Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Mestrado Integrado em Engenharia Informática

Sistemas de Base de Dados

Análise do PostgreSQL 9.3

Autores:
Grupo 18

Daniel Magro - nº 41725
Frederico Alves - nº 41713
João Mateus - nº 41801

Docente:
José Júlio Alferes

1 de Junho de 2014

Conteúdo

Capítulo 1

Introdução

1.1 Motivação

O presente relatório foi elaborado no contexto da cadeira de Sistemas de Base de Dados, como trabaho final, e tem como tema principal o estudo de uma sistema de gestão de base de dados, segundo os tópicos abordados nas aulas teóricas. Estes tópicos cobrem temas como o armazenamento e estrutura de ficheiros, indexação, processamento e otimização de perguntas, gestão de transações e controlo de concorrência e ainda o suporte de base de dados distribuídas.

O sistema de gestão de base de dados, ou SGBD, escolhido foi o PostgresSQL. Esta escolha deve-se principalmente a este sistema ser *open source*, ter uma documentação online completa e bastante acessível e ainda temos o facto de estarmos a lidar com um sistema multi-plataforma, que tem tido uma boa adesão da comunidade por ser uma sistema robusto.

Este relatório é composto por sete capítulos, sendo o presente o primeiro. No segundo capítulo abordamos o armazenamento e estrutura de ficheiros utilizados, onde falamos do sistema de ficheiros utilizado, na organização em termos de tabelas e tuplos, partições, do sistema de *buffers* e outros assuntos relacionados. O terceiro capítulo aborda os tipos de índices suportados pelo sistema de gestão de base de dados, os comandos que permitem a utilização desses índices e outros aspetos relacionados com índices. No quarto capítulo falamos de todo o processo de processamento e otimização de perguntas, abordando assuntos como a construção e escolha dos planos que o sistema executa para obter os resultados das perguntas. A gestão de transações e controlo de concorrência é abordada no capítulo cinco, em que falamos de como são tratadas as transações e as questões que advêm da utilização de concorrência num sistema destes. O capítulo seis faz um breve passagem sobre o tema de base de dados distribuídas e o seu suporte no sistema. No capítulo sete damos a conhecer algumas particularidades do PostgreSQL. É de salientar que ao longo do relatório são feitas comparações entre o PostgreSQL e o Oracle 11g.

Na secção seguinte fazemos uma breve introdução à história do PostgreSQL.

1.2 Breve História do PostgresSQL

O PostgreSQL tem origens no projeto Ingres de Berkeley, Universidade da California, em 1982, que era liderado pelo professor Michael Stonebraker. Este abandonou Berkeley de forma a tentar comercializar o sistema de gestão de base de dados Ingres, que tinha tido começado por ser apenas um projeto académico.

Em 1985 Michael Stonebraker volta a Berkeley, com o objetivo de começar um projeto pós-Ingres que tratasse do problema do suporte e definição de tipos, mais precisamente suporte de objetos.

O projeto Postgres tem início em 1986, com uma implementação dos conceitos base do sistema.

Várias versões seguem-se, com a inclusão redefinição de alguns conceitos. É criada um versão comercializada do Postgres, o Ilustra.

Até que em 1995, Andrew Yu e Jollu Chen, na altura alunos alunos de Michael Stonebraker, entram no projeto e substituem o interpretador POSTQUEL por um interpretador SQL. Tendo o projeto sido renomeado para Postgres95.

Em 1996, devido a dimensão que o projeto tomou, o Postgres95 passou a ser um projeto *open source* e foi renomeado para PostgreSQL, mesmo que ainda hoje se continue a referir a este sistema com Postgres. A primeira versão do PostgreSQL é lançada no ano seguinte, sendo esta a versão 6.0, tendo como base o número de versões desde a primeira versão do Postgres.

A versão mais atual deste sistema é o PostgreSQL 9.4 Beta 1, tendo sido anunciada a 15 de Maio de 2014.

Hoje o sistema é vastamente utilizado, sendo suportado maioritariamente pela comunidade. Este sistema tornou-se de tal forma popular, que, mesmo sendo um projeto *open source*, é utilizado por empresas de grande dimensão mundial, como a Yahoo!, Skype, Sony Online e alguns sites e redes sociais muito populares como o Reddit, o MySpace e o Instagram. Também é de salientar a oferta do PostgreSQL como serviço por empresas como a Amazon.com e o Heroku.

Capítulo 2

Armazenamento e Sistema de Ficheiros

Neste capítulo abordamos a forma como o PostgreSQL armazena os ficheiros e a forma como está organizada o seu sistema de ficheiros. O sistema que o PostgreSQL utiliza é o sistema de ficheiros do sistema operativo em que está instalado, sendo *PGDATA* o nome para o seu *cluster* de dados, ou seja, a diretoria onde guarda todos os seus dados. Sendo a organização de ficheiros feita por subdiretorias, a diretoria *PGDATA* contém várias subdiretorias, estando essas listadas abaixo.

PG VERSION Ficheiro onde está o número da versão do PostgreSQL;

base Subdiretoria onde estão as subdiretoria de cada base de dados;

global Subdiretoria onde se encontram as tabelas comuns a todos os *clusters*;

pg_clog Subdiretoria onde está a informação sobre o estado dos *commits* das transações;

pg_multixact Subdiretoria onde está a informação sobre o estado das multi-transações;

pg_notify Subdiretoria onde está a informação sobre o sistema de notificações;

pg_serial Subdiretoria onde está a informação sobre transações serializáveis que foram *commited*

pg_snapshots Subdiretoria onde estão *snapshots* exportados;

pg_stat_tmp Subdiretoria onde estão ficheiros temporários para o subsistema de estatística;

pg_subtrans Subdiretoria onde estão dados sobre o estado de subtransações;

pg_tblspc Subdiretoria onde estão *symbolic links* para *tablespaces*;

pg_twophase Subdiretoria onde estão ficheiros sobre o estado de transações preparadas;

px_xlog Subdiretoria onde estão os ficheiros de registo WAL;

postmaster.opts Ficheiro que contém as configurações da última inicialização do servidor;

postmaster.pid Um ficheiro que contém o PID do *post master* corrente, o ID da memória partilhada, *timestamp* de quando foi inicializado o *postmaster* e o *path* para o *data cluster*.

Podem existir vários *clusters* numa só máquina, pelo que para que para cada base de dados no *cluster* existe uma subdiretoria em *PGDATA/base*, com o nome do OID (*Object ID*) da base de dados, sendo essa subdiretoria *default* para os ficheiros da mesma. Quanto às tabelas e aos índices, esses são guardados em ficheiros separados, tendo os ficheiros das tabelas o nome da respetiva tabela e os ficheiros dos índices o nome do número do *filenode* do índice, que pode ser encontrados no ficheiro. No caso em que as relações são temporárias, o nome do ficheiro tem a forma *tBBB_FFF*, onde *BBB* é o identificador de onde foi feita a criação do ficheiro e *FFF* o número do *filenode*. As tabelas e índices têm um tamanho, por omissão, de 1 GB, sendo esse limite ajustável. Quando uma tabela ou índice excedem o tamanho de 1 GB, estes são divididos em segmentos de 1 GB. O primeiro segmento detém o nome do *filenode* respetivo, sendo os subsequentes segmentos nomeados *filenode.1, filenode.2*, etc. Esta esta estratégia tem como objetivo evitar problemas com as limitações de tamanho de ficheiros que o sistema operativo possa ter. Todas as tabelas e índices têm também um *free space map*, que contém a informação sobre o espaço disponível, e também um *visibility map*, onde está a informação sobre quais as páginas que têm apenas tuplos visíveis.

2.1 Free Space Map

Cada relação de *heap* e de índice, com a exceção das relações de índice *hash*, possuem um *freespace map* que mantém a informação sobre o espaço livre da relação. O mapa é organizado como uma árvore de páginas FSM, em que as folhas guardam o espaço livre de cada relação, usando um *byte* para representar cada página. Em cada nível superior está agregada a informação do nível inferior. Cada uma das páginas FSM contém uma árvore binária, guardada num *array* com um *byte* por nó. Cada nó representa uma página *heap*, ou uma página FSM de nível inferior. Em cada nó que tenha filhos, é guardado o maior valor dos mesmos, assim, o maior valor das folhas é guardado na raiz.

2.2 Visibility Map

Cada relação de *heap* tem um *Visibility Map* (VM) que mantém o registo de quais são os tuplos que são visíveis a todas as transações. Os índices não têm VMs. Estes mapas guardam um *bit* por página *heap*. Um *bit* assinalado significa que todos os tuplos da tabela são visíveis a todas as transações, e caso isso aconteça, então a página não contém tuplos que precisam de ser limpos. Os mapas são conservativos no sentido em que se certificam de que se um *bit* está assinalado, a condição acima é verdadeira, caso contrário, a condição não é necessariamente falsa, apenas não sabemos se é verdadeira.

2.3 Técnica TOAST

O PostgreSQL usa páginas de tamanho fixo e não permite que os tuplos sejam expandidos por outras páginas, pelo que não é possível guardar valores muito grandes diretamente. Para se resolver este problema, os dados são comprimidos ou divididos em múltiplos registos, de maneira transparente ao utilizador. A técnica TOAST (The Oversized-Attribute Storage Technique) consiste nesta solução.

Existem quatro estratégias para se guardar entradas TOAST:

PLAIN Previne a compressão e a decomposição, sendo a única possível estratégia para entradas que não necessitem de TOAST;

EXTENDED Permite compressão e a decomposição, sendo a estratégia usada por omissão. Primeiro executa-se a compressão, e depois, se a entrada permanecer demasiado grande, a decomposição;

EXTERNAL Permite a decomposição, mas não a compressão. Esta estratégia otimiza a manipulação dos dados, sendo que tem a desvantagem de ocupar mais espaço de armazenamento;

MAIN Permite compressão e decomposição, sendo que a decomposição ocorre apenas como último recurso, quando não existe nenhuma maneira de fazer a entrada caber na página.

2.4 Slotted Pages

Para armazenar as tabelas e os índices são utilizadas as *slotted pages* de modo a permitir registos de tamanho variável. Todas as tabelas e índices são guardadas em *arrays* de um tamanho fixo (normalmente 8 kB). No caso das tabelas, um registo pode ser guardado em qualquer uma das páginas, porque estas são equivalentes. O mesmo não se sucede com os índices em que a primeira página normalmente é reservada como uma *metapage*, que servirá para informação de controlo. Também pode haver diferentes tipos de página para o índice, dependendo do tipo de acesso ao mesmo. Os conteúdos de uma *slotted page* encontram-se listados abaixo.

PageHeaderData 24 *bytes* que contêm a informação geral sobre a página, incluindo apontadores de espaço livre;

ItemIdData *Array* de pares (*offset*, *length*) que apontam para os dados;

Free Space Espaço livre;

Items Registos que estão guardados na página;

Special Space Espaço reservado para acessos por índices. Vazio em tabelas normais.

2.5 Partição de Tabelas

O PostgreSQL suporta a partição básica de tabelas, que é a divisão de uma tabela em várias tabelas mais pequenas, passando a existir uma representação lógica da tabela completa e as sub-tabelas onde ficam armazenados os dados. A partição de tabelas tem algumas vantagens em termos de performance das perguntas a uma tabela. Nos casos em que apenas uma partição é acedida, ou no caso em que se quer remover uma grande parte da tabela, poderá bastar remover algumas partições. Tabelas de muito grandes dimensões beneficiam mais das vantagens da partição de tabelas, sendo que, nos outros casos não compensa tanto, até podendo chegar a ser prejudicial em termos de performance dado o aumento da complexidade da representação da tabela. É possível fazer o particionamento das tabelas das seguintes formas:

Range Partitioning O particionamento é feito através de intervalos definidos por uma coluna chave ou um conjunto de colunas, sem sobreposição de intervalos;

List Partitioning O particionamento é feito utilizando uma lista onde se colocam os valores que se querem em cada partição.

Para criar uma tabela particionada é necessário fazer o seguinte:

1. Criar uma tabela principal, pai, da qual as partições irão receber as propriedades por herança, propriedades como restrições de tabela;
2. Criar várias tabelas filhas, que serão as partições. Estas herdam da tabela pai;
3. Adicionar restrições às partições que definam quais serão os valores chave permitidos em cada partição;
4. Para cada partição criar um índice nas colunas chave;
5. Assegurar que o parâmetro de configuração *constraint exclusion* está ativo no ficheiro *postgresql.conf*, caso contrário as perguntas não serão otimizadas como desejado.

Podemos também, opcionalmente, definir *triggers* ou regras para redirecionar os dados inseridos na tabela pai para a partição correta.

2.6 Gestão de *Buffers*

O PostgreSQL implementa o seu próprio sistema de gestão de *buffers* que, tendo em conta que o sistema de ficheiros que utiliza é o do sistema operativo, pode entrar em conflito com o sistema de gestão de *buffers* do sistema operativo onde está a correr. Cada entrada do *buffer* aponta para blocos de 8 kB onde se encontra uma etiqueta que identifica o ficheiro que a entrada guarda. Associados aos blocos existem *flags* que servem para indicar o estado em que se encontra o bloco. As *flags* podem ter o valor *pinned*, que indica que o conteúdo do *buffer* está a ser usado por um processo e não permite o acesso a qualquer outro processo, ou *dirty*, que serve para indicar se o conteúdo foi modificado

desde a sua leitura de disco, o que é útil para ter informação sobre quais as páginas a atualizar. Os blocos que são guardados no *buffer* são aqueles que correspondem às páginas com o maior número de acesso, sendo que, caso já não exista espaço para inserir um novo bloco, é utilizada a estratégia *Last Recently Used*, que remove os blocos que são menos utilizados. Para apagar dados que já não existam na base de dados, mas que persistam no *buffer*, o sistema utiliza a operação *vacuum*, servindo neste caso como *garbage collector*.

2.7 Clustering

O PostgreSQL suporta *clustering* de tabelas de acordo com um determinado índice, sendo que o índice tem de estar definido na tabela antes de se proceder à operação de *cluster*. Quando uma tabela é *clustered*, esta é reorganizada em disco com base no índice. A operação de *cluster* apenas afeta os dados até ao momento em que foi feita a operação, ou seja, caso tenhamos uma tabela que foi *clustered* e foi feita uma atualização à tabela, os novos dados não estarão *clustered*. Isto acontece porque os novos tuplos não são guardados tendo em conta o seu índice, sendo necessário realizar outra operação de *cluster* para que a tabela seja toda reorganizada. Quando uma tabela está a ser reorganizada, está é adquirida através de um *access exclusive lock* para prevenir quaisquer acessos à tabela até que a operação tenha terminado. Em perguntas nas quais seria necessário efetuar uma junção de tabelas, a estratégia de *multitable clustering* traria benefícios em termos de desempenho pois não seria necessário efetuar a junção. Esta é uma estratégia que não se encontra implementada no PostgreSQL.

Segue-se um exemplo de utilização:

```
CLUSTER students USING students_index;
```

2.8 Comparação com o Oracle

Comparando os dois sistemas, em termos de sistema de ficheiros, chegámos à conclusão de que os sistemas utilizam diferentes estratégias, sendo que o PostgreSQL se baseia no sistema de ficheiros do sistema operativo onde se encontra instalado, contrariamente ao Oracle, que implementa o seu próprio sistema de ficheiros. Embora o sistema de ficheiros do Oracle tenha uma maior complexidade, este ganha e nível de desempenho, o que no PostgreSQL, está dependente do sistema operativo onde se encontra instalado.

Quanto ás tabelas e tuplos, ambos os sistemas fazem a sua organização por *heaps*. Em termos de particionamento de tabelas, podemos observar que ambos permitem o particionamento por intervalos e por listas, estando a diferença no Oracle, que permite mais dois tipos de particionamento que não se encontram presentes no PostgreSQL, sendo estes o *Hash Partitioning* e o *Composite Partitioning*.

Quanto à gestão de *buffers*, ambos têm o seu próprio sistema de gestão, ambos utilizam a técnica LRU, visando o menor número de acessos a disco, estando ambos os sistemas mais apropriados a bases de dados de médias a largas dimensões.

Por fim, comparando as estratégias de *clustering* dos dois sistemas, chegámos à conclusão de que ambos implementam o *clustering* de tabelas, com a diferença de que o Oracle permite o *multitable clustering*, estratégia que pode melhorar o desempenho em consultas que se tenha de fazer junção de tabelas.

Capítulo 3

Indexação e Hashing

Um índice é uma estrutura de dados que pode servir para pesquisar uma base de dados de uma forma mais eficiente, sendo que estes aliam um predicado comum a um determinado número de tuplos numa base de dados. Com estas estruturas é possível retornar um número reduzido de tuplos de uma pesquisa numa base de dados de grandes dimensões, com milhares ou até milhões de tuplos, de uma forma rápida. Caso estes não fossem utilizados estaríamos condenados a fazer pesquisas totais às tabelas das base de dados, o que para o caso de o resultado ser de uma dimensão muito menor do que o número de tuplos numa tabela seria muito ineficiente fazer um pergunta, ou seja, se numa tabela com um milhão de tuplos fizermos uma pergunta em que o resultado é apenas um dezena de tuplos, então teríamos de percorrer toda a tabela à procura destes dez tuplos, o que se revelaria uma tarefa bastante dispendiosa não só em termos de operações *input/output*, mas também em termos temporais.

A utilização de índices numa base de dados pode realmente ser uma fonte de eficiência, mas se estes forem mal utilizados ou se deixarem de ser necessários, podem tornar-se um fardo. Por isso, é sempre necessário definir muito bem que índices são realmente necessários para uma base de dados.

A criação de um índice no PostgreSQL pode fazer-se simplesmente com o comando:

```
CREATE INDEX index_name ON table_name (column);
```

Este comando vai criar um índice de um tipo pré-definido. Para criar um índice de uma forma mais completa, escolhendo várias parametrizações, podemos utilizar a seguinte sintaxe:

```
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 ]
```

Para remover um índice, basta utilizar o comando:

```
DROP INDEX index_name;
```

Como a criação de um índice referente a uma tabela muito grande pode demorar muito tempo, o PostgreSQL só permite leituras das tabelas em paralelo com esta criação. Já as operações de escrita são bloqueadas e deferidas para o fim da criação do índice. É de salientar que é possível configurar este comportamento, mas é necessário ter algumas coisas em conta.

Após a criação de um índice, o sistema fica responsável pela sua utilização e atualização, ou seja, tem de manter o índice coerente com a tabela. Contudo isto pode ser uma fonte de ineficiência, o que nos remete para o que foi dito a cima, querendo isto dizer que índices que não são utilizados devem ser removidos.

Nas seguintes secções são descritos os tipos de índices suportados pelo PostgreSQL, sendo sempre que possível feita uma pequena comparação com o Oracle 11g, e ainda falamos de índices multi-coluna, combinação de índices, índices parciais e outros.

3.1 Índice B-Tree

As árvores B são o índice predefinido na criação de índices no PostgreSQL. O B é uma abreviatura para *Balanced*, já que a ideia principal é ter o mesmo número de dados em cada lado da árvore, o que leva a que a árvore criada tenha um número baixo de níveis e assim o tempo de percorrer a árvore tende a ser reduzido, consoante o tamanho da árvore. Este tipo de índice são bastante eficientes para operadores de comparação como `<, <=, =, >=, >`, sendo que o sistema de criação de planos irá ter sempre em conta este tipo de índice sempre que um destes operadores estiver a ser usado numa comparação. Deve-se também salientar que operações equivalentes às acima mencionadas, como o BETWEEN e o IN, também podem ser utilizadas neste tipo de índice. Uma das características mais apelativas deste tipo de índice é a possibilidade de utilização de valores *NULL*. O otimizador poderá também utilizar este tipo de índice se houver perguntas com as operações LIKE e `~`, de forma a comparar um certo padrão, mas só se este estiver no início da *string* do padrão e este for constante.

É apresentado um exemplo da utilização deste tipo de índice na secção de exemplos deste capítulo.

O sistema de gestão de base de dados Oracle 11g também suporta índices B-tree.

3.2 Índice de Hash

Os índices de *hash* apenas utilizam operadores de igualdade, sendo assim o otimizador de planos apenas vai ter em conta este tipo de índice no caso de uma comparação de igualdade. Estes são os únicos índices no PostgreSQL que não suportam recuperação de falhas, por exemplo, se houver uma falha no sistema e houve alterações que não chegaram a ser escritas, então o índice terá de ser reconstruido. Isto deve-se ao facto de as operações deste tipo de índice não serem WAL-*logged*. Para além de não suportar falhas, estes também não fazem replicação de dados.

A implementação deste índice baseia-se no *hashing linear* de Witold Litwin. Esta implementação permite a expansão de um tabela de *hash* uma *slot* de cada vez, sendo que assim o custo da expansão será repartido pelas várias inserções

na tabela em vez de ser contabilizado todo de uma vez. Isto permite que haja *rehash* mais pequenos e por isso com menor custo.

Com isto a utilização de um índice de Hash, em alguns casos, não compensará tanto no caso de se poder utilizar um índice B-tree.

Apresentamos na secção de exemplos, um exemplo da criação de um índice de Hash que é usado escolhido, e por isso compensa a sua criação.

O sistema de gestão de base de dados Oracle 11g também suporta índices de *hash*.

3.3 Índice GiST

Índices GiST ou *Generalized Search Tree*, este tipo de índice permite construir árvores balanceadas. Este não é um único índice, mas sim uma estrutura que pode ser implementada de várias formas, como B-tree e R-trees.

Estes índices conseguem usar operações de comparação, mas também podem utilizar operadores para tipos geométricos bidimensionais de dados: `<<`, `&<`, `&>`, `>>`, `<<|`, `&<|`, `|&>`, `|>>`, `@>`, `<@`, `~=`, `&&`.

Com esta capacidade usar tipos geométricos bidimensionais de dados, este tipo de índice consegue otimizar procuras do tipo *nearest-neighbor*. Isto pode ser observado, no exemplo dado na documentação em que se tenta encontrar os dez pontos mais perto de um certo ponto:

```
SELECT * FROM places ORDER BY location <-> point '
    (101,456)' LIMIT 10;
```

Também é de salientar que estes também são bons para pesquisa do tipo *full-text*.

Aqui está um exemplo da criação de um índice deste tipo:

```
CREATE INDEX index_name ON table_name USING gist (
    column);
```

É ainda apresentado um exemplo da utilização deste índice na secção de exemplos deste capítulo.

Por fim, é de referir que este tipo de índice não está presente no SGBD Oracle 11g.

3.4 Índice GIN

Os índices GIN, ou Generalized Inverted Indexes, tem a capacidades de suportar casos em que os valores a serem indexados são valores compostos e os valores a serem pesquisados encontram-se em valores compostos. Na documentação referem um exemplo bastante simples para compreender que valores são trabalhados neste tipo de índice, em que os valores compostos são documentos e as perguntas referem-se a à pesquisa de palavras nestes documentos. Como os índices GiST, este tipo de índice suporta vários tipos de indexação e vários tipos de operadores, como por exemplo: `<@`, `@>`, `=`, `&&`.

A criação de um índice GIN pode ser feita da seguinte forma:

```
CREATE INDEX index_name ON table_name USING gin (
    column);
```

Uma das vantagens deste tipo de índice é o facto de este permitir o desenvolvimento de tipos de dados específicos, ou seja, o índice não sabe que operação vai acelerar. Este aspeto também é semelhante aos índices GiST. É de salientar que o objetivo principal dos índices GIN é o suporte de pesquisas *full text* altamente escaláveis.

A implementação deste índice é mantida por Teodor Sigaev e Oleg Bartunov.

Este tipo de índice também é suportado pelo Oracle 11g.

3.5 Índices multi-coluna

Os índices multi-coluna são índices que são definidos em mais do que uma coluna de uma tabela. Atualmente, a utilização deste tipo de índice está restrita ao uso de índices B-tree, GiST e GIN, e o número máximo de colunas que podem ser especificadas é 32.

A utilização de um índice multi-coluna com um índice B-tree pode ser usado em qualquer sub-conjunto de colunas das colunas definidas para esse índice. Contudo, para que a utilização deste índice seja mais eficiente, são impostas restrições sobre as colunas mais à esquerda. A regra enunciada na documentação diz que devem ser aplicadas restrições de igualdade nas primeiras colunas pelo índice. Para além disto, todas as restrições de desigualdade na primeira coluna, que não tem restrições de igualdade, vão ser usadas para limitar o conjunto de tuplos pesquisados pelo índice. Já restrições nas colunas à direita destas colunas vão ser pesquisadas no índice, de forma a poupar visitas à tabela. No entanto, isto não limita o conjunto de tuplos que tem de ser pesquisados pelo índice. Na documentação dão um exemplo de um índice com as colunas `(a,b,c )` e condição de uma pergunta é `WHERE a = 5 AND b >= 42 AND c < 77`, este índice vai ter de ser analisado desde o primeiro tuplo com `a = 5 e b = 42` até ao último com `a = 5`. Já tuplos do índice com `c > 77` seriam passados a frente, mas teriam na mesma de ser analisados. É de salientar que também é referido na documentação que este índice poderia ser utilizado em perguntas que contêm restrições em b e c, ou só num deles, mas sem qualquer restrição na coluna a, no entanto todo o índice teria de ser analisado, portanto o sistema de geração de planos iria preferir uma análise sequencial à tabela.

No caso da utilização de índices multi-coluna do tipo GiST, este pode ser usado com qualquer tipo de condições de perguntas que envolvam um qualquer sub-conjunto das colunas do índice. É de salientar que são as restrições na primeira coluna que vão determinar quantos tuplos do índice serão analisados, mas no que diz respeito à restrição de resultados, são as restrições nas outras colunas que vão condicionar os tais resultados. Também se deve mencionar o facto de que com índices GiST teremos alguma ineficiência, se a primeira coluna tiver poucos valores distintos, mesmo que as outras tenham valores muito variados.

Com índice multi-coluna do tipo GIN, podemos utilizar utilizar qualquer tipo de condições na pergunta, envolvendo qualquer sub-conjunto de colunas, mas ao contrário dos índices B-tree e GiST, a análise deste índice não está dependente de que colunas estão na condição da pergunta.

Para criar um índice multi coluna deverá utilizar um comando com esta sintaxe:

```
CREATE INDEX index_name ON table_name (column_1,
    column_2);
```

Segundo a documentação, os índices multi-coluna devem ser usados cautelosamente e com moderação, já que na maioria das vezes um índice sobre uma coluna é suficiente e pode poupar espaço e tempo na construção e manutenção do índice.

Este tipo de índice também está presente nas possibilidades do Oracle 11g.

3.6 Combinação de índices

O PostgreSQL permite a combinação de índices de forma a conseguir suportar casos que não são possíveis de ser implementados por um único índice. Por exemplo, numa situação em que numa pergunta as colunas de um índice são separadas por um OR. Com esta possibilidade, consegue-se aplicar condições com AND e OR.

De forma a conseguir isto, o sistema do PostgreSQL segue os seguintes passos:

- Separa a condição pelos OR e pelos AND;
- Obtém os resultados individuais dessas condições, utilizando os índices mais adequados;
- Por fim, junta os resultados individuais, para obter o resultado final.

Para conseguir a combinação de múltiplos índices, o sistema faz as análises aos índices necessários e constrói um *bitmap* em memória, com a localização dos tuplos resultantes da aplicação das condições atribuídas a cada índice. Após isto, é feita a combinação de resultados usando operações de OR e AND nos *bitmaps*, de forma a conseguir a localização dos tuplos do resultado final. Por fim, este tuplos são visitados nas tabelas e retornados. É de referir que qualquer ordem imposta pelos índices usados é perdida e qualquer ordem requerida por uma cláusula `ORDER BY`, terá de ser tratada separadamente.

No SGBD Oracle 11g não é possível utilizar combinação de múltiplos índices. No entanto, deve-se referir que no Oracle 11g é possível usar índices *bitmap* explicitamente, mas no PostgreSQL tal não é possível, sendo apenas usado internamente.

3.7 Índices parciais

Um índice parcial é construído a partir de um sub-conjunto dos elementos de uma tabela, baseando-se numa condição/predicado. Portanto o índice apenas contém os tuplos de uma tabela que satisfazem esse predicado.

Uma das razões para a utilização deste tipo de índice é a necessidade de ignorar valores muito comuns, já que na procura de valores comuns o índice não seria muito utilizado. Este tipo de índice reduz o tamanho de um possível índice, o que vai fazer com que as perguntas que usam este índice sejam mais rápidas. As operações de atualização também serão mais rápidas, devido ao tamanho o índice.

Para criar um índice deste tipo podemos utilizar o seguinte comando:

```
CREATE INDEX index_name ON table_name (column)
    WHERE predicate ;
```

Convém dizer que para utilizar este tipo de índice devemos ter a certeza que este tipo de índice vai trazer melhorias ao desempenho de perguntas. Isto deve-se, segundo a documentação, ao facto de na maioria dos casos a vantagem deste tipo de índices em comparação com os anteriores é mínima.

No que diz respeito ao Oracle 11g, este não suporta índices parciais.

3.8 Índices únicos

Este tipo de índice permite que os valores de um certa coluna possam ser únicos ou, se for necessário, uma combinação de valores de uma ou mias colunas.

Atualmente, só os índices B-tree é que podem ser declarados únicos.

Neste caso, quando um índice é declarado único, tuplos com valores indexados iguais não são permitidos, bem como valores de múltiplas colunas indexadas num índice multi-coluna. Deve-se referir que os valores *null* não são considerados iguais.

O PostgreSQL utiliza este tipo de índices internamente, na criação de tabelas onde existe uma restrição *unique* ou uma chave primária.

Para criar um índice único utilizamos o comando:

```
CREATE UNIQUE INDEX index_name ON table_name (
    column [, ...]);
```

Índices deste tipo também são suportados pelo Oracle 11g.

3.9 Índices sobre expressões

Índices sobre expressões servem para ter um rápido acesso às tabelas, tendo como base uma função aplicada a uma ou mais colunas.

Um exemplo referido na documentação, é o caso da comparação de *strings* de forma a ignorar se uma *string* é maiúscula ou minúscula. Por exemplo, esta pergunta:

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

Esta pergunta poderá utilizar um índice criado com a função *lower*, de forma a obter os resultados de forma mais rápida.

A sintaxe para a criação de um índice deste tipo, é a seguinte:

```
CREATE INDEX index_name ON table_name ((expression
    ));
```

Este tipo de índice envolve custos de manutenção muito altos. Isto deve-se ao facto de na criação e a cada atualização, os valores das expressões utilizadas terão de ser computados. Por isso, estes índices devem ser utilizados quando a velocidade de obtenção de resultados é mais importante do que a velocidade de inserção e atualização de valores.

Por fim, estes índices também são suportados pelo Oracle 11g.

3.10 Exemplos

Nesta secção apresentamos alguns exemplos da utilização de índices. Mostrando também os resultados de custo quando temos esses índices e quando não os temos.

Como primeiro exemplo, mostramos um caso de como é que podemos usar índices *b-tree* e de *hash* e em que perguntas estes são usados. Começamos por mostrar a tabela *city*:

```
                Table "public.city"
   Column    |          Type          | Modifiers
-------------+------------------------+-----------
 name        | character varying(35)  | not null
 country     | character varying(4)   | not null
 province    | character varying(35)  | not null
 population  | numeric                |
 longitude   | numeric                |
 latitude    | numeric                |
Indexes:
    "citykey" PRIMARY KEY, btree (name, country,
       province)
    "city_country_hindex" hash (country)
    "city_country_index" btree (country)
Check constraints:
    "citylat" CHECK (latitude >= (-90)::numeric
       AND latitude <= 90::numeric)
    "citylon" CHECK (longitude >= (-180)::numeric
       AND longitude <= 180::numeric)
    "citypop" CHECK (population >= 0::numeric)
```

Esta tabela é proveniente da base de dados Mondial e apresenta alguns índices criados por nós, sendo que a sua criação foi feita da seguinte forma:

```
create index city_country_index on city (country);
create index city_country_hindex on city using
   hash (country);
```

Para demonstrar a utilização deste índices por parte do sistema, decidimos construir duas perguntas:

```
explain analyze select * from city where country>=
   'Z';
explain analyze select * from city where country='
   P';
```

Os custos obtidos antes da criação dos índices foram:

```
Seq Scan on city  (cost=0.00..62.89 rows=29 width
   =40) (actual time=14.048..14.394 rows=33 loops
   =1)
        Filter: ((country)::text >= 'Z'::text)
        Rows Removed by Filter: 3078
                Total runtime: 14.502 ms
```

```
Seq Scan on city  (cost=0.00..62.89 rows=24 width
   =40) (actual time=0.458..1.673 rows=24 loops=1)
        Filter: ((country)::text = 'P'::text)
        Rows Removed by Filter: 3087
                Total runtime: 1.755 ms
```

Conseguimos observar a utilização de pesquisas sequências na tabela *city* e os seus custos, sendo que tanto para a primeira pergunta e para a segunda pergunta obtivemos um custo entre 0.00 e 62.89. Os resultados das perguntas com os índices já criados são os seguintes:

```
Bitmap Heap Scan on city  (cost=4.50..30.15 rows
   =29 width=40) (actual time=24.745..25.072 rows
   =33 loops=1)
        Recheck Cond: ((country)::text >= 'Z'::
           text)
        ->  Bitmap Index Scan on
           city_country_index  (cost=0.00..4.50
           rows=29 width=0) (actual time
           =17.105..17.105 rows=33 loops=1)
        Index Cond: ((country)::text >= 'Z'::text)
        Total runtime: 25.164 ms

Bitmap Heap Scan on city  (cost=4.19..29.29 rows
   =24 width=40) (actual time=21.828..21.884 rows
   =24 loops=1)
        Recheck Cond: ((country)::text = 'P'::text
           )
        ->  Bitmap Index Scan on
           city_country_hindex  (cost=0.00..4.18
           rows=24 width=0) (actual time
           =8.584..8.584 rows=24 loops=1)
        Index Cond: ((country)::text = 'P'::text)
        Total runtime: 21.986 ms
```

No primeiro resultado conseguimos observar a utilização do índice *b-tree*, já que fazemos uma pergunta em que no predicado temos uma comparação em que este tipo de índice é bastante eficiente. No segundo resultado é usado o índice de *hash*, já que na pergunta efetuada temos um predicado de igualdade, que por sua vez é o tipo de predicado em que a utilização de um índice *hash* pode compensar. Também conseguimos verificar uma diminuição no limite superior dos custos, sendo que para a primeira pergunta baixou de 62.89 para 30.15 e para a segunda pergunta baixou de 62.89 para 29.29, o que demonstra que houve uma melhoria substancial em termos de custos.

O nosso segundo exemplo serve para demonstrar uma situação em que os índices GiST são utilizados. Para isso criámos uma tabela *spot*, que apenas tem dois atributos, *name* e *location* com o seguinte comando:

```
CREATE TABLE spot (
            name            varchar(80),
            location        point
        );
```

Como se pode ver pelo comando *name* é um *varchar* com 80 caracteres e *location* é um ponto geométrico com duas coordenadas.

E ainda foram inseridos 10000 tuplos na tabela com várias posições, podendo estas serem repetidas.

Após esta preparação, foi feita a seguinte pergunta:

```
explain analyze SELECT * FROM spot ORDER BY
   location <-> '(1,2)' LIMIT 5;
```

que basicamente devolve-nos os cinco tuplos que estão mais perto do ponto com coordenadas `(1,2)`.

Para esta pergunta sem termos criado qualquer índice, obtivemos este resultado:

```
Limit  (cost=360.10..360.11 rows=5 width=24) (
   actual time=37.658..37.676 rows=5 loops=1)
        ->  Sort  (cost=360.10..385.10 rows=10000
           width=24) (actual time=37.652..37.658
           rows=5 loops=1)
                Sort Key: ((location <-> '(1,2)'::
                   point))
                Sort Method: top-N heapsort
                   Memory: 17kB
                        ->  Seq Scan on spot  (
                           cost=0.00..194.00 rows
                           =10000 width=24) (
                           actual time
                           =0.024..19.991 rows
                           =10000 loops=1)
        Total runtime: 37.738 ms
```

Com este resultado podemos observar o uso de um método de ordenação antes de fazer a pesquisa sequencial pela tabela *spot*. Também é possível ver o custo dessa pesquisa sequencial que é pode estar entre 0.00 e 194.00.

Se criarmos o seguinte índice:

```
CREATE INDEX spot_gistidx ON spot using gist (
   location);
```

e executarmos a mesma pergunta, vamos obter este resultado:

```
Limit  (cost=0.15..0.57 rows=5 width=24) (actual
   time=0.327..0.367 rows=5 loops=1)
        ->  Index Scan using spot_gistidx on spot
            (cost=0.15..844.15 rows=10000 width
           =24) (actual time=0.308..0.331 rows=5
           loops=1)
        Order By: (location <-> '(1,2)'::point)
        Total runtime: 0.488 ms
```

Aqui conseguimos não só ver a utilização de o índice criado, mas também a ausência do método de ordenação que causa sempre algum *overhead*.

Capítulo 4

Processamento e Otimização de Perguntas

A execução de uma pergunta no PostgreSQL passa por vários passos. Esses passos são:

1. É estabelecida uma **conexão** ao servidor que tem o PostgreSQL. A aplicação que iniciou a comunicação envia a pergunta e fica à espera de uma resposta do servidor com os resultados. Os restantes passos são executados exclusivamente no servidor.

2. É feito um ***parsing* da pergunta** para verificar se a sintaxe está correta e cria uma *query tree*.

3. O **sistema de reescrita** recebe a *query tree* e procura por regras nos catálogos do sistema que possa aplicar à pergunta.

   Nesta fase também é feito o processamento das *views*. Sempre que uma *query* refere uma *view*, esta é reescrita de forma a utilizar as tabelas usadas na sua definição.

4. O **planeador/otimizador** recebe a *query tree* reescrita e cria o plano que será executado na fase seguinte, escolhendo o plano de menor custo de entre todos os que gerem o resultado esperado.

5. Por fim, o **executor** percorre recursivamente a árvore do plano recebido da fase anterior, executa-o e devolve os tuplos do resultado.

Cada uma destas fases será descrita nas secções seguintes em maior detalhe.

4.1 Conexão

O PostgreSQL utiliza um modelo cliente-servidor que recorre a um processo independente para cada conexão, ou seja, para cada processo cliente há um processo servidor diferente.

Como o número de conexões que serão feitas não pode ser previsto, é utilizado um processo *master* que lança um novo processo sempre que uma conexão

é estabelecida. Estas conexões podem ser feitas a partir de *sockets* TCP/IP ou *sockets* Unix. Os processos do servidor utilizam memória partilhada e semáforos sempre que necessitam de aceder aos dados de forma concorrente sem comprometer integridades destes.

Quando uma conexão é estabelecida entre o servidor e o cliente, este envia a pergunta em *plaintext*, ou seja, todo o processamento é feito no servidor. Depois de o servidor processar a *query*, os resultados são enviados pela mesma conexão.

4.2 *Parsing*

A fase de parsing divide-se em duas partes:

1. *Parsing* da pergunta;
2. Processo de transformação da árvore de *parsing* através de modificações.

O *parser* das perguntas SQL tem a sua gramática definida no ficheiro *gram.y* e o léxico no ficheiro *scan.l*. Utilizando as ferramentas do Unix *bison* e *flex*, é possível compilar um *parser* a partir destes ficheiros.

O *parser* começa por verificar a sintaxe da pergunta, recebida em *plaintext*. Se a pergunta for válida, e construída uma *parse tree*; caso contrário, é devolvido um erro.

Depois de construída a *parse tree*, o processo de transformação faz uma interpretação semântica, verificando que tabelas, funções ou operadores são referidos pela pergunta. Com base nesta informação, é construída uma *query tree*.

4.3 Reescrita

O sistema de reescrita é utilizado sempre que a pergunta faz referência a *views*. Inicialmente, o PostgreSQL recorria ao sistema de reescrita apenas na fase execução, sempre que um tuplo era acedido. Contudo, isto foi mudado em 1995, quando o projeto Berkeley Postgres foi transformado no Postgres95.

Desde então, o sistema de reescrita é implementado utilizando uma técnica chamada reescrita de perguntas. Esta técnica é bastante semelhante à expansão de macros, em linguagens como C ou C++, substituindo as referências às *views* pelas perguntas que estas representam.

No entanto, se as perguntas envolverem operações de modificação das tabelas (`INSERT`, `UPDATE` ou `DELETE`), a reescrita nem sempre é trivial. Se a *view* for complexa (por exemplo, manipulando várias tabelas), é necessário recorrer a *triggers* definidos pelo utilizador.

A partir da versão 9.3, o PostgreSQL passou a suportar *views* materializadas. Uma *view* materializada é criada da seguinte forma:

```
CREATE MATERIALIZED VIEW table_name
    [ (column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ... ]
       ) ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]
```

Uma *view* materializada permite armazenar em disco os resultados da pergunta representada pela *view*, tornando o acesso aos resultados de uma view materializada mais eficiente.

Uma das desvantagens de uma *view* materializada é não estar constantemente atualizada. Para refrescar os resultados, o utilizador da base de dados necessita de correr o comando:

```
REFRESH MATERIALIZED VIEW name
    [ WITH [ NO ] DATA ]
```

Este tipo de views também está disponível no Oracle.

4.4 Planeamento/Otimização

Uma pergunta SQL pode ser executada de diversas formas, devolvendo sempre o mesmo resultado. O objetivo da fase de planeamento/otimização é encontrar a forma menos computacionalmente dispendiosa e gerar o plano de execução respetivo.

O planeador/otimizador começa por gerar todos os planos possíveis para cada uma das tabelas envolvidas na pergunta.

Por exemplo, se a pergunta tiver uma restrição semelhante a `atributo1 < atributo2` e houver um índice de árvore B+ que tenha como chave o `atributo1` , então será criado um plano em que é utilizado o índice referido no *scan* da relação. Como todas as tabelas podem ser percorridas sequencialmente, este plano também é sempre considerado.

Se a pergunta envolver uma junção de duas ou mais tabelas, o planeador/otimizador gera todos os planos possíveis para esta operação só depois de ter encontrado as melhores alternativas para fazer o *scan* individual de cada tabela. O PostgreSQL tem à sua disposição três estratégias de junção: *nested loop join*, *merge join* e *hash join*. Estes algoritmos serão descritos mais adiante neste capítulo.

Caso a junção envolva três ou mais tabelas, é preciso encontrar a melhor sequência de junções. Para construir o melhor plano, o planeador/otimizador testa as junções entre pares de tabelas para as quais exista uma clásula de junção no comando `WHERE`. Caso haja uma tabela que não possua uma cláusula de junção, também são consideradas as junções entre esta e todas as outras tabelas.

Para além das junções, o planeador/otimizador também tem em conta qual a melhor sequência para outras operações, como seleção, projeção ou ordenação. Por exemplo, na maioria dos casos, é preferível fazer estas operações antes das junções, de forma a diminuir o tamanho das tabelas e melhorar a eficiência. Os nós da árvore do plano que ficam responsáveis por este tipo de operações são anotados com essa informação.

No final da fase, o planeador/otimizador terá criado o melhor plano de execução que encontrou para a pergunta, que deve ser, posteriormente, utilizado pelo executor.

4.4.1 Genetic Query Optimizer

Caso o número de tabelas envolvidas numa junção ultrapasse o valor definido pelo parâmetro `geqo_threshold` (por omissão 12, com a possibilidade

de ser alterado pelo utilizador), a técnica descrita anteriormente, de realizar uma procura exaustiva do melhor plano, deixa de ser eficiente. Em alternativa, o PostgreSQL utiliza uma técnica chamada Genetic Query Optimization (GEQO).

O GEQO aborda o plano da junção de tabelas de forma semelhante ao conhecido problema do "Caixeiro Viajante". Cada plano de execução é codificado numa *string* que representa a sequência em que deve ser realizada cada junção entre duas tabelas. Por exemplo, a string '4-2-3-1' representa a seguinte junção $(((4 \bowtie 2) \bowtie 3) \bowtie 1)$, sendo que cada algarismo representa uma tabela.

Inicialmente, o GEQO gera aleatoriamente vários planos de junção. O GEQO calcula o custo de cada plano, testando os vários algoritmos de junção para cada um dos nós da árvore de junção. O planos com um um menor custo são considerados mais aptos que os outros, sendo utilizados para gerar novos planos. Os menos aptos são descartados. Este processo é repetido várias vezes. Por fim, é escolhido o plano que apresenta o menor custo de entre todos os que foram gerados ao longo de todas as iterações.

O PostgreSQL tem vários parâmetros que permite ao utilizador configurar a forma como o GEQO é executado, tais como:

geqo_pool_size O tamanho da população em cada iteração;

geqo_generations O número máximo de iterações;

geqo_seed A semente utilizada na geração aleatório dos planos.

4.5 Execução

O executor recebe o plano gerado pelo planeador/otimizador e processa-o recursivamente para obter os tuplos necessários. Isto é realizado usando um mecanismo de *demand-pull pipeline*, ou seja, sempre que um nó do plano é chamado, tem de devolver mais um tuplo ou informar que não há mais tuplos para serem obtidos.

Essencialmente, cada nó computa e devolve o próximo tuplo sempre que é chamado, sendo o responsável por executar as operações de seleção ou projeção que lhe foram anotadas pelo planeador/otimizador.

Caso a pergunta a ser executada for uma seleção (ou seja, uma pergunta do tipo `SELECT`), o executor apenas necessita de enviar ao cliente cada um dos tuplos devolvidos pela árvore de execução. Caso a pergunta modifique a base de dados (`INSERT`, `UPDATE` ou `DELTE`), cada um dos tuplos é enviado para um nó especial, denominado `ModifyTable`. No caso de uma inserção, basta enviar o tuplo a inserir para o nó `ModifyTable`. Caso for uma atualização, para cada tuplo, envia-se todos os atributos atualizados juntamente com o TID (*tuple ID*). Por fim, se for uma remoção, basta enviar o TID, que o nó `ModifyTable` utiliza para marcar o tuplo como tendo sido apagado.

4.6 Mecanismos para Expressões Complexas

Tal como o Oracle 11g, o PostgreSQL tem suporte para os principais mecanismos para expressões complexas: materialização, *pipelining* e paralelização.

Porém, não há qualquer forçar o uso destes mecanismos, dado que, ao contrário do Oracle, o PostgreSQL não tem suporte para *hints* nas perguntas SQL, ficando a decisão de que mecanismo usar a cargo do planeador/otimizador.

4.6.1 Materialização

A materialização é utilizado sempre que é conveniente armazenar o resultado de um nó da árvore de execução. Por exemplo, o *nested loop join* necessita de percorrer várias vezes a tabela da direita, então, caso esta não caiba em memória, é preferível materializá-la temporariamente em disco.

4.6.2 *Pipelining*

Nos casos em que não é necessário ou não traz vantagens escrever os outputs intermédios em memória, utiliza-se a técnica de pipelining. Esta técnica utiliza um modelo de iterador, onde um nó da árvore de execução vai obtendo um tuplo de cada vez por um nó filho. Desta forma, não há custo associado na escrita do *output* em disco por parte de um nó e conseguinte leitura por outro.

4.6.3 Paralelização

A paralelização permite vários processos processarem uma pergunta SQL em paralelo. Esta técnica pode ser utilizada em várias operações, tais como ordenações, junções, leituras sequenciais de ficheiros grandes, agregações, verificação de condições, etc.

4.7 Algoritmos

Neste capítulo apresentamos uma pequena descrição sobre os algoritmos implementados pelo PostgreSQL para realizar as operações de seleção, ordenação, junção e agregação.

4.7.1 Algoritmos de Seleção

Scan Sequencial

Este é o algoritmo mais simples para a operação de seleção. Consiste em percorrer toda a tabela e testar a condição da pergunta contra cada um dos tuplos. Devido ao facto de percorrer toda a tabela, é o algoritmo de seleção menos eficiente.

Index Scan

Caso a operação de seleção envolva uma igualdade ou um intervalo de valores sobre um atributo que tenha um índice, este índice é utilizado como estrutura auxiliar na pesquisa pelos tuplos que cumpram a condição. No entanto, se a operação envolver a obtenção de muitos tuplos, o *index scan* poderá revelar-se menos eficiente que o *scan* sequencial, dado que necessita dois acessos ao disco (um ao índice e outro à tabela).

Bitmap Index Scan

No *bitmap index scan*, primeiro é utilizado um índice existente para construir um *bitmap* com base nos IDs dos tuplos. Seguidamente, este *bitmap* é utilizado para saber que tuplos se deve obter da *heap*, permitindo fazê-lo sequencialmente e aceder a cada página da *heap* apenas uma vez.

Os *bitmaps* também são úteis para predicados booleanos complexos. No entanto, caso os atributos utilizados como chave do índice *bitmap* tenham vários valores distintos, este tipo de índice pode-se revelar pouco eficiente.

4.7.2 Algoritmos de Ordenação

External Merge Sort

O *external merge sort* divide a tabela em várias porções, denominadas *runs*, que são ordenadas em memória. Depois de concluída esta primeira fase, os vários *runs* são combinados de forma a termos toda a tabela ordenada.

Quicksort

Caso a tabela caiba totalmente em memória, o PostgreSQL utiliza o conhecido algoritmo de ordenação *quicksort*. O *quicksort* é um algoritmo recursivo que em cada chamada divide os elementos em duas partes com base num elemento *pivot*: os maiores que o *pivot* e os menores que o *pivot*. Depois de ordenadas cada uma das partes, concatena-as, ficando todos os elementos por ordem.

4.7.3 Algoritmos de Junção

Nested-Loop Join

No algoritmo de junção *nested-loop* é feito um *scan* completo à relação da direita por cada tuplo na relação da esquerda, sendo verificada a condição em cada leitura de um tuplo. O PostgreSQL também implementa uma variante onde utiliza um índice na relação da direita.

Merge Join

No *merge join* cada uma das relações é ordenada segundo os atributos de junção. De seguida, é feito um *scan* paralelo a cada uma das relações, sendo os tuplos combinados sempre que cumprirem a condição de junção. Este algoritmo é mais eficiente que o *nested-loop join* porque necessita de fazer apenas um *scan* a cada relação.

Hash Join

No *hash join* a relação da direita é carregada em memória numa *hash table*, usando os atributos de junção como chave. De seguida, é feito um *scan* na relação da esquerda e, para cada tuplo, utiliza-se os atributos de junção para encontrar os tuplos correspondentes na *hash table*.

4.7.4 Algoritmos de Agregação

Agregação com base na ordenação

Na agregação com base na ordenação, a relação é ordenada pelos atributos de agregação, os tuplos são reunidos em grupos e, finalmente, as operações de agregação são realizadas em cada um dos grupos.

Agregação com base no *hash*

A agregação com base no *hash* é semelhante ao algoritmo anterior, com a diferença de se utilizar uma *hash table* com base nos atributos de agregação. Esta versão é preferível quando o tamanho da relação é menor.

4.7.5 Algoritmos de Eliminação

Os algoritmos de eliminação são equivalentes aos de agregação, com a diferença de se eliminarem os tuplos que forem idênticos em vez de se aplicarem as operações de agregação.

4.8 Estimativas e Planos de Execução

Tal como no Oracle 11g, o PostgreSQL possui comandos que permite ao utilizador atualizar as estatísticas, utilizadas pelo planeador/otimizador para verificar os custos de cada plano, e verificador o plano de execução de uma pergunta.

4.8.1 ANALYZE

Para atualizar as estatísticas utiliza-se o seguinte comando

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

Usando as opções `table_name` e `column_name` é possível parametrizar que tabelas e atributos se pretende atualizar.

As estatísticas recolhidas por este comando são armazenados na tabela `pg_statistic`. Há também a *view* `pg_stats` que apresenta apenas parte da informação mas de uma forma mais legível.

4.8.2 EXPLAIN

Caso se pretenda verificar o plano que é executado para uma dada pergunta, o PostgreSQL dispõe do seguinte comando:

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

Este comando apresenta que algoritmos são utilizados em cada nó da árvore de execução e algumas estimativas, nomeadamente:

Custo estimado de arranque Tempo que um nó demora até começar a produzir *output* (por exemplo, o tempo gasto a ordenar inicialmente os tuplos num nó de ordenação);

Custo estimado total Tempo que um nó demora a fazer o *output* de todos os tuplos;

Número estimado de tuplos Quantos tuplos um nó produz como output;

Comprimento médio estimado Tamanho médio dos tuplos produzidos como *output* por um nó.

4.8.3 Exemplos de planos

Para verificar a utilização dos algoritmos, podemos utilizar os comandos ANALYZE e EXPLAIN para analisar os planos de execução. Nos exemplos seguintes, utilizamos a base de dados Mondial para executar as perguntas.

Por exemplo, se corremos a seguinte pergunta:

```
EXPLAIN ANALYZE SELECT * FROM ismember;
```

verificamos que o plano é:

```
 Seq Scan on ismember  (cost=0.00..127.08 rows
    =8008 width=15) (actual time=0.181..100.621
    rows=8008 loops=1)
 Total runtime: 203.953 ms
(2 rows)
```

Nesta pergunta, como se trata de apenas obter todos os tuplos e todos os atributos de uma única tabela, o melhor algoritmo a utilizar é o *scan* sequencial, como podemos verificar pelo plano.

Caso se faça uma ordenação por um atributo que possuía um índice, como na pergunta seguinte:

```
EXPLAIN ANALYZE SELECT * FROM ismember ORDER BY
   country;
```

já se torna mais vantajoso utilizar um índice para realizar o scan, o que é confirmado pelo plano:

```
 Index Scan using memberkey on ismember  (cost
    =0.28..452.40 rows=8008 width=15) (actual time
    =0.503..117.244 rows=8008 loops=1)
 Total runtime: 209.026 ms
(2 rows)
```

Porém, caso se faça uma ordenação por um atributo que não seja a chave principal de um índice, como na pergunta seguinte:

```
EXPLAIN ANALYZE SELECT * FROM ismember ORDER BY
   organization;
```

o PostgreSQL já opta por realizar primeiro um scan sequencial e, só depois de os tuplos estarem todos, realizar uma ordenação. Como o tamanho da tabela é relativamente pequeno, o PostgreSQL utiliza um simples quicksort em detrimento de um algoritmo mais pesado, como o merge sort. Isto pode ser verificado no plano seguinte:

```
Sort  (cost=646.29..666.31 rows=8008 width=15) (
    actual time=216.876..301.210 rows=8008 loops
    =1)
  Sort Key: organization
  Sort Method: quicksort  Memory: 634kB
  ->  Seq Scan on ismember  (cost=0.00..127.08
      rows=8008 width=15) (actual time
      =0.175..95.842 rows=8008 loops=1)
Total runtime: 373.814 ms
(5 rows)
```

Caso se adicione uma igualdade na cláusula `WHERE`, como na pergunta seguinte:

```
EXPLAIN ANALYZE SELECT * FROM ismember WHERE
   country = 'GB' ORDER BY organization;
```

o PostgreSQL utiliza o índice existente para criar um *bitmap* temporário onde armazena os IDs de todos os tuplos cujo atributo `country` é `GB`. De seguida, utiliza este *bitmap* para aceder sequencialmente às páginas do *heap* e obter os tuplos necessários. Por fim, com os tuplos carregados em memória, o PostgreSQL utiliza o *quick sort* para os ordenar. Este processo está representado no plano da seguinte forma:

```
Sort  (cost=56.97..57.15 rows=74 width=15) (
    actual time=3.465..4.600 rows=74 loops=1)
  Sort Key: organization
  Sort Method: quicksort  Memory: 29kB
  ->  Bitmap Heap Scan on ismember  (cost
      =4.86..54.67 rows=74 width=15) (actual time
      =0.811..1.904 rows=74 loops=1)
        Recheck Cond: ((country)::text = 'GB'::
            text)
        ->  Bitmap Index Scan on memberkey  (cost
            =0.00..4.84 rows=74 width=0) (actual
            time=0.737..0.737 rows=74 loops=1)
              Index Cond: ((country)::text = 'GB'
                  ::text)
Total runtime: 5.989 ms
(8 rows)
```

Quanto a junções, podemos verificar o uso de um *hash join* na pergunta seguinte:

```
EXPLAIN ANALYZE SELECT c.code,c.name,l.name FROM
   Country c INNER JOIN Language l ON c.code = l.
   country;
```

Nesta pergunta, é obtida a lista de países de todos os países, juntamente com as linguagens respetivas. Neste caso, é utilizado um *hash join* usando os atributos `name` da tabela `Country` e `country` da tabela `Language` para efetuar a junção. Isto está representado no plano da seguinte forma:

```
Hash Join  (cost=8.36..12.78 rows=144 width=20) (
    actual time=27.100..39.507 rows=144 loops=1)
```

```
   Hash Cond: ((l.country)::text = (c.code)::text)
   ->  Seq Scan on language l  (cost=0.00..2.44
      rows=144 width=11) (actual time=0.071..5.577
       rows=144 loops=1)
   ->  Hash  (cost=5.38..5.38 rows=238 width=12) (
      actual time=26.915..26.915 rows=238 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage:
            11kB
         ->  Seq Scan on country c  (cost
            =0.00..5.38 rows=238 width=12) (actual
             time=0.018..23.174 rows=238 loops=1)
 Total runtime: 53.880 ms
(7 rows)
```

Se tirarmos a condição de junção, o PostgreSQL já opta por um *nested loop join*, já que a utilização de tabelas de *hash* é desnecessária neste caso.

```
EXPLAIN ANALYZE SELECT c.code,c.name,l.name FROM
   Country c,Language l;
```

A esta pergunta corresponde o seguinte plano:

```
 Nested Loop  (cost=0.00..436.58 rows=34272 width
    =20) (actual time=0.607..1092.346 rows=34272
    loops=1)
   ->  Seq Scan on country c  (cost=0.00..5.38
      rows=238 width=12) (actual time=0.499..2.952
       rows=238 loops=1)
   ->  Materialize  (cost=0.00..3.16 rows=144
      width=8) (actual time=0.009..1.519 rows=144
      loops=238)
         ->  Seq Scan on language l  (cost
            =0.00..2.44 rows=144 width=8) (actual
            time=0.037..2.388 rows=144 loops=1)
 Total runtime: 1447.693 ms
(5 rows)
```

4.9 Comparação com o Oracle

Comparativamente com o PostgreSQL, o Oracle apresenta um processamento de perguntas semelhante, também apresentando uma fase de *parsing*, otimização (utilizando estatísticas sobre as tabelas) e execução. Uma diferença substancial no Oracle relativamente à otimização é a presença de *hints*, que permite ao utilizador forçar a utilização de certos algoritmos ou índices.

Quanto aos algoritmos e processamento de perguntas complexas, a diferença entre o PostgreSQL e Oracle também é diminuta.

Capítulo 5

Transações e Controlo de Concorrência

O PostgreSQL utiliza um modelo de multiversão, o *Multiversion Concurrency Control* (MVCC) para assegurar a consistência dos dados e gestão de transações, isto pois suporta várias sessões de bases de dados em simultâneo e, como tal necessita de garantir isolamento entre transações e entre as sessões, para isso usando *snapshots* da base de dados. Uma das vantagens de usar este método de controlo de concorrência é o facto de os *locks* de MVCC de *read* não entrarem em conflito com os *locks de write* e portanto operações de leitura nunca bloqueiam operações de escrita e vice-versa. O PostgreSQL consegue assegurar estas propriedades mesmo usando o nível de isolamento mais restrito através do uso do nível *Serializable Snapshot Isolation* (SSI).

Por omissão, todos os comandos são executados como transações, mas também é permitido definir blocos de transações explicitamente, usando a instrução `BEGIN` para demarcar o inicio da transação a acabando com `COMMIT`. Para um controlo mais específico, podem ser criados pontos de restauro através da instrução `SAVEPOINT`, aos quais podemos voltar usando `ROLLBACK TO`. O PostgreSQl não tem implementação para *nested transactions*, mas é possível atingir o mesmo efeito fazendo uso dos pontos de restauro. O sistema também não impõe limites na duração das transações de longa duração, embora seja desencorajado o uso de transações muito longas.

Para os exemplos seguintes usaremos a tabela: bank(name varchar(10) primary key, balance int), com os tuplos ('Bob', 100) e ('Alice', 200).

Exemplo de uma transação com uso de savepoints e rollback:

```
BEGIN;
        UPDATE bank SET balance = balance - 100
           where name = 'Bob';     SAVEPOINT
           savepoint1;
        UPDATE bank SET balance = balance + 100
           where name = 'Alice';
        ROLLBACK TO savepoint;
COMMIT;
```

Os resultados desta transação são os tuplos ('Bob', 0) e ('Alice', 200), o que era de esperar pois voltámos ao ponto em que o update ao segundo tuplo ainda

não tinha sido executado.

5.1 Locks e Resolução de Deadlocks

O PostgreSQL utiliza vários modos de *lock* para o controlo de concorrência de acessos a tabelas, sendo que a maioria das suas instruções adquirem os *locks* adequados à situação automaticamente.

O PostgreSQL possui uma variada lista de *locks* ao nível das tabelas, sendo que estes são adquiridos automaticamente, podendo, no entanto, recorrer-se à instrução `LOCK` para adquirir algum dos *locks* explicitamente. Os tipos de *lock* são: *access share, row share, share update exclusive, share, share row exclusive, exclusive e access exclusive*, estando estes ordenado por número de conflitos com outros *locks*, por orden crescente, sendo que o *access exclusive* não permite acesso a mais nenhum outro.

Para além dos *locks* ao nível das tabelas, o PostgreSQL também possui *locks* ao nível dos tuplos, sendo que estes podem ser exclusivos ou partilhados. Os exclusivos são adquiridos automaticamente quando se atualiza ou apaga um tuplo, mas para os adquirir explicitamente, pode usar-se a instrução `SELECT FOR UPDATE` para os exclusivos, ou `SELECT FOR SHARE` para os partilhados.

Ao adquirir *locks* explicitamente, aumenta-se o risco de causar um *deadlock* entre transações. O PostgreSQL deteta esses casos automaticamente e resolve-os abortando uma das transações, para que a outra possa seguir a sua execução.

Exemplo de transações concorrentes:

- T1: `begin;`
- T2: `begin;`
- T1: `insert into bank (name, balance)values ('Carlos', 100);`
- T2: `insert into bank (name, balance)values ('Carla', 100);`
- T1: `insert into bank (name, balance)values ('Carla', 100);`
- T2: `insert into bank (name, balance)values ('Carlos', 100);`
- T1: `commit;`
- T2: `commit;`

Neste exemplo ambas as transações estavam a adquirir *locks* automaticamente, pelo que, quando cada uma tentou inserir o segundo tuplo na tabela, ficaram em espera, causando um *deadlock*. O PostgreSQL abortou T2 para que T1 pudesse continuar a sua execução.

5.2 Níveis de Isolamento

O PostgreSQL permite ao utilizador a escolha entre os quatro níveis de isolamento definidos pelo standard SQL, embora, internamente apenas implemente três desses níveis, o *Read Committed, Repeatable Read e Serializable*. A razão pela qual apenas estão implementados estes três é porque esta é uma das melhores maneiras de fornecer os níveis de isolamento standard numa arquitetura de controlo de concorrência multi-versão.

5.2.1 Read Committed

O *Read Committed* é o nível de isolamento definido por omissão no PostgreSQL, sendo que com este nível tem uma visibilidade sobre todas as alterações de dados, desde que as transações que os alteraram tenham feito *commit*. Este é o nível mais baixo de isolamento pois uma transação vê as alterações feitas por outras transações aos dados que manipula. Para fornecer este nível de isolamento, o sistema cria *snapshots* da base de dados antes da execução de cada instrução.

Exemplo de utilização:

Nestes exemplos usaremos o seguinte escalonamento de transações e iremos comparar o resultado conforme o nível de isolamento.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

- T1: `begin;`
- T2: `begin;`
- T1: `update bank set balance = balance + 100 where name = 'Bob ';`
- T2: `delete from bank where balance = 100;`
- T1: `commit;`
- T2: `commit;`

Neste caso, apesar de o nível de isolamento ser *Read Committed*, como não foi executado *commit* antes da operação de T2, esta acabou por ficar suspensa até T1 terminar.

5.2.2 Repeatable Read

No caso deste nível de isolamento, que só tem visão sobre os dados que foram *commited* antes da transação começar, não tendo visão sobre valores *uncommitted* ou modificações por transações concorrentes, o sistema cria apenas uma *snapshot* antes do início da transação. Assim consegue-se uma melhor garantia de que a transação tem uma visão consistente da base de dados.

Exemplo de utilização:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Também, neste caso, T2 ficou suspensa ao tentar executar a operação de *delete*.

5.2.3 Serializável

Para o nível de isolamento seriaizável, que é o nível mais rigoroso, pois emula uma execução em série de transações concorrentes, o sistema usa os mesmos métodos do *Repeatable Read*, com a exceção da adicional monitorização de condições que possam tornar a execução de um conjunto de transações serializáveis concorrentes inconsistente em todas as possíveis serializações. Uma estratégia que o PostgreSQL utiliza para garantir a serialização de transações é

o uso do protocolo de *predicate locking*, que mantém *locks* que permitem que o sistema determine quando uma qualquer operação de escrita de uma transação possa afetar o resultado de uma operação de leitura prévia de uma transação concorrente, caso esta tenha iniciado primeiro. No PostgreSQl a utilização destes *locks* não acarreta o risco de *deadlock* pois estes não são bloqueantes, servindo apenas para assinalar e identificar dependências entre transações serializáveis concorrentes que, em certas combinações, podem levar a problemas de serialização.

Exemplo de utilização:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

5.3 Consistência de Dados

Em termos de consistência, dada a natureza do nível de isolamento *Read Committed*, o PostgreSQL tem grandes dificuldades em garantir a consistência dos dados sem suporte adicional. Já com o nível *Repeatable Read*, o sistema tem um pequeno problema em manter a consistência caso não tenha cuidados extra. Esse problema deve-se ao facto de usar *snapshots* do tipo MVCC, é chamado conflito *read/write*. Este caso acontece quando se forma um ciclo no grafo de precedências das transações. Caso o nível de isolamento utilizado para todos os *reads e writes* não seja o *serializable*, então é necessário fazer uso dos *locks* para assegurar a consistência dos dados, sendo que o PostgreSQL os adquire automaticamente caso o nível não seja o acima mencionado. Como existem níveis de isolamento podem não garantir a consistência dos dados, o PostgreSQL oferece ao utilizador um método de verificação de integridade da base de dados, que pode ser configurado para correr no fim de cada transação (`DEFERRED`), ou entre as instruções das trasnações (`IMMEDIATE`).

Exemplo de utilização:

```
SET CONSTRAINTS constraint1 DEFERRED;
```

5.4 Recuperação de Dados

Como mecanismo de recuperação, o PostgreSQL mantém *logs* do tipo WAL *(Write Ahead Log)*. Estes registam qualquer tipo de alteração à base de dados, servindo primariamente para recuperação de *crash*.

O uso deste mecanismo permite garantir a durabilidade de uma transação, sem precisar de que toda a operação seja escrita em disco, escrevendo apenas no *log* e, depois em disco, quando se der a operação de escrita de *logs*. Por outro lado, este mecanismo permite simplificar o caso em que uma transação é abortada pois o sistema não tem que alterar ficheiros em disco para fazer *rollback*, bastando, para isso, alterar o *log* em que foi escrito o resultado da transação. O facto de o PostgreSQL utilizar o MVCC já dá algum suporte a este mecanismo de recuperação.

5.5 Comparação com o Oracle

Comparando ambos os sistemas concluímos que no aspeto de controlo de concorrência são bastante semelhantes, havendo algumas pequenas diferenças, como por exemplo um maior número de *lock modes* que o PostgreSQL possui.

Capítulo 6

Suporte para Bases de Dados Distribuídas

Hoje em dia grande parte das maiores empresas, dos maiores sistemas e grande parte das aplicações utilizam base de dados distribuídas. Isto deve-se não só à quantidade de dados que é necessário guardar, mas também devido à necessidade de disponibilidade da base de dados em si e também devido a uma questão de eficiência nas respostas aos pedidos efetuados. Com suporte para base de dados distribuídas conseguimos obter melhores tempos de resposta e maior segurança na disponibilidade destas.

Esta distribuição pode ser feita de duas formas: utilizando uma base de dados com distribuição homogénea, em que se utiliza o mesmo sistema de gestão de base de dados em todos os pontos, os dados são distribuídos por todos os pontos e o objetivo é esconder esta distribuição dos utilizadores, de forma a que estes pensem estar a lidar com uma base de dados local; ou utilizando uma base de dados com distribuição heterogénea, que pode ter diferentes sistemas de gestão de base de dados em cada ponto. Este tipo tem como objetivo a integração de várias base de dados, de forma a conseguir obter um funcionalidade útil aos utilizadores.

Porém, o PostgreSQL não oferece este tipo de suporte como descrito em cima. Contudo, existem mecanismos deste sistema de gestão de base de dados que permitem fazer alguma replicação dos dados de um base de dados, como por exemplo o *Streaming Replication* e o *Synchronous Replication*. Para além disso, também existem algumas extensões que possibilitam alguma distribuição.

Nas secções seguintes falamos destes tipos de replicação e de algumas extensões que podem ser utilizadas.

6.1 Streaming Replication

Neste tipo de replicação temos um servidor com o nome de servidor *standby* e um servidor primário. O servidor *standby* é uma replica do servidor primário que vai guardando as alterações feitas no servidor primário, ou base de dados central. Com isto, o servidor *standby* liga-se ao servidor primário, que lhe transmite registos WAL(*Write Ahead Logging*) à medida que estes são gerados.

Este tipo de replicação é assíncrono, o que quer dizer que pode haver algum atraso entre o *commit* de uma transação no servidor primário e a visibilidade das alterações no servidor *standby*.

6.2 Synchronous Replication

Se ocorrer uma falha no servidor primário e algumas transações, que já tinham feito *commit*, não forem replicadas para o servidor *standby*, então teremos perda de dados devido à replicação ser assíncrona.

Contudo o *Synchronous Replication* dá-nos a habilidade de confirmar que todas as alterações feitas por uma transação foram enviadas para um servidor de *standby* síncrono, o que faz com o nível de durabilidade de uma transação seja ampliado. Este tipo de replicação permite que a replicação de dados seja síncrona, ou seja, cada transação de escrita vai esperar até que haja confirmação que o *commit* foi escrito no *log* de transações do servidor primário e do servidor *standby*. Já para transações de leitura e *rollbacks* de transações não é necessário esperar por uma respostas dos servidores *standby*. No caso de carregamento de dados ou criação de índices, estes também não esperam até ao *commit* final. No que diz respeito a ações *two-phase commit*, estas tem de espera pelo *commit*.

A única forma de perder dados neste tipo de replicação é se o servidor primário e os servidores *standby* tiveram uma falha ao mesmo tempo.

6.3 Extensões

Nesta tabela estão algumas das extensões, que são possíveis integrar com o PostgreSQL. Contudo não falamos mais delas, já que não fazem parte do sistema estudado.

Extensão	**Licença**	**Estado**	**Método de replicação**	**Sincronismo**
PgCluster	BSD	Estagnado	Master-Master	Síncrono
pgpool-I	BSD	Estável	Statement-Based Middleware	Síncrono
Pgpool-II	BSD	Recente	Statement-Based Middleware	Síncrono
slony	BSD	Estável	Master-Slave	Assíncrono
Bucardo	BSD	Estável	Master-Master, Master-Slave	Assíncrono
Londiste	BSD	Estável	Master-Slave	Assíncrono
Mammoth	BSD	Estagnado	Master-Slave	Assíncrono
rubyrep	MIT	Estagnado	Master-Master, Master-Slave	Assíncrono

Tabela 6.1: Extensões de suporte a bases de dados distribuídas.

Em contraste com o PostgreSQL, o Oracle oferece suporte nativo para bases de dados distribuídas homogéneas e heterogéneas.

Para o caso das bases de dados distribuídas homogéneas, o Oracle fornece um mecanismo de alias, que permite abstrair a localização de uma tabela na rede de servidores quando um utilizador executa uma pergunta.

Quanto às bases de dados heterogéneas, o Oracle também abstrai o utilizador da heterogeneidade das bases de dados que não sejam Oracle. Para isso, basta implementar um agente responsável pela comunicação com as outras bases de

dados. Este utiliza, por exemplo, o protocolo ODBC para executar perguntas na base de dados heterogénea remota.

Capítulo 7

Outras Características do PostgreSQL

7.1 Pesquisa de Texto

A pesquisa de texto do PostgreSQL permite identificar documentos em linguagem natural que satisfaçam uma pergunta e ordená-los por ordem de relevância.

Embora o SQL standard tenha operadores como `~`, `~*`, `LIKE` e `ILIKE`, todos eles têm problemas como:

- As expressões regulares não têm suporte para palavras derivadas, por exemplo *query* e *queries* ;
- Não ordenam os resultados por ordem de relevância;
- Habitualmente são lentos, porque não têm suporte para índices.

A pesquisa de texto resolve este problema através do pré-processamento dos documentos e a criação de um índice para efetuar pesquisas eficientes. O pré-processamento está dividido em três fases:

Parsing* do documento em *tokens Esta fase é útil para identificar diferentes tipos de classes, por exemplo números, palavras, endereços de email, etc;

Conversão dos *tokens* em *lexemes* Um *lexeme* é uma *string*, semelhante a um *token*, que foi transformada para diferentes formas de uma palavra serem semelhantes (por exemplo, remoção dos sufixos que tornam uma palavra plural). Nesta fase também são removidas as palavras de uso comum, como "a"ou "de";

Armazenar os documentos otimizados para pesquisa Cada documento pode ser representado por um *array* de *lexemes* juntamente com a sua posição aproximada no documento.

Também é possível efetuar a pesquisa sem fazer o pré-processamento. Segue-se um exemplo de uma pergunta do género:

```
SELECT title
FROM articles
WHERE to_tsvector(body) @@ to_tsquery('query');
```

Esta pergunta obtém os títulos de todos os artigos que tenham no corpo palavras semelhantes a *query* (*query*, *queries*, *queried*, *querying*, etc).

7.2 Suporte de XML

O PostgreSQL tem várias funções que permitem manipular e criar ficheiros XML. Para ativar estas funções, é necessário instalar o PostgreSQL com a opção `configure --with-libxml`.

O PostgreSQL também tem à disposição do utilizador uma função que permite realizar perguntas XPath, usando o comando:

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

Caso um utilizador pretenda armazenar documentos ou fragmentos XML na base de dados, o PostgreSQL tem implementado o tipo de dados `xml`.

Para criar um documento (ou fragmento) XML a partir de uma string de caracteres, há a função XMLPARSE, que tem a seguinte estrutura:

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

O parâmetro `DOCUMENT` e `CONTENT` indica se a *string* `value` é um documento XML completo ou apenas um fragmento.

A operação inversa, ou seja, converter um documento ou fragmento XML numa sequência de caracteres, pode ser realizada da seguinte forma:

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

7.3 Linguagens Procedimentais

Para além de SQL e C, o PostgreSQL também permite escrever funções numa miríade de outras linguagens. As linguagens disponíveis na distribuição padrão do PostgreSQL são PL/pgSQL, PL/Tcl, PL/Perl e PL/Python.

Um utilizador também pode acrescentar suporte a outras linguagens através de *handlers* escritos em C. Um *handler* tem a responsabilidade de interpretar e executar a linguagem.

O PostgreSQL também tem suporte oficial para as API ODBC e JDBC.

7.4 Segurança

O controlo de acessos a uma base de dados PostgreSQL está configurado no ficheiro `pg_hba.conf`. Cada linha deste ficheiro indica as permissões de um utilizador com base no seu endereço.

O PostgreSQL suporta vários métodos para autenticar um utilizador: Trust, Password, GSSAPI, SSPI, Kerberos, Ident, Peer, LDAP, RADIUS, Certificate e PAM.

7.5 Suporte Objeto-Relacional

O PostgreSQL é um sistema de gestão de base de dados objeto-relacional. Isto significa que tem suporte para mecanismos como objetos, herança de tabelas, definição de tipos de dados personalizados, entre outros.

7.6 Ferramentas

A distribuição de PostgreSQL contém vários utilitários que são instalados juntamente com o SGBD. Algumas dessas ferramentas são:

psql Terminal interativo do PostgreSQL. Permite, através da linha de comandos, escrever perguntas SQL, enviá-las ao PostgreSQL e ver os resultados;

pg_dump Extrai a base de dados para um *script* ou ficheiro de arquivo;

pg_restore Restaura a base de dados com base num ficheiro de arquivo criado pelo `pg_dump`;

pg_config Imprime parâmetros de configuração da versão de PostgreSQL instalada;

createlang Permite adicionar ao PostgreSQL uma nova linguagem procedimental;

droplang Remove uma linguagem procedimental do PostgreSQL.

Para além destes utilitários, uma ferramenta regularmente utilizado é o `pgAdmin`. O `pgAdmin` é uma interface gráfica para o utilitário `psql` que permite gerir, desenvolver e administrar uma base de dados PostgreSQL, semelhante à aplicação SQL Developer usada no Oracle.

Bibliografia

[1] Abraham Silberschatz, Henry F. Korth, S. Sudarshan *Database System Concepts 6th Edition*. McGraw Hill, 2011

[2] PostgreSQL 9.3.4 Documentation, `http://www.postgresql.org/docs/9.3/static/index.html`

[3] PostgreSQL Wiki, `https://wiki.postgresql.org/wiki/Main_Page`

[4] Wikipédia - PostgreSQL, `http://en.wikipedia.org/wiki/PostgreSQL`

[5] Efficient Use of PostgreSQL Indexes, `https://devcenter.heroku.com/articles/postgresql-indexes`

[6] Wikipédia - Linear hashing, `http://en.wikipedia.org/wiki/Linear_hashing`

[7] Wikipédia - GiST, `http://en.wikipedia.org/wiki/GiST`

[8] Gin for PostgreSQL, `http://www.sai.msu.su/~megera/wiki/Gin`