFERRABLE
```

- **Read Commited -** Sendo este o nível usado por omissão no PostgreSQL , o mecanismo usado baseia-se em *snapshots*. É realizada uma *snapshot* da base de dados antes de cada comando e portanto só são visiveis as alterações que tenham sido commited até aquele momento. Se esta operação tentar modificar um recurso que entretanto já foi modificado por outra transação então tem de aguardar que a outra transação termine. No sistema Oracle 11g passa-se o mesmo.

- **Repeatable Read -** A diferença entre o *Read Commited* e o *Repeatable Read* é que no segundo em vez de ser realizada uma *snapshot* antes de cada operação só é feita uma única para toda a *query*. Ou seja, tem um nível de restrição mais elevado o que pode levar a mais conflitos. Os conflitos são resolvidos da mesma forma que no *Read Commited* com a diferença de se outra transação suceder é abortada. Isto só ocorre quando ocorrem operações de escrita, uma vez que as operações de leitura não geram conflitos.

- **Serializable -** Tal como o nome indica o *Serializable* é um nível de isolamento que garante que as transações são executadas em série. Isto indica que é o nível mais restritivo e com maior probabilidade de levar a *rollbacks*. É implementado de forma semelhante ao anterior com a variante de ser feita uma monitorização às condições que podem levar a conflitos. Esta monitorização é feita através de *predicate locking*, que determina se uma escrita vai ter impacto numa leitura realizada por uma transação concorrente. O *predicate locking* não causa bloqueios nas transações e portanto não provoca deadlocks. Esta monitorização acrescenta algum custo às transições, no entanto o uso de *Serializable transactions* é a escolha mais eficiente para alguns casos.

5.3 Locks e níveis de granularidade

Quando a utilização de MVCC não é a mais adequada podesse recorrer a *locks* que não precisam de um isolamento total para controlar os acessos concorrentes. Estes *locks* são adquiridos implicitamente pelas operações podendo também ser adicionados explicitamente e podem ser aplicados a dois níveis de granularidade, a nível das tabelas e a nível dos tuplos. Os *locks* são guardados numa tabela intitulada `pg_locks` e só são libertados depois da transação correspondeste terminar e fazer *commit*. Apesar de haver *locks* incompatíveis entre si é possivel uma tabela ter vários sem que estes criem conflitos e bloqueiem a transação. Os locks podem ser adquiridos explícitamente com o seguinte comando:

```
LOCK [ TABLE ] [ ONLY ] name [ * ] [, ...]  [ IN  ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE
| SHARE UPDATE EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE  MODE
] [ NOWAIT ]
```

Em seguida segue uma lista por ordem de conflito crescente dos *locks* disponíveis e os comandos que os obtêm implicitamente(não é realizada uma descrição de cada um deles pois os seus nomes são auto descritivos):

- **ACCESS SHARE**

  Todas as instruções de SELECT para as tabelas referidas usam este lock. Entra em conflito com o ACCESS EXCLUSVE.

- **ROW SHARE**

  As instruções SELECT FOR UPDATE e SELECT FOR SHARE obtêm este lock. Entra em conflito com EXCLUSIVE e ACCESS EXCLUSIVE.

- **ROW EXCLUSIVE**

  Os comandos UPDATE, DELETE e INSERT obtêm este lock para as tabelas referidas. Entra em conflicto com SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, e ACCESS EXCLUSIVE.

- **SHARE UPDATE EXCLUSIVE**

  Este lock é utilizado para as instruções VACUUM sem FULL, ANALYZE, CREATE INDEX CONCURRENTLY e alguns tipos de ALTER TABLE. Entra em conflito com SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE.

- **SHARE**

  Utilizado no comando CREATE INDEX sem CONCURRENTLY. Entra em concflito vom ROW

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

- **SHARE ROW EXLCUSIVE**

  Este lock não é obtido automaticamente por nenhum comando, portanto tem de ser o utilizador a indicar explicitamente o seu uso. Entra em conflito com ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EX- CLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE.

- **EXCLUSIVE**

  O lock EXCLUSIVE tal como o anterior não é obtido automaticamnete por nenhum comando. Entra em conflito com ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE.

- **ACESS EXCLUSIVE**

  Por fim o ACCESS EXCLUSIVE é utilizado pela sinstruções ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER e VACUUM FULL. Entra em conflito com todos os outros locks, sendo o lock mais restritivo.

5.4 Consistência

A consistência de uma base de dados pode ser garantida através de duas maneiras: utilizando *Serializable* como nível de isolamento porque no caso de haver algum problema de inconsistência uma das transações é abortada (possivelmente a melhor forma de manter a consistência) ou através da utilização de *locks* explicítos. O método de manutenção da consistência a utilizar depende do sistema, por exemplo, no caso de aplicações financeiras deve haver um elevado nível de consistência, noutros sistemas não é tão importante. A juntar a isto existe a hipotése de escolha quanto ao momento de verificação da consistência, que pode ser no fim da transação e ser DEFERRED ou entre oprerações e ser IMMEDIATE. Para selecionar o modo pretendido utiliza-se o comando:

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

5.5 Atomacidade e durabilidade

No PostgreSQL a atomicidade é garantida através do uso das intruções BEGIN, COMMIT e ROLLBACK já explicadas no ínicio deste capitulo. Já para a durabilidade é utilizada a técnica *Write-Ahead Logging* (WAL) que resumidamente permite que o sistema possa ser recuperado em caso de falha. O que acontece é que é mantido um log com as alterações feitas na base de dados de forma a que possam voltar a ser efectuadas caso algo falhe. Este log só é actualizado periodicamente e quando são executados

commits. A utilização deste mecanismo trás um custo associado devido às escritas em disco, por isso são permitidos commits assíncronos que adiam as operações de commit e escrita em disco, no entanto ao ganhar-se performance aumenta também o risco de perder dados caso ocorra uma falha depois do commit ser pedido mas antes de ser executado.

6 Suporte para bases de dados distríbuidas

Para permitir grande disponibilidade e fiabilidade do serviço de base de dados e para diminuir o tempo de resposta, servidores de bases de dados costumam trabalhar em conjunto, ou seja, não existe só uma base de dados centralizada. Numa visão ideal, os servidores de bases de basos deveriam trabalhar juntos sem problemas visiveis, mas isso só funciona em certos casos, como por exemplo bases de dados "*read-only*". Para os restantes casos é muito mais complicado, porque com as escritas em qualquer um dos servidores tem de ser propagado para os outros para que os pedidos de leitura a qualquer servidor tenham respostas com dados consistentes. Por isso o problema da sincronizaçao é a dificuldade fundamental para servidores que em paralelo. Existem múltiplas soluções para resolver esse problema, já que não existe uma solução que consiga eliminar todo o impacto deste problema. Alguns dessas soluções são *Shared Disk Failover*, *File System Replication*, *Transaction Log Shipping* (que é o usado pelo PostgreSQL), entre outros.

De seguida vai-se explicar a solução usada pelo PostgreSQL.

6.1 Log-Shipping

Pode-se criar alta disponibilidade no serviço da base de dados, pode-se configurar um *cluster* de forma a que exista um ou mais servidores *standby* que estão sempre prontos para substituir o servidor primário quando este falhar. Esta capacidade é referida como *warm standby* ou *Log-Shipping*.

Log-Shipping é o processo de transferência de registos *WAL* da base de dados primária para as secundárias. Em PostgreSQL esta funcionalidade está implementada de forma a ser transferido um registo de cada vez e, como os ficheiros são de baixa dimensão (16MB), a sua transferência é feita facilmente e sem grandes custos. Para se suportar esta funcionalidade, a largura de banda requerida varia de acordo com a taxa de transacções a partir do servidor primário.

A movimentação de registos *WAL* é feita de forma assíncrona, ou seja, os registos só são transferidos depois do *commit* das transacções correspondentes. Assim, existe uma janela temporal no qual as transacções que não foram transferidas serão perdidas. O tamanho dessa janela pode ser modificado através do parâmetro `archive_timeout`, mas se for introduzido um valor muito baixo neste parâmetro, a largura de banda necessária para a transferência de ficheiros tem de ser melhor.

Existem dois tipos de *standby servers*:

- *Warm Standby* - solução que replica o *cluster* de bases de dados para um arquivo que pode ser levantado por um servidor *standby* muito rapidamente. Esta variante promove alta disponibilidade.
- *Hot Standby* - solução que oferece replicação assíncrona a um ou mais servidores *standby*. Os servidores também se podem tornar *hot standby* se estes poderem responder a perguntas *read-only*

enquanto estão no modo de recuperação de arquivos ou em *Standby*. Este é útil para fins de replicação e para restaurar *backups* com grande precisão.

Nas subsecções seguintes vão ser explicados vários mecanismos de replicação.

6.1.1 Streaming replication

Este tipo de replicação permite que o servidor *standby* fique mais actualizado do que com *log shipping* devido ao *stream* de registos *WAL* assim que são gerados do servidor primário para o *standby*, sem esperar que o ficheiro *WAL* seja preenchido.

A replicação por *stream* é feita de forma assíncrona por omissão, em que neste caso faz com que haja um certo *delay* entre o *commiting* da transacção no servidor primário e a visualização das alterações nos *standby* (embora este *delay* seja menor que o *log shipping*.

Ao longo da execução deste estilo de replicação, esta pode ser monitorizada para se poder ver o estado de eficácia da replicação. A eficácia pode ser avaliada através da verificação do número de registos *WAL* gerados no servidor primário mas que ainda não foram aplicados nos *standby*.

6.1.2 Cascading Replication

Esta variante de replicação permite que um servidor *standby* receba conexões de replicação e que faça *stream* dos registos *WAL* para outros *Standbys*, comportando-se como um retransmissor. Esta técnica faz com que seja reduzido o número de conexões directas ao servidor primário e diminui o *overhead* da banda larga entre sites.

Um servidor *standby* que se comporta como um *receiver* e um *sender* é chamado de *cascading standby* e tem dois tipos adjacentes: se se ligar mais directamente ao servidor primário é chamado de servidor *upstream*, se se ligar mais a outros *standbys* é chamado de servidor *downstream*.

Neste mecanismo não impõe limite de ligações *downstream*, mas cada *standby* só se liga a um *upstream* que eventualmente estará ligado ao servidor primário. Assim, se um ligação com um *upstream* terminar, a replicação por *stream* continua pelo *downstream* enquanto houver registos *WAL* disponíveis.

Este tipo de replicação é do tipo assíncrono.

6.1.3 Synchronous Replication

Por omissão, a replicação por *streaming* é feita assíncronamente, fazendo com que, se o servidor primário falhar, algumas transacções que tenham sido *commited* não sejam replicadas para os servidores *standby*, levando a perda de dados.

A replicação síncrona oferece a habilidade de confirmar todas as mudanças feitas numa transacção tenham sido transferidas para um servidor *standby* síncrono.

Nesta variante, cada *commit* de uma transacção de escrita vai ter de esperar até à confirmação de que o *commit* foi escrito na *transaction log* no servidor primário e no servidor *standby*. Isto acrescenta mais segurança e durabilidade dos dados mas aumenta o tempo de resposta para uma transação. As transações de leitura ou *rollbacks* não precisam de esperar pelas respostas dos servidores *standby*.

Na replicação síncrona, normalmente é requerido que os servidores *standby* sejam cuidadosamente planeados e colocados de forma a garantir que as aplicações dependentes tenham desempenho aceitável. Falta de cuidado no uso desta variante faz com que a *performance* das aplicações seja reduzida devido aos tempos de resposta mais elevados e a uma maior contenção. O PostgreSQL permite que esta variante possa ser aplicada a todo o sistema ou só a determinados utilizadores, conexões ou transacções.

6.2 Alternativas

Sendo o sistema de base de dados PostgreSQL um sistema de código aberto, houve equipas que pegaram no código e criaram alternativas *third-party* que respondem melhor às necessidades de uso do PostgreSQL de forma distríbuida.

De seguida são apresentados alguns exemplos de alternativas *third-party*, dando principal destaque à forma de replicação e tipo de sincronização usada em cada um.

Programa	Tipo de replicação	Tipo de Sincronização
PgCluster	*Master - Master*	Síncrono
pgpool-I	*Statement-Based Middleware*	Síncrono
pgpool-II	*Statement-Based Middleware*	Síncrono
slony	*Master - Slave*	Assíncrono
Bucardo	*Master - Master; Master - Slave*	Assíncrono
Londiste	*Master - Slave*	Assíncrono
Mammoth	*Master - Slave*	Assíncrono
rubygrep	*Master - Master; Master - Slave*	Assíncrono

Tabela 3: *Alternativas à distribuição do* PostgreSQL

Para efeitos de explicação, o *Statement-Based Middleware* é uma forma de replicação que usa um programa que intercepta todas as perguntas *SQL* e envia-as para um ou todos os servidores (em que cada servidor funciona independentemente). Os *Read-Writes* são enviados para todos os servidores, enquanto

que os *Read-Only* pode ser enviado para um só servidor, distribuindo assim o *workload* da leitura.

6.3 Ligação entre bases de dados

O PostgreSQL providencia um módulo adicional chamado `dblink` para se poder abrir conexões persistentes de forma segura (ou não) entre vários servidores. Também fornece funções para executar *queries* e operações remotas, aberturas e fechos de cursores, obtenção de informações da conexão, entre outros. Este módulo só pode ser usado entre instâncias de sistemas de gestão de base de dados baseados em PostgreSQL.

6.4 Comparação com o Oracle 11g

Como só nas versões mais recentes do PostgreSQL é que foi introduzido o suporte para sistemas distribuídos, este suporte ainda é um bocado limitado, comparando com o suporte oferecido no Oracle 11g. O Oracle 11g também usa replicação de dados da mesma forma que o PostgreSQL, ou seja, permite replicação síncrona, assíncrona e em forma de *stream* entre vários servidores primários ("*Multimaster Replication*").

O Oracle 11g também possibilita o uso de SQL distribuido, que basicamente é o uso de SQL normal só que afecta vários servidores (Oracle e non-Oracle) mas é invisível essa interacção para o utilizador.

7 Outras características do *PostgreSQL*

Nesta secção serão apresentadas algumas características que o PostgreSQL fornece aos seus utilizadores. São elas suporte ao XML, linguagens procedimentais, triggers e modos de autenticação e segurança.

7.1 Suporte de XML

O PostgreSQL suporta documentos na linguagem XML, que permitem que o armazenamento de dados seja feito de forma estruturada. Para poder utilizar esta funcionalidade é necessário instalá-la. Através do comando

`configure -with-libxml`

o sistema instala os módulos respeitantes ao XML.

Para a criação de um documento XML, a partir de uma *string*, recorre-se ao comando

`XMLPARSE ( DOCUMENT | CONTENT value)`,

e para a criação de uma *string* partindo de um documento XML, é utilizado o comando

`XMLSERIALIZE ( DOCUMENT | CONTENT value AS type` ,

com `type` a poder ser do tipo `character, character varying`, ou `text`.

7.2 Linguagens procedimentais

As linguagens procedimentais suportadas pelo PostgreSQL são PL/pgSQL, PL/Tcl, PL/Perl e PL/Python, entre outras. Permite ainda que sejam criados *triggers* através destas linguagens e até mesmo em C. Embora as suporte, não é o PostgreSQL que as trata: recorre a um *handler* próprio para o efeito.

7.3 Autenticação e Segurança

O processo de autenticação é controlado pelo PostgreSQL através do ficheiro de configuração `pg_hba.conf`, guardado na base de dados. O conteúdo deste ficheiro é construído à base de records, um por linha, que especifica o tipo de conexão, uma gama de endereços IP do cliente, o nome da base de dados, um username e o método de autenticação.

O parâmtro método de autenticação é aquele que está sujeito a mais possibilidades. Este pode ser do tipo:

- **trust** permite que qualquer utilizador possa realizar login como sendo qualquer utilizador.
- **reject** permite especificar, por exemplo, um host ao qual não se pretende atribuir a possibilidade de se ligar à base de dados.

- **md5** requer que o cliente forneça uma password com encriptação do tipo MD-5.
- **password** requer que o utilizador apenas forneça uma password, não sendo necessário que esteja encriptada.
- **gss** utiliza o método de autenticação GSSAPI para autenticar o utilizador.
- **sspi** utiliza o método SSPI para autenticar o utilizador (apenas disponível no sistema Windows).
- **krb5** usa o sistema de autenticação Kerberos v5 para autenticar o utilizador.
- **ident/peer** obtém o username do sistema operativo do utilizador.
- **ldap** permite autenticação através de um servidor LDAP.
- **radius** autenticação através de um servidor RADIUS.
- **cert** autenticação através de certificados SSL.
- **pam** autentica utilizando para tal p *Pluggabçe Authentication Module* fornecido pelo sistema operatico.

Referências

[1] *PostgreSQL documentation 9.3*. http://www.postgresql.org/docs/9.3/interactive/index.html, 2014.

[2] FinTheBest *Comparison PostgrSQL vs Oracle 11g*. http://database-management-systems.findthebest.com/compare/36-43/Oracle-vs-PostgreSQL

[3] Silberschatz, Abraham, Korth, Henry F., Sudarshan, S. *Database System Concepts*. McGraw-Hill, 6th edition, 2010, ISBN: 78-0-07-352332-3.

[4] *How PostgreSQL Executes a Query*. http://etutorials.org/SQL/Postgresql/Part+I+General+PostgreSQL+Use/Chapter+4.+Performance/Understanding+How+PostgreSQL+Executes+a+Query/

[5] Momjian, Bruce *Explaining the Postgres Query Optimizer*. http://momjian.us/main/writings/pgsql/optimizer.pdf, 2012.

[6] Momjian, Bruce *PostgreSQL Internals Through Pictures*. http://momjian.us/main/writings/pgsql/internalpics.pdf, 2012.

[7] Lane, Tom *A Tour of PostgreSQL Internals*. http://www.postgresql.org/files/developer/tour.pdf, 2000.

[8] *PostgreSQL Wiki*. http://www.postgresql.org/files/developer/tour.pdf, 2014.

[9] *Wikipedia – PostgreSQL*. http://en.wikipedia.org/wiki/PostgreSQL, 2014.

[10] *Relational database management systems*. http://db.cs.berkeley.edu/papers/UCB-MS-zfong.pdf,

[11] Conway, Neil *Query Execution Techniques in PostgreSQL*. http://www.neilconway.org/talks/executor.pdf, 2007.

[12] Oracle *Memory Architecture*. http://docs.oracle.com/cd/B28359_01/server.111/b28318/memory.htm#CNCPT007

[13] Smith, Greg *In and Out of the PostgreSQL Shared Buffer Cache*. http://www.pgcon.org/2010/schedule/attachments/156_InOutBufferCache.pdf, 2010.

[14] Oracle *Master Replication Concepts and Architecture*. http://docs.oracle.com/cd/E11882_01/server.112/e10706/repmaster.htm#REPLN201