SISTEMAS DE BASE DE DADOS

Mestrado Integrado em Engenharia Informática

Maio 2014

GRUPO 19

PEDRO SOMSEN MIEI 41847 P3

JOÃO LIBÓRIO MIEI 41863 P2

FRANCISCO PINHEIRO MIEI 44297 P2

DOCENTES: JOSÉ ALFERES & RICARDO SILVA

Índice

Capítulo 1

Introdução

Este projecto tem como objectivo estudar um Sistema de Gestão de Base de Dados, neste caso, o *PostgreSQL*.

Sobre este SGBD irão ser abordados capítulos sobre a matéria leccionada, ao longo do ano, nomeadamente: gestão de armazenamento, estruturas de indexação, mecanismo de optimização de perguntas, gestão de transacções e controlo de concorrência e ainda base de dados distribuídas.

1.1 Introdução Histórica do Sistema

O PostgreSQL evoluiu de um projecto chamado *Ingres*, começado na *University of California* em *Berkeley,* 1985, desenvolvido pelo Professor Michael Stonebraker. O professor tentou comercializar o *Ingres* abandonando a faculdade. Este não teve muito sucesso pelo facto de não conseguir compreender tipos e relações, voltando assim para a faculdade onde deu início ao famoso projecto *Postgres*. Este foi nada mais, nada menos, do que uma evolução do *Ingres* onde foram corrigidos os seus pontos fracos. Várias versões foram lançadas sem grandes alterações, até que, o projecto foi abandonado pela faculdade.

Anos mais tarde, em 1994, dois alunos da *University of California* deram início a uma nova versão do *Postgres*, adicionando-lhe um interpretador de SQL que vinha a substituir a linguagem de interrogação QUEL (proveniente do Ingres). *Postgres* passou assim a ser chamado de *Postgres95*.

Pelo facto de o Postgres95 ser capaz de suportar a linguagem SQL, foi então que em 1997 mudou novamente de nome e passou a der denominado por PostgresSQL.

Hoje em dia, este sistema de base de dados tem uma utilização bastante frequente em projectos de pequena dimensão, principalmente por ser *Open-Source*. Caracteriza-se por ser um sistema bastante robusto e completo, oferecendo um grande leque de funcionalidades, tais como, chaves estrangeiras, *triggers*, vistas, integridade transaccional, controlo de concorrência, consultas complexas e ficheiros de indexação.

Capítulo 2

Armazenamento e Estrutura de Ficheiros

O armazenamento é um factor importante a considerar no desempenho de uma base de dados pois uma das operações mais demoradas em termos de tempo é a procura (colocar o disco na posição certa para ler um determinado bloco do disco) para ir buscar o bloco certo a disco e a leitura.

Um bom esquema de indexação permite melhorar o desempenho, no entanto se os dados escritos estiverem espalhados pelo disco para ler uma tabela poderá ser necessário fazer várias procuras que vão piorar a *performance*.

Nesta secção vai-se abordar a gestão de *buffers*. Veremos como o *PostgreSQL* define quais os blocos que devem sair de memória quando é necessário ir buscar outros, qual o *file system* usado e como são guardados os ficheiros em disco, a estrutura interna desses ficheiros e se *multi table clustering* é suportado e como são feitas as partições de relações com um grande tamanho.

2.1 Gestão de *Buffers*

O *PostgreSQL* utiliza um sistema de gestão de *buffers* próprio. A sua função é controlar quais os blocos em RAM de forma a melhorar a *performance* das *queries*. Para isso, usa um algoritmo baseado no LRU (*clocksweep*) e três variáveis: *pin*, *dirty* e *usagecount*. *Pin* indica que um processo está a aceder a essa entrada. *Dirty*

indica se existem alterações que têm de ser escritas em disco. *Usagecount* é uma variável com valores entre 0 e 5 e representa a utilização que esse bloco tem. Para aceder a um bloco é necessário "obter um *pin*", ou seja, incrementar a variável caso não esteja já no máximo, e obter um *lock*, que pode ser partilhado ou exclusivo. Esses *locks* não vão ser aprofundados nesta análise.

Dependendo da situação existem duas estratégias de substituição de *buffers*, ambas seguindo o algoritmo *clocksweep*. A estratégia normal é aplicar o algoritmo a todo o *buffer*. O *buffer* é simplesmente um *array* com um tamanho definido, e o P*ostgreSQL* trata esse *array* como um *array* circular. Uma variável ("*clock hand*") indica qual a próxima entrada do *buffer* a considerar para ser substituída. Existe também uma lista de *buffers* que são candidatos a ser removidos, que é consultada quando é necessário arranjar espaço. Entradas vazias estão sempre nesta lista, e outras entradas poderiam ser adicionadas, mas actualmente isso nunca é feito.

O algoritmo normalmente seguido para a substituição de entradas no *buffer* é ir à lista de *buffers* candidatos a serem removidos e remove-se a sua primeira entrada. Caso possa ser usada (não está "*pinned*" e *usagecount* é 0) é essa a entrada usada. Caso não possa ser usada é ignorada e passa-se para a próxima entrada na lista, até a lista estar vazia ou se encontrar uma entrada usável.

Se a lista estiver vazia vai-se ver o valor do *clock hand* e incrementa-se o seu valor de forma circular (se o seu valor apontar para a ultima posição do *buffer*, o novo valor vai apontar para a primeira posição) e vê-se se é usável. Caso não possa ser usada, o seu *usagecount* é decrementado em 1 (caso não seja já 0) e vê-se a próxima entrada no *buffer*. Quando se encontrar uma entrada que possa ser substituída, marca-se como estando a ser usada e é a seleccionada para ser substituída pelo novo bloco que é necessário trazer para memória.

Esta estratégia tem o problema de em pesquisas sequenciais de uma relação, em que vários blocos serão necessários, mas provavelmente não vão ser usados outra vez, substituírem blocos que são mais usados. Para esse tipo de *queries* usa-se *buffer ring replacement*. É praticamente igual à estratégia normal de substituição, mas aplica-se apenas a uma pequena parte do *buffer*, especificada para esse efeito.

Especificamente, para pesquisas sequenciais o *PostgreSQL* usa uma área de 256KBs. As vantagens é que menos blocos de outros processos serão substituídos, e como consequência o número de escritas em disco de blocos que possam ter sido modificados enquanto em memória é também reduzida. No entanto, para pesquisas sequenciais resultantes de *UPDATE* ou *DELETE*, por exemplo, esta vantagem é extremamente reduzida. Nestes casos, um bloco que tenha sido modificado não é imediatamente escrito em disco. Em vez disso, essa posição é removida do anel e

outra, escolhida pela estratégia normal, é seleccionada para ser acrescentada ao anel. Assim acaba por ser bastante semelhante em *performance*. Apenas em *scans* de *read-only* se consegue notar uma melhoria da performance significativa.

Existem 2 casos especiais para *buffer ring replacement*: *VACUUM* e escritas em massa (*bulk writes*), actualmente aplicando-se apenas a *COPY IN* e *CREATE TABLE AS SELECT. VACUUM* é uma função que funciona como *garbage collector*. Quando tuplos são actualizados ou removidos, não são removidos fisicamente da tabela. *VACUUM* faz a compactação de espaço removendo esses tuplos. Nesse caso usa uma pesquisa sequencial, no entanto quando encontra um bloco que é modificado não o remove do anel e procura a próxima entrada no *buffer*. Em vez disso as modificações são escritas em disco, limpando o WAL (*write-ahead logging*) para a operação de escrita em disco se necessário. Para escritas em massa usa-se um anel de 16MBs em vez de 256KBs (ou até um máximo de 1/8 do espaço total do *buffer* se 16MBs for superior a isso).

Para melhor a performance, o *PostgreSQL* implementa ainda um *background writer*. A sua função é ir escrevendo blocos em memória modificados em disco, mesmo que não seja necessário remove-los de memória ainda, reduzindo a necessidade de escritas em disco quando se faz uma *query* que necessite de outros blocos e consequentemente aumentando a performance. Essa escrita começa pela entrada apontada pela *hand clock* que indica qual o próximo bloco a ser considerado para substituição, mas não modifica essa variável.

2.2 Sistema e Armazenamento de Ficheiros

O P*ostgreSQL* não implementa um sistema de ficheiros, usando o sistema de ficheiros do SO onde corre. Isso faz com que o sitio onde se decide guardar a base pode ter um impacto significativo na performance. A base de dados é guardada em vários ficheiros.

Fazendo uma abordagem bastante superficial aos sistemas de ficheiros, *journaling file systems* são os mais recomendados, isso porque em caso de um *crash* perde-se menos tempo a verificar a integridade dos ficheiros. Existem desvantagens destes sistemas de ficheiros, nomeadamente o WAL serve uma função semelhante, e portanto alguns registos de escritas no disco acaba por ser redundante e provocar uma perda de performance, no entanto, devido ao *overhead* que registar tudo têm, não só para a base de dados mas para o próprio sistema operativo, normalmente apenas se regista apenas escritas na *metadata*, não escritas de dados nos blocos. Isto vai ser vantajoso para o P*ostgreSQL* porque a base de dados consegue verificar

erros nos blocos, mas não na *metadata* do sistema operativo. Uma análise mais detalhada do impacto dos vários tipos de sistemas de ficheiros poderia ser feita mas não se considerou como o objectivo desta secção.

O *PostgreSQL* guarda cada tabela e índice em ficheiros separados. Relações temporárias são também guardadas em ficheiro se necessário, mas com usando um sistema de nomes diferente. Para além do ficheiro com a tabela e o índice, para cada tabela existem ainda outros 2 ficheiros. Um com o mapa de espaço livre, que indica em que blocos há espaço para inserir tuplos, e um mapa de visibilidade, que indica que páginas não têm "tuplos mortos" (foram removidos ou actualizados, no entanto esses dados não foram removidos fisicamente). Quando uma tabela ou índice atinge um tamanho superior a 1GB o ficheiro é separado em segmentos. Isto não é uma limitação da base de dados, é feito por questões de compatibilidade com sistemas de ficheiros que possam não suportar ficheiros maiores que esse tamanho.

As tabelas são guardadas em páginas (blocos) de 8Kbs, e não é permitido um tuplo ocupar mais que uma página. Em alguns casos isso pode levantar problemas se uma coluna tiver entradas demasiado grandes para que se possa inserir numa única página. Nesse caso haverá um ficheiro extra para uma tabela TOAST (The *Oversized-Attribute Storage Technique*). Pode também haver um ficheiro de inicialização para tabelas ou índices vazios.

2.3 Estrutura de armazenamento de tabelas

As tabelas são guardadas como um *array* de páginas, e os tuplos guardados num *heap*, ou seja, um tuplo não está confinado a uma página. Pode ser posto em qualquer página em que haja espaço, sendo a única limitação que um tuplo não pode estar dividido por duas páginas diferentes. A página em si está dividida em várias secções. Um *header* de 24 bits contêm meta-informação sobre a página, incluindo onde começam as outras secções. Um *array* de apontadores para onde estão guardados os tuplos na página, espaço livre, os tuplos e uma secção reservada para dados específicos de acesso do índice (vazio em tabelas normais). De uma forma prática pode-se ver o *array* de apontadores, espaço livre e os tuplos como uma secção. Quando a página não contêm tuplos existem apenas espaço livre, quando se insere tuplos, insere-se um apontador no início do espaço livre e o tuplo no fim. Os tuplos têm todos a mesma estrutura geral, um *header* de 23 bytes com meta-informação, um *null bitmap* se necessário e opcionalmente um *object id*. O *null bitmap* é apenas usado caso existam atributos com o valor *null*. O *bitmap*

contêm um bit por coluna da relação, 1 indica que o valor não é *null*, enquanto que 0 indica o oposto. No caso deste *bitmap* não estar presente assume-se que não existem valores nulos.

Os tuplos não têm um tamanho fixo. Para resolver esse problema o P*ostgreSQL* usa o *TOAST*. Essa técnica é usada apenas em tipos de dados com tamanho variável, em que os primeiros 32 bits indicam o tamanho da variável (contando com os bits usados para guardar o tamanho). Ao usar o *TOAST* retira-se 2 bits do 32 reservados para o tamanho, diminuindo o tamanho total que a variável pode ter, no entanto ganha-se em opções para guardar variáveis que seriam muito grandes caso contrário. Quando os 2 bits são 0 não foi usado nenhuma técnica do TOAST e é um valor que pode ser lido normalmente. No caso do bit mais significativo (ou menos significativo, depende se o sistema usa *big-endian* ou *little-endian*, vai-se assumir na análise *big-endian*) ser 1, então o tamanho ocupa apenas 1 byte em vez dos 4 bytes normais, e os 7 bits restantes indicam o tamanho da variável. Caso os 7 bits restantes estejam todos a 0, o valor é um apontador para os dados guardados na tabela *TOAST* respectiva. O tamanho desse apontador é dado no 2º byte. Caso o bit mais significativo seja 0 mas o bit seguinte é 1 significa que os dados foram comprimidos. Nesse caso os restantes 30 bits dão o tamanho dos dados comprimidos.

Na tabela *TOAST* segue-se a mesma organização das outras tabelas, e portanto guardar valores que possam ser muito grandes para caber numa página ainda representa um problema. Para resolver isso os dados guardados nessa tabela são divididos em bocados com um tamanho máximo pré-definido. No *PostgreSQL* esse valor está definido de forma a que caibam 4 bocados numa página. Esses bocados são guardados como tuplos, tendo como atributos o *chunk_id*, que identifica o atributo partido, *chunk_seq* que identifica em que posição na sequência esse bocado deve ficar, e *chunk_data* que tem os dados. O apontador para um valor numa tabela *TOAST* vai ter de ter o *id* da tabela, *chunk_id* para ir buscar o valor certo. No *PostgreSQL* o apontador contêm também o tamanho original dos dados guardados e o tamanho guardado, que poderá ser diferente se tiver sido usada compressão. Importante referir que caso os dados tenham sido comprimidos e guardados numa tabela *TOAST* não existe essa informação nos 2 bits reservados para o *TOAST* na entrada da tabela. Essa informação estará apenas no apontador.

O *TOAST* tem 4 opções para guardar valores, que o utilizador pode alterar. *PLAIN* "desactiva" o *TOAST*, deixando os 2 bits de estarem reservados. *EXTENDED* é o *default* para valores que cumpram os requisitos para serem *TOAST*. Nesse caso tenta-se primeiro compressão, e caso ainda seja demasiado grande para caber

numa página usa-se a tabela *TOAST*. *EXTERNAL* desactiva a compressão. Todos os valores que não caibam na página serão guardados na tabela *TOAST*. *MAIN* desactiva guardar valores na tabela *TOAST*. Na prática ainda vai ser usada essa tabela, mas é relegada para ultimo recurso. O método de *TOAST* pode ser alterado para cada coluna em especifico com o comando *ALTER TABLE SET STORAGE*.

O *PostgreSQL* não permite *multi table clustering* em ficheiros.

2.4 Particionamento

O *PostgreSQL* permite partições definidas pelo utilizador através de herança de tabelas. Existe uma tabela "principal", normalmente vazia, e os dados estão distribuídos por várias tabelas filhas. As partições podem ser definidas por *range* ou *list*. *Range* particiona a tabela por valores num atributo. *List* particiona a tabela por valores explicitamente indicados pelo utilizador em que tabela aparecem.

Como especificar partições no *PostgreSQL*:

1) Cria-se a tabela principal. Não se colocam dados nesta tabela e portanto não vale a pena definir índices ou *unique constraints*. Poderão-se definir *check constraints* caso se queira que todas as tabelas filhas as apliquem igualmente também.

2) Criar as tabelas filhas que vão herdar a tabela principal. São logicamente partições da relação, mas acabam por ser apenas tabelas normais. Podem ser criadas com o comando

```
CREATE TABLE name () INHERITS (name-master-table)
```

3) Criar os *check constraints* em cada tabela filha de forma a permitir apenas os valores dessa partição. Não existe uma sintaxe diferente para *range* ou *list*, apenas diferem no tipo de restrições aplicadas.

Por exemplo:

```
CHECK (county IN ('Oxfordshire', 'Buckinghamshire', 'Warwickshire'))
```

seria uma restrição para colocar numa partição por *list*, enquanto

```
CHECK ( outletID >= 100 AND outletID < 200 )
```

seria uma restrição numa partição por *range*. É muito importante no entanto ter a certeza que não existe valores que possam estar em mais que uma partição. Por exemplo, adicionar uma restrição noutra partição em que outletID <= 100.

4) Criar índices nas colunas chave e outros que se queira.

5) Criar um *trigger*, permitindo que se insira tuplos na tabela principal, e de forma transparente sejam inseridos na partição correcta em vez da tabela principal.

É importante verificar que *constraint_exclusion* está activada nas configurações ou as *queries* não vão ser devidamente optimizadas.

Pesquisar em tabelas divididas em partições é simples, usando-se a sintaxe normal. É possível no entanto especificar que se quer procurar procurar em apenas uma tabela com a *keyword ONLY*, ou em todas as tabelas que herdam da tabela em que se está a fazer a procura ao adicionar * ao nome da tabela (este é o comportamento *default* do P*ostgreSQL* e portanto não é necessário normalmente). Esta simplicidade não se aplica aos comandos de inserir ou copiar, em que o *PostgreSQL* não propaga a inserção automaticamente. É necessário um *script* para isso.

É possível uma tabela herdar de várias outras. Nesse caso vai ter as colunas de todas as tabelas, mais as suas próprias. Caso existam colunas com o mesmo nome, as duas colunas são fundidas. Se essas colunas tiverem tipos diferentes ocorre um erro. Uma tabela já criada também pode ser alterada de forma a herdar de outra tabela, mas é preciso que seja compatível. Tem de ter as colunas da tabela principal (pode conter mais colunas para além dessas), com o mesmo nome e o mesmo tipo e ter *check constraints* com o mesmo nome e expressões que a tabela principal. Da mesma forma, a herança de uma tabela pode ser removida após ter sido criada. Alterar colunas ou *check constraints* vai-se propagar para todas as tabelas que herdam da tabela que foi alterada.

Relativamente aos privilégios, fazer uma *query* na tabela principal vai aceder tuplos nas tabelas filhas, no entanto aceder directamente as tabelas filhas não é possível sem permissões específicas para isso.

As limitações desta implementação é que índices e restrições são aplicados a uma tabela, e não a um conjunto de tabelas. Assim é possível ter por exemplo dois

tuplos numa relação particionada com a mesma chave. Isso não deverá acontecer se as restrições que definem que tuplos estão em cada partição forem bem definidos, mas ainda assim é possível acontecer. Aplica-se também a qualquer outro atributo que seja *unique* na relação, mas não tenha sido incluindo nos *check constraints* que definem o *range* ou *list* da partição por exemplo. *Foreign keys* também são aplicadas em apenas uma tabela. Seria necessário especificar em cada tabela que um atributo é uma chave externa.

2.5 Oracle vs PostgreSQL

O Oracle implementa também um sistema de gestão de *buffers* próprio com um algoritmo proprietário baseado em LRU. Em relação ao *file system*, é também implementado um DBFS. As tabelas são igualmente organizadas por *heap*, no entanto tem mais opções em relação a partições, permitindo também *hash* e *composite partitioning*. É também possível *multi table clustering.*

Capítulo 3

Indexação e Hashing

A indexação tem um papel extremamente importante nos Sistemas de Gestão de Base de Dados, por permitir aceder a dados armazenados nos discos de uma forma mais eficiente, sem que seja necessário percorrer sequencialmente todos os tuplos de uma tabela (*sequencial scan*), quando é feita uma interrogação à base de dados.

Uma analogia bastante comum da indexação são os índices existentes nos livros. Quando se procura uma palavra ou tema específico num livro, utiliza-se o índice para se saber em que página está o que se pretende em vez de se percorrer o livro todo e na pior das hipóteses nem existir o que se procura.

Os comandos no PostgreSQL para a criação, alteração e remoção de um índice:

Criação

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

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

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

    [ TABLESPACE tablespace ]

    [ WHERE predicate ]
```

Remoção

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

Alteração

```
ALTER INDEX [ IF EXISTS ] name RENAME TO new_name

ALTER INDEX [ IF EXISTS ] name SET TABLESPACE tablespace_name

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

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

Também existe um comando que garante que alguma alteração num índice seja feita com sucesso. Para isso existe o comando:

```
ALTER INDEX [ IF EXISTS ] name RENAME TO new_name

ALTER INDEX [ IF EXISTS ] name SET TABLESPACE tablespace_name

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

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

O PostgreSQL após a criação de um índice encarrega-se de manter o sincronismo com as tabelas em que atributos têm uma estrutura de indexação associada. Compete a este SGBD decidir se compensa utilizar índices em *queries* ou fazer um *sequencial scan*.

Apesar de os índices trazerem muitas vantagens, é importante usá-los com alguma precaução, pois podem gerar *overhead*.

3.1 Estruturas de Dados

O PostgreSQL disponibiliza vários tipos de índices, tais como: B-tree, Hash, GiST, SP-GiST e GIN.

- B-tree

- Por defeito, quando se executa o comando de criação de um índice, é criada uma B-tree por ser a estrutura que é mais vantajosa nas diversas *queries*. Normalmente está associada aos operadores: <, <=, =, >=, >. Também além destes, é usada nas condições *IN*, *BETWEEN*, *IS NOT NULL* e *IS NULL*.

- Hash

- Este tipo de índice está normalmente associado a queries em que as condições contêm apenas o operador =. Uma vez que este índice está implementado em *linear hashing*, corresponde a uma estrutura completamente dinâmica não necessitando de ser optimizada periodicamente.

Comando para criação de um índice Hash:

```
CREATE INDEX name ON table USING hash (column);
```

- ***G**eneralized **S**earch **T**ree* (GiST)

- Este tipo de índice é baseado em árvores permitindo a implementação de várias estratégias de indexação (*operator class*). É importante referir que estamos perante um índice com perdas, podendo produzir resultados não pretendidos. O PostgreSQL já inclui *operator classes* para vários tipos geométricos bidimensionais de dados, tais como, <<, &<, &>, >>, <<|, &<|, |&>, |>>, @>, <@, ~= e &&. O GiST possibilita ainda pesquisas de *neares-neighbor*.

Comando para criação de um índice GiST:

```
CREATE INDEX name ON table USING gist(column);
```

- ***S**pace **P**articionaed **G**eneralized **S**earch **T**ree* (SP-GiST)

- É um tipo de índice bastante semelhante ao acima referido que permite o desenvolvimento de tipos de dados customizados e oferece uma infra-estrutura que suporta várias *operator classes*. Estas estruturas são implementadas em blocos de

disco não equilibrados, tais como, quad-trees, k-d trees, and radix trees. Suporta os seguintes tipos de operadores: <<, >>, ~=, <@, <^ e >^.

- *Generalized Inverted Index* (GIN)

- Ao contrário de GiST, este tipo de índice não tem perdas. No entanto, a sua eficiência depende logaritmicamente do número de palavras distintas. Normalmente está implementado para índices que têm mais do que uma chave, como por exemplo, os *arrays*. Os operadores associados a este tipo de índice são <@, @>, = e &&.

Existem várias comparações entre o GIN e o GiST:

- O GIN é cerca de três vezes mais rápido a pesquisar que o GiST;
- O GIN demora cerca de três vezes mais tempo a ser criado que o GiST;
- Os índices GIN são ligeiramente mais lentos a ser actualizados que os índices GiST;
- Os índices GIN ocupam cerca de duas a três vezes mais espaço que os índices GiST.

Comando para a criação de um índice GIN:

```
CREATE INDEX name ON table USING gin(column);
```

3.2 Índices Multicoluna

Actualmente apenas os índices B-tree, GiST e GIN suportam índices multicoluna. O limite de colunas que se pode criar um índice é 32 (mas é possível trocar este valor no ficheiro pg_config_manual.h).

A utilização deste tipo de índices tem que ser de acordo com o tipo das queries feitas à base de dados em questão, pois a muitas vezes não compensa a sua criação. Nestes casos o planeador dá prioridade a pesquisas sequenciais.

Nas B-tree a ideia é que o índice tenha um operador de igualdade nas colunas mais à esquerda (primeiros argumentos), de maneira a restringir a porção de pesquisa.

3.3 Índices Parciais

É possível definir uma estrutura de indexação para apenas um subconjunto do domínio de um atributo.

Um exemplo de um comando para criar índices parciais:

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)

WHERE NOT (client_ip > inet '192.168.100.0' AND

           client_ip < inet '192.168.100.255');
```

3.4 Índices para Organização de Ficheiros

Apesar da organização de ficheiros no PostgreSQL não estar organizada consoante os índices (*non-clustered*), existe um comando que faz com que os ficheiros se organizem pelo índice.

```
CLUSTER [VERBOSE] table [USING index_name]
```

No entanto, caso seja feitas alterações sobre tuplos destas tabelas, é normal que tenha que ser executado novamente o comando acima referido.

3.5 Índices e Ordenação

Os índices podem ser usados para fazer uma ordenação caso o atributo pelo que se vai ordenar esteja indexado. Caso contrário, faz-se uma pesquisa sequencial e ordena-se explicitamente.

Entre estas duas abordagens, apenas é vantajoso o uso de índice para ordenação caso não se tenha que percorrer a tabela toda, isto é, a *query* use o termo LIMIT n para limitar o número de tuplos a serem apresentados. Caso contrário, se se irá apresentar todos os tuplos da tabela, a ordenação explícita acaba por ser mais eficaz pois existem menos operações de I/O.

Exemplo de uma *query* com a cláusula *ORDER BY* e *LIMIT*:

```
SELECT select_list
    FROM table_expression
    [ ORDER BY ... ]
    [ LIMIT { number | ALL } ] [ OFFSET number ]
```

3.6 Estruturas Temporariamente Inconsistentes

Uma vez que este SGBD permite concorrência, como irá ser abordado mais à frente, é necessário ter um controlo de acesso a estruturas de índices.

Apesar de existir a possibilidade de fazer *LOCK* para controlar o acesso às estruturas de dados, actualizações concorrentes podem gerar inconsistências entre a estrutura do índice e a tabela. Por este motivo e também pelo facto de ser permitido deferir a verificação de restrições de integridade no final das transacções, é necessária a existência de estruturas temporariamente inconsistentes.

3.7 Oracle vs PostgreSQL

Dos vários tipos de índices apresentados para o PostgreSQL, o SGBD Oracle apenas suporta o índice B-tree. Em relação aos seguintes pontos, todos eles são suportados pelo Oracle.

Capítulo 4

Processamento e Optimização de Perguntas

Neste capítulo vai ser explicado um pouco como funciona a estrutura interna do *backend* do PostgreSQL, para ter uma ideia de como funciona o processamento de uma query. Vamos analisar um pouco as operações que ocorrem no *backend*, desde o momento em que a *query* é recebida até ao momento em que o resultado é apresentado ao cliente.

4.1 Processamento de uma Query

Após a introdução de uma *query* SQL, esta é processada por uma série de passos de forma a obter um resultado. Todo este processo pode ser dividido em cinco fases:

Transmition – Após estabelecida a conexão com o servidor, a aplicação transmite uma *query* e aguarda que o resultado seja processado e enviado de volta.

Parser stage – Verifica se a *query* que está a ser transmitida contém algum erro de sintaxe. Se estiver sintacticamente correcta, o *parser* transforma a *query* numa *query tree*. A *query tree* é uma representação interna da instrução SQL. Esta instrução é repartida tendo em consideração todas as regras, interpretações semânticas, funções e operadores e junção, de forma a compreender melhor o seu significado formal.

Rewrite system – Procura por alguma regra (guardada no *system catalogs*) que possa ser aplicada à *query tree* gerada na *Parser stage*. Quando estamos a aplicar uma *query* a uma *view*, este sistema pode reescrever a *query* efectuada pelo utilizador de maneira a que esta aceda às tabelas base definidas na *view*.

Planner/optimizer – A partir da *query tree* reescrita, obtida na fase anterior, vai ser criado o plano de execução da *query* que servirá de entrada para a fase seguinte, o *Executor*. Para obter este plano de execução, inicialmente são gerados todos os caminhos possíveis para o mesmo resultado. Na situação em que existe mais do que um caminho possível, é calculado o custo estimado e escolhido o caminho com o custo mais baixo. O caminho mais barato é expandido para um plano completo, para ser usado pelo Executor.

Executor – Percorre o plano de execução para ir buscar os tuplos da forma como está representado no plano. Isto é essencialmente um mecanismos de *demand-pull pipeline*. Cada vez que um nó do plano é chamado, tem que devolver um ou mais tuplos, ou dizer que já não vai devolver mais tuplos.

4.2 Operações Básicas

4.2.1 Algoritmos de Selecção

Sequential Scan – É o algoritmo de selecção mais pesado, em que tem que se percorrer todos os tuplos da tabela e verificar individualmente se a condição de selecção é verificada. Funciona para todos os tipos de selecções e não necessita de estruturas auxiliares para funcionar.

Index Scan – Necessita que exista um índice, no atributo de junção, previamente criado. Desta forma, é possível percorrer todos os índices para obter os tuplos que verificam a condição de selecção. Os tuplos são devolvidos pela ordem do índice e não pela ordem em que estão guardados no disco. Isto acontece porque no PostgreSQL os dados não estão guardados no disco pela ordem do índice. Significa que às vezes pode não ser tão vantajoso utilizar este algoritmo em comparação com o *Sequential Scan*, visto que têm que ser efectuadas muitas operações de I/O.

Index-only scan – Semelhante ao índex scan com a vantagem de não necessitar de aceder à tabela física para obter os dados.

Bitmap index scan – Necessita que exista um bitmap que é uma estrutura, composta por bits, em que cada bit representa se um tuplo tem um valor de um

determinado atributo. Ou seja, para cada valor distinto do atributo é necessário ter um bitmap que indica os tuplos que contêm esse valor. Uma vantagem de usar este tipo de selecção é que nas operações como o COUNT basta contar o número de bits para obter o resultado. Torna-se complicado usar estas estruturas quando os atributos têm muitos valores distintos.

4.2.2 Algoritmos de Junção

Nested Loop join – Consideremos a tabela do "lado esquerdo" como **outer table** e a tabela do "lado direito" como **inner table**. Neste algoritmo, por cada tuplo na tabela do lado esquerdo, é feita a junção para todos os tuplos na tabela do lado direito, caso a condição da junção se verifique. Este processo é repetido para todos os tuplos na tabela do lado esquerdo.

Merge join – Cada relação é ordenada pelo atributo de junção, onde depois é aplicada uma exploração em paralelo às duas tabelas em que se vai verificando as condições da junção.

Hash join – Para este caso, é feito uma análise à tabela do lado direito e é aplicada uma função de *hash* ao atributo de junção, obtendo uma tabela de hashing. O mesmo é feito para cada tuplo da outra tabela, e se o valor do *hash* estiver contido na tabela de hashing previamente calculada, é efectuada a junção desses dois tuplos.

4.2.3 Algoritmos de Ordenação

Quicksort – Este algoritmo é utilizado quando ambas as tabelas cabem em memória, aplicando recursivamente dois passos. Em primeiro lugar, escolhe um tuplo pivot da tabela. Em segundo lugar, ordena todos os tuplos de maneira a que aqueles com menores valores apareçam primeiro e aqueles com maior valor apareçam depois.

External sort-merge – No caso em que as tabelas não cabem em memória, este é o algoritmo utilizado. A tabela é partida em pedaços que cabem em memória, é feita uma ordenação e são guardados num ficheiro temporário. De seguida, os ficheiros temporários são combinados até se formarem num único ficheiro, que será a tabela final ordenada.

4.3 Expressões Complexas

Na maior parte das vezes, o *PostgreSQL* utiliza *pipelining*. Quando isto não é possível ou simplesmente não compensa, o resultado é guardado em disco. Por exemplo, no caso do *merge join* ou *hash join* não é possível usar *pipelining*, então os dados têm que ser materializados, assumindo que estes têm que ser ordenados.

4.4 Visualização de Planos e Estimativas

4.4.1 EXPLAIN

O *PostgreSQL* imagina um plano de execução para cada *query* que recebe. Escolher o melhor plano correspondente à estrutura da *query* e às propriedades dos dados é crítico para uma boa performance, por isso, o sistema inclui um *planner* complexo que tenta escolher bons planos. O comando EXPLAIN permite consultar o plano criado pelo *planner*.

Um exemplo trivial da execução do comando:

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

Como a *query* não contém nenhuma cláusula *WHERE*, tem que efectuar uma pesquisa por todos os tuplos da tabela, por isso o *planner* escolheu usar um *sequential scan* para o plano. Os números dentro dos parênteses são:

Custo inicial estimado: É o tempo gasto antes do resultado ser apresentado.

Custo total estimado: Isto é apresentado assumindo que todas as linhas são utilizadas, na prática podemos parar a meio da leitura (por exemplo usando o comando *LIMIT*).

Estimação do número de linhas por plano: Novamente, assumindo que todas as linhas são utilizadas.

Estimação do tamanho médio das linhas: Tamanho em *bytes* das linhas para cada nó do plano.

4.4.2 EXPLAIN ANALYZE

É possível verificar a precisão das estimativas efectuadas pelo *planner/optimizer* usando o comando *EXPLAIN ANALYZE*. Com esta opção, o *EXPLAIN* executa a *query* e depois apresenta o verdadeiro valor para o número de linhas e para o tempo acumulado gasto em cada nó do plano, juntamente com as outras estimativas que já são apresentadas no *EXPLAIN*.

Exemplo:

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;


                                QUERY PLAN
-------------------------------------------------------------------------
--------
 Nested Loop     (cost=4.65..118.62   rows=10   width=488)   (actual
time=0.128..0.377 rows=10 loops=1)
   ->   Bitmap Heap Scan on tenk1 t1   (cost=4.36..39.47  rows=10
width=244) (actual time=0.057..0.121 rows=10 loops=1)
         Recheck Cond: (unique1 < 10)
         ->   Bitmap Index Scan on tenk1_unique1   (cost=0.00..4.36
rows=10 width=0) (actual time=0.024..0.024 rows=10 loops=1)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91
rows=1 width=244) (actual time=0.021..0.022 rows=1 loops=10)
         Index Cond: (unique2 = t1.unique2)
 Total runtime: 0.501 ms
```

De frisar que o valor do tempo actual está em milissegundos, enquanto os custos estimados estão representados por unidades arbitrárias, isto significa que é improvável que as duas coincidam. O que normalmente é mais importante de ver é se a estimativa do número de linhas está perto da realidade. Neste exemplo em concreto isso não é possível ser visto, mas na prática não é muito comum isto acontecer.

4.5 Comandos de Parametrização

Como o *PostgreSQL* não garante que seja apresentado o melhor plano de execução de uma *query*, é possível utilizar os seguintes parâmetros:

Enable_bitmapscan (boolean) – Activa ou desactiva o uso de *bitmap-scan* no plano de execuções.

Enable_hashagg (boolean) – Activa ou desactiva o uso de agregação por *hashing* no plano de execuções.

Enable_hashjoin (boolean) – Activa ou desactiva o uso de *hash join* no plano de execuções.

Enable_indexscan (boolean) – Activa ou desactiva o uso de *index scan* no plano de execuções.

Enable_indexonlyscan (boolean) – Activa ou desactiva o uso de *index-only scan* no plano de execuções.

Enable_material (boolean) – Activa ou desactiva o uso de materialização no plano de execuções. É impossível impedir inteiramente o seu uso, mas desactivando esta variável o *planner/optimizer* previne a sua utilização, excepto nos casos em que é mesmo necessário.

Enable_mergejoin (boolean) – Activa ou desactiva o uso de *merge-join* no plano de execuções.

Enable_nestloop (boolean) – Activa ou desactiva o uso de *nested-loop join* no plano de execuções. É impossível impedir inteiramente o seu uso, mas desactivando esta variável o *planner/optimizer* é desencorajado a utilizar este método se existem outros disponíveis.

Enable_seqscan (boolean) – Activa ou desactiva o uso de *sequential scan* no plano de execuções. É impossível impedir inteiramente o seu uso, mas desactivando esta variável o *planner/optimizer* é desencorajado a utilizar este método se existem outros disponíveis.

Enable_sort (boolean) – Activa ou desactiva o uso de ordenação no plano de execuções. É impossível impedir inteiramente o seu uso, mas desactivando esta variável o *planner/optimizer* é desencorajado a utilizar este método se existem outros disponíveis.

4.6 Oracle vs PostgreSQL

O *PostgreSQL*, ao fazer junção de tabelas, apresenta mais algoritmos de optimização que o *Oracle*, o que pode prejudicar a sua *performace*. Ou seja, como o *PostgreSQL* suporta muitos mais algoritmos, surgem muito mais combinações quando se está a tentar chegar ao melhor plano de execução. Desta forma, apesar de o *PostgreSQL* conseguir alcançar um melhor plano de execução, este é obtido à custa de tempo. O *Oracle* está preparado para bases de dados de grandes dimensões, daí ter um menos número de algoritmos disponíveis. O *PostgreSQL* investe mais no processamento de grandes consultas, ao contrário do *Oracle*.

O *PostgreSQL* contém mais comandos de parametrização que o *Oracle*, podendo ser vantajoso quando se está a procurar o melhor plano de execução para uma *query*.

Capítulo 5

Gestão de Transações e Controlo de Concorrência

A gestão de transacções e o controlo de concorrência são características da base de dados que afectam bastante a performance e a consistência. Um mau controlo de concorrência pode levar a que *queries* devolvam tuplos ou valores que mostram a base de dados num estado inconsistente, ou a que isso não aconteça, mas as *queries* demorem mais tempo do que o aceitável a serem concluídas.

5.1 Transacções

A noção de transacção existe no *PostgreSQL* como seria de esperar. Cada *query SQL* é tratada como se estivesse dentro de uma transacção, não sendo necessário especificar o início e o fim. No entanto isso não é suficiente em vários caso. Um exemplo clássico é o de retirar dinheiro de uma conta e colocar noutra. Essa operação é feita com várias *queries* e no seu conjunto não são atómicas, apesar de individualmente serem. Para resolver isto o *PostgreSQL* disponibiliza os comandos *BEGIN*, *COMMIT*, *ROLLBACK* e *SAVEPOINT*. Estes comandos podem ser usados explicitamente pelo utilizador, permitindo-lhe definir que blocos de *queries* o *PostgreSQL* deve tratar como uma transacção, criar *checkpoints* a meio da transacção, de forma a não fazer *rollback* de toda a transacção caso não queira, ou fazer *rollback* de todos os comandos executados até á altura.

Não existem *nested transations* no *PostgreSQL*, o comando *savepoint* é usado para o mesmo efeito. Esse comando dá alguma suporte a transacções de longa duração

ao permitir que uma transacção que tenha sido abortada por algum motivo não tenha de ser toda repetida.

5.2 Gestão de Concorrência

O *PostgreSQL* usa o *MVCC* (*Multiversion Concurrency Control*) para gerir a concorrência dos acessos e garantir o isolamento das transacções. É implementado através do *SSI* (*Serializable Snapshot Isolation*) que cria um *snapshot* da base de dados. Isso traz vantagens em relação a *locks* porque operações de leitura não bloqueiam escrita e escrita não bloqueia a leitura, aumentando a concorrência e a performance. Uma desvantagem é que ocupa mais espaço. Sempre que há uma operação de escrita é necessário criar uma nova versão do objecto que está a ser escrito. O *PostgreSQL* lida com isso através da função *VACUUM*, que pode ser chamada explicitamente pelo utilizador, e é executada automaticamente pela base de dados. No entanto há situações em que o utilizador pode querer usar *locks* explicitamente. O *PostgreSQL* dá essa possibilidade também.

5.3 *Locks* e Níveis de Granularidade

O *PostgreSQL* usa 4 tipos de *locks*: *table-level*, *row-level*, *advisory lock* e *page-level*. Dentro destes tipos existem diferentes modos. *Advisory lock* é um tipo de especial de *lock* em que é definido pelo utilizador e a base de dados não o garante. É da responsabilidade da aplicação garantir, caso queira, que o *lock* é efectuado. Para os outros *locks* o *PostgreSQL* utiliza-os automaticamente, mas é ainda possível o utilizador explicitar o *lock* que quer usar com o comando LOCK, com excepção de *page-level* que é usado automaticamente apenas. *Locks page-level* são mais curtos que os outros, sendo usados apenas para ler ou escrever um tuplo no bloco, e assim que essa operação é concluída são imediatamente libertados. É possível ver os *locks* actualmente usados através de uma vista de sistema.

Os *locks* têm uma granularidade pré-definida, no entanto esta é variável e é o maior factor distinguido os diferentes tipos de *locks*, com excepção do *advisory lock*. *Table-level* aplica o *lock* a uma tabela, *row-level* aplica o *lock* a um tuplo, e *page-level* aplica o *lock* a uma página (tipicamente um bloco).

Para *locks table-level* existem 8 modos de utilização. Duas transacções não podem ter *locks* com modos incompatíveis na mesma tabela, no entanto não há limitações a quantas transacções podem ter *locks* numa tabela se os seus modos não forem incompatíveis. Quando um *lock* é adquirido numa transacção normalmente só é

libertado quando a transacção termina, com excepção de um *rollback* para um *savepoint*. Nesse caso a transacção ainda não terminou, mas o *lock*, se tiver sido adquirido após o *savepoint*, é libertado.

Breve descrição de quando se usa os vários modos de *locks table-level* e os seus conflitos:

1) *Access share*: Usado para operações de apenas leitura. Conflitos: *access exclusive.*

2) *Row share*: Usado normalmente em *SELECT FOR UPDATE* e *SELECT FOR SHARE*. Apesar do nome, é aplicado a toda a tabela. Conflitos: *exclusive* e *access exclusive.*

3) *Row exclusive*: Usado normalmente por qualquer operação que vai modificar dados na tabela. Tal como o modo anterior, aplica-se à tabela toda. Conflitos: *share*, *share row exclusive*, *exclusive*, *access exclusive*.

4) *Share update exclusive*: Usado normalmente pelo *VACUUM*, *ANALYSE*, *CREATE INDEX CONCURRENTLY*, e alguns comandos que alteram o esquema das tabelas. Foi especificado para impedir mudanças concorrentes no esquema de uma tabela. Conflitos: *share update exclusive*, *share*, *share row exclusive*, *exclusive*, *access exclusive*.

5) *Share*: Usado normalmente pelo *CREATE INDEX*. Protege uma tabela de alterações concorrentes de dados. Conflitos: *row exclusive*, *share update exclusive*, *share row exclusive*, *exclusive* e *access exclusive*.

6) *Share row exclusive*: Semelhante ao modo *share*, a diferença é que entra em conflito consigo mesmo, ou seja, apenas uma *transação* o pode adquirir. Conflitos: *row exclusive*, *share update exclusive*, *share*, *share row exclusive*, *exclusive* e *access exclusive*.

7) *Exclusive*: Faz com que apenas operações *read-only* possam ter acesso à tabela. Conflitos: *row share*, *row exclusive*, *share update exclusive*, *share*, *share row exclusive*, *exclusive* e *access exclusive*.

8) *Access exclusive*: Faz com que apenas a transacção que detém este *lock* possa aceder à tabela. É o *default* para comandos *LOCK TABLE* e é usado por *ALTER TABLE*, *DROP TABLE*, *TRUNCATE*, *REINDEX*, *CLUSTER* e *VACUUM FULL*. Conflitos: todos os outros modos.

Locks row-level tem apenas 2 modos, *exclusive* e *shared*. Apenas uma transacção pode ter um *lock* exclusive num tuplo, e esse *lock* é mantido até a transacção terminar. Várias transacções podem ter *locks shared* no mesmo tuplo, mas nenhuma pode o pode alterar. Caso o queira, precisa de obter um *lock exclusive*, e apenas o conseguirá obter quando não houver nenhum *lock shared* nesse tuplo.

O facto do *PostgreSQL* permitir ao utilizador explicitar os *locks* usado aumenta a probabilidade de ocorrer um *deadlock*. O *PostgreSQL* consegue detectar estas situações e para as resolver aborta uma das *transações* que está a causa o *deadlock*. Não é especificado qual a transacção abortada.

Advisory locks são usados para representar situações que não devem de acontecer, mas são difíceis de especificar no controlo de concorrência. Pode-se obter estes *locks* para a sessão ou para uma transacção. No caso de se adquirir para a sessão são mantidos até a sessão terminar ou serem explicitamente libertados. No caso de ser para a transacção o *lock* é libertado quando a transacção acaba, não sendo necessário liberta-lo explicitamente.

5.4 Níveis de Isolamento

No *PostgreSQL* é possível pedir os 4 níveis de isolamento do *SQL* através do comando SET TRANSITION, no entanto internamente apenas são usados 3. Os níveis de isolamento implementados são *read committed*, *repeateable read* e *serializable*. Quando se selecciona *read uncommited*, o nível que se vai obter é *read commited*. Para além disso, devido à implementação do *repeatable read*, esse nível não permite *phantom reads*, que no standard *SQL* é permitido nesse nível. *Phantom read* acontece quando uma transacção executa uma *query* que já tinha feito anteriormente e o conjunto de tuplos devolvidos é diferente da 2ª vez por terem sido modificados por uma transacção concorrente.

Read commited é o nível de isolamento *default* do *PostgreSQL*. Nesse nível uma *query* apenas vê dados cometidos antes de começar a sua execução. No entanto uma *query* irá ver os dados alterados dentro da sua *transação*, apesar de ainda não ter sido cometidos. É também possível que duas *queries* sucessivas leiam dados diferentes caso uma transacção concorrente faça *commit* após a primeira *query* começar, mas antes da segunda.

No caso de a *query* ser *UPDATE*, *DELETE*, *SELECT FOR UPDATE* ou *SELECT FOR SHARE* existe o problema adicional de o tuplo ter sido alterado depois de a *query* começar e até o tuplo ser encontrado. Nesse caso a *query* espera que a transacção concorrente faça *commit* ou *rollback* caso ainda não tenha terminado. No caso de um *rollback* continua normalmente, no caso de um *commit* ignora no caso de ter sido apagado, ou tenta aplicar as suas operações no tuplo, reavaliando se o tuplo ainda tem os requisitos para isso se aplicável.

É possível que uma *query* que tente alterar tuplos veja a base de dados num estado inconsistente pois alguns tuplos já poderão ter sido modificados e outros não quando são encontrados. Isso faz com que este nível de isolamento não seja adequado para *queries* com instruções complexas.

Repeatable read é um nível de isolamento mais forte que *read commited*. Neste nível uma *query* apenas vê os dados cometidos antes da sua transacção começar. Uma *query* pode no entanto ver alterações nos dados feitas dentro da sua transacção, apesar de ainda não terem sido cometidos.

No caso de *queries* que alterem os tuplos existe um comportamento diferente. É possível que uma *query* encontre um tuplo que já foi alterado por uma transacção concorrente (mais provável que em *read commited*, pois em *read commit* considera-se o estado dos tuplos quando a *query* começou, e em *repeatable read* quando a transacção começou, portanto mais tempo poderá ter passado). Nestes caso a transacção apenas continua no caso da *transação* concorrente ter feito *rollback*. No caso do tuplo ter sido alterado ou apagado a transacção falha e tem de se tentar executa-la toda outra vez. Da segunda vez que tenta vai ver o novo estado dos tuplos, e não entrará em conflito com a outra transacção. Mas poderá ainda ocorrer conflitos com outras transacções concorrentes.

Neste nível de isolamento as *queries* irão ver sempre um estado consistente da base de dados.

Serializable é o nível de isolamento mais estrito. É simulada a execução das transacções em série, apesar de não ser essa a sua implementação. Na prática, *serializable* é implementado da mesma forma que *repeatable read*, mas com uma

condição adicional. São verificadas condições que tornam a execução concorrente das transições inconsistentes com todos os esquemas de execução em série dessas transacções. No caso de isso acontecer, a transacção é abortada.

Para verificar essas condições o *PostgreSQL* usa *predicate locks*. No entanto esses *locks* não bloqueiam o acesso a um tuplo. Serve apenas como *flag* para verificar se uma operação de escrita teria impacto numa operação de leitura anterior da transacção. Esses *locks* podem ser visto tal como os outros *locks* numa vista de sistema, e a sua granularidade pode ser alterada durante a transacção, de forma a impedir que toda a memória fique toda ocupada. Por exemplo vários *locks* aplicados a tuplos podem ser combinados num único *lock* relativo ao bloco onde esses tuplos estavam.

Uma transacção que seja apenas de leitura pode libertar esses *locks* antes do *commit* se conseguir ver que já não é possível haver conflitos.

5.5 Consistência e Recuperação

O *PostgreSQL* permite verificar a consistência imediatamente ou apenas no fim da transacção. Na altura da sua criação, cada restrição pode ter um de 3 modos: *DEFERRABLE INITIALLY DEFERRED*, *DEFERRABLE INITIALLY IMMEDIATE*, ou *NOT DEFERRABLE*. Os 2 primeiros podem ser mudados posteriormente, o 3º a verificação é sempre feita no fim da *query*. Restrições do tipo *NOT NULL* e *CHECK* não são afectadas por esta definição, e são sempre verificadas imediatamente. Por *default UNIQUE* e *EXCLUDE* são também verificadas imediatamente e *PRIMARY KEY* e *REFERENCES* são verificadas no fim da transacção, mas este comportamento pode ser mudado com o comando *SET CONSTRAINTS {all | name [,...]} {DEFERRED | IMMEDIATE}*. Se mais uma restrição faça corresponda a *name*, todas irão ser mudadas.

A mudança é retroactiva quando se muda de *deferred* para *immediate*, ou seja, todas as restrições que iriam ser verificadas apenas no fim da transacção serão verificadas imediatamente se for aplicável. Caso alguma restrição falhe, o comando aborta e não é mudada a altura de verificação.

Para garantir a durabilidade da base de dados o *PostgreSQL* usa o *WAL*. As mudanças na base de dados são escritas no *log* antes de serem escritas no disco. Isso tem o efeito de que os dados não precisam de ser escritos em disco para uma transacção ficar cometida, basta que as modificações sejam escritas no *log*, e

consequentemente reduz o número de escritas em disco porque não é necessário escrever as modificações no fim de cada transacção.

O *log* pode ser também usado criar backups da base de dados. Tendo um backup físico e um *log* com as modificações feitas após essa altura, é possível restaurar a base de dados para um qualquer ponto no tempo coberto pelo *log*.

Para diminuir a possibilidade de erros no de escrita no *log*, cada entrada por um valor CRC de 32 bits.

5.6 Comparação com o Oracle

A implementação do Oracle é bastante próxima do *PostgreSQL*, mas com umas ligeiras diferenças. Os comandos para as transacções são diferentes (*commit work* em vez de *commit* por exemplo), mas oferecem essencialmente a mesma funcionalidade. Para resolver *deadlocks* o Oracle aborta a operação bloqueada há mais tempo, no *PostgreSQL* é difícil de prever qual será a abortada.

O Oracle implementa apenas 2 dos níveis de isolamento em comparação com os 4 (3 na prática) do *PostgreSQL*, mas oferece um 3º que é semelhante a outro dos níveis standard do *SQL*. Em termos de granularidade ambos oferecem a possibilidade de aplicar *locks* a tabelas ou a tuplos.

Capítulo 6

Suporte para Bases de Dados Distribuídas

O *PostgreSQL* permite mecanismos de replicação de dados a partir de uma arquitectura Cliente/Servidor, existindo servidores primários denominados por *master servers* capazes de executar operações de escrita e leitura e aqueles que guardam as alterações feitas no *master*, os *standby/slave servers*. Destes, existem servidores que não conseguem ser acedidos até que sejam promovidos a *master*, os *warm standby servers*, e aqueles que aceitam conexões e servem pedidos de leitura, os *hot standby servers,* permitindo assim distribuição de carga.

Esta arquitectura permite alta disponibilidade pelo facto de existirem vários standby servers que a qualquer momento podem assumir o papel de primário, caso o servidor primário falhe. O servidor primário está constantemente em *archiving mode*, enquanto de cado um dos *standby servers* estão em *recovery mode,* lendo os *WAL files* do servidor primário.

Os *WAL files* podem ser propagados ficheiro a ficheiro, tipicamente na ordem dos 16 MB. A esta técnica de replicação chama-se *Log Shipping*. No entanto, os *standby servers* também conseguem obter estes ficheiros através de uma ligação TCP com o servidor primário, *Streaming Replication*.

Capítulo 7

Outras Características do Sistema Estudado

7.1 XML

O *PostgreSQL* suporta tipos de dados XML que traz várias vantagens sobre tipo de dados em texto simples. Aquando a instalação do PostgreSQL, para se ter acesso à biblioteca que tem várias funções de XML, tem que se executar o comando: configure --with-libxml.

Alguns dos comandos possíveis:

Para gerar um documento ou conteúdo em XML a partir de uma string:

XMLPARSE ({ DOCUMENT | CONTENT } value)

Para fazer o inverso:

XMLSERIALIZE ({ DOCUMENT | CONTENT } value AS type)

Entre outros:

SET xmloption TO { DOCUMENT | CONTENT };

7.2 JSON

Para além do XML, o PostgreSQL ainda suporta tipo de dados JSON.

Algumas das funções disponibilizadas são:

json_array_length(json) – retorna o número de elementos do JSON array

row_to_json(record [, pretty_bool]) – retorna uma row no formato JSON

json_each(json) – transforma cada objecto JSON num conjunto valor/chave

Referências

[1] Documentação *PostgreSQL,* http://www.postgresql.org/docs/9.3, 2014

[2] No Nested Transactions, No Transactions in PL/pgSQL, http://sqlblog.com/blogs/alexander_kuznetsov/archive/2013/11/25/learning-postgresql-no-nested-transactions-no-transactions-in-pl-pgsql.aspx, 2014

[3] Documentação *PostgreSQL* sobre *buffers*,

http://git.postgresql.org/gitweb/?p=postgresql.git;a=blob;f=src/backend/storage/buffer/README;hb=HEAD, 2014

[4] *Inside the PostgreSQL Shared Buffer Cache*, http://www.westnet.com/~gsmith/content/postgresql/InsideBufferCache.pdf, 2014

[5] Documentação *PostgreSQL*, https://wiki.postgresql.org/wiki/Main_Page, 2014

[6] *How PostgreSQL Executes a Query*, http://etutorials.org/SQL/Postgresql/Part+I+General+PostgreSQL+Use/Chapter+4.+Performance/Understanding+How+PostgreSQL+Executes+a+Query/, 2014