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

PostgreSQL 9.3

Sistemas de Bases de Dados
Ano letivo 2013/2014

Professor: José Alferes

Trabalho realizado por:

Grupo 10
André Alves nº41867
João Rosa nº41973
Manuel Almas nº42185

Índice

1. Introdução

Neste trabalho é apresentada uma análise ao sistema de base de dados *PostgreSQL*. Nomeadamente é feita uma breve introdução histórica ao mesmo, seguida de uma análise sobre o armazenamento e estrutura de ficheiros, indexação e *hashing*, processamento e otimização de perguntas, gestão de transações e controlo de concorrência, suporte para bases de dados distribuídas e descrição de outras características que considerámos relevantes no contexto do *PostgreSQL*.
Em cada tema abordado é feita a sua comparação com o respetivo tema no sistema de base de dados *Oracle 11g*.

Contexto histórico do sistema

O sistema *PostgreSQL* nasceu de um projeto desenvolvido em ambiente académico na Universidade Berkeley. O projeto passou por várias versões até ser o sistema robusto que conhecemos hoje. Durante as várias atualizações e melhoramentos que foram feitos ao sistema foram sendo adicionadas funcionalidades. Primeiramente começou-se por adicionar um interpretador SQL (Postgres95) e outros detalhes da usabilidade do sistema. De seguida e até chegar ao sistema *PostgreSQL* como o conhecemos hoje, foram essencialmente corrigidos problemas com a implementação do servidor.

Atualmente o *PostgreSQL* é um sistema robusto e completo conhecido por possuir algumas funcionalidades tais como:

- existência de *queries* complexas
- integridade transacional
- existência de *triggers*
- existência de chaves estrangeiras
- existência de *triggers*

O sistema tem adquirido forte reconhecimento na comunidade *open source*, e é utilizado em grandes empresas internacionais. Um outro melhoramento que foi introduzido nas últimas versões do sistema foi a introdução de replicação de dados, suportando assim bases de dados distribuídas.

2. Armazenamento e estrutura de ficheiros

2.1. Organização

O sistema *PostgreSQL* implementa o seu próprio sistema de ficheiros, não dando uso ao do sistema operativo. Mantém também uma lista de diretorias e ficheiros no sistema de ficheiros do SO, para o armazenamento dos ficheiros necessários, os chamados *tablespaces*. Existe uma hierarquia de ficheiros e diretorias, que permitem a simplicidade e clareza na implementação. A diretoria principal PGDATA contém as várias diretorias e ficheiros de controlo. A tabela abaixo apresenta o conteúdo da diretoria PGDATA.

Item	**Descrição**
PG_VERSION	Ficheiro que contém a versão do *PostgreSQL*
base	Subdiretoria que contém todas as subdiretorias por base de dados
global	Subdiretoria que contém as tabelas comuns a todo o *cluster*
pg_clog	Subdiretoria que contém informação sobre o estado das transições
pg_multixact	Subdiretoria que contém informação sobre o estado das multi-transições
pg_notify	Subdiretoria que contém informação sobre o estado (LISTEN/NOTIFY) dos dados
pg_serial	Subdiretoria que contém informação sobre transições serializáveis *commited*
pg_snapshots	Subdiretoria que contém *snapshots* exportados
pg_stat_tmp	Subdiretoria que contém ficheiros temporários usados no subsistema de estatísticas
pg_subtrans	Subdiretoria que contém dados referentes ao estado das sub-transações
pg_tblspc	Subdiretoria que contém links simbólicos para *tablespace*
pg_twophase	Subdiretoria que contém ficheiros de estado para transações preparadas
pg_xlog	Subdiretoria que contém ficheiros WAL (Write Ahead Log)
postmaster.opts	Ficheiro que contém as configurações com que o servidor foi inicializado na última vez
postmaster.pid	Ficheiro que contém o PID corrente e o ID de memória partilhada

2.2. Paginação

No que diz respeito à representação de tabelas, o sistema utiliza um esquema de páginas com tamanho fixo de cada página de 8Kb e não permite que um tuplo se encontre dividido em mais que uma página, impedindo assim que sejam guardados campos com grandes dimensões. As páginas seguem a estrutura do tipo *slotted-page*,

o que permite a organização facilitada dos registos armazenados em cada página. A estrutura de uma página é representada na tabela abaixo.

Item	**Descrição**
PageHeaderData	Localização onde está armazenada informação geral e apontadores para o espaço livre
ItemIdData	Representa um vetor de apontadores para os registos da página
Free space	Representa o espaço livre para novos dados
Items	Representa os registos já guardados na página
Special space	Representa o espaço reservado para acessos aos registos através de índices

2.3. Buffer Management

No *PostgreSQL* o *buffer* é utilizado como um sistema de cache que permite gerir os acessos à informação em disco por parte os vários processos. Cada *buffer* tem associado uma *flag* que pode ter o estados *pinned* ou *unpinned*, consoante o mesmo esteja a ser ou não utilizado pelo sistema. Versões anteriores do *PostgreSQL* funcionavam segundo a política *least-recently-used* (LRU) e possuíam a cache para manter as páginas recentemente referenciadas em memória. No entanto, o algoritmo LRU não tinha em consideração o número de vezes que uma determinada entrada na cache era acedida, dessa forma se fossem efetuadas pesquisas extensas nas tabelas, poderiam ir parar à cache páginas desnecessárias. Dessa forma o sistema passou a implementar um algoritmo de cache que utiliza quatro listas separadas para detetar as páginas que foram acedidas recentemente mais vezes e dessa forma optimizar dinamicamente a sua substituição. Este novo algoritmo leva a um uso mais eficiente do *buffer* partilhado.

2.4. TOAST

Uma vez que o *PostgreSQL* utiliza um tamanho de página fixo (8Kb) e que não é permitido estender os tuplos por várias páginas, o sistema utiliza uma técnica chamada TOAST que é aplicada somente a tipos de dados que possam produzir valores de campos muito grandes, armazenando-os numa área secundária *out-of-line*. Esta técnica causa somente impacto na componente servidor, sendo totalmente transparente para o utilizador.

O sistema trata todos valores de entrada de forma TOAST, recorrendo à

chamada *PG_DETOAST_DATUM* para cada valor. De modo a suportarem o TOAST, os tipos de dados possuem um tamanho de representação variável, *varlena* de 32 bits em que utiliza os seus dois bits mais significativos por forma a limitar o tamanho dos dados a 1 Gb (2^30-1bytes). A definição ou não dos dois bits mais significativos permite concluir se o tipo de dados foi comprimido e ou armazenado *out-of-line*.

Quando alguma coluna da tabela é dividida, existe uma tabela *TOAST* associada, cujo OID é armazenado na entrada *pg_class.reltoastrelid* da tabela, sendo este o resultado obtido (excerto) se executarmos o seguinte comando: "SELECT relname, reltoastrelid FROM pg_class".

	relname name	reltoastrelid oid
1	pg_statistic	2840
2	pg_type	0
3	pg_toast_2619	0
4	pg_toast_2619_index	0
5	pg_authid_rolname_index	0
6	pg_authid_oid_index	0
7	pg_attribute_relid_attn	0
8	pg_attribute_relid_attn	0

Os valores divididos *out-of-line* são mantidos na tabela *TOAST*, da seguinte forma:

- PLAIN - não permite compressão ou armazenamento *out-of-line*. Esta é a única estratégia possível para as colunas com tipo de dado que não necessitem de TOAST.

- EXTENDED - permite tanto compressão quanto o armazenamento *out-of-line*. Este é o padrão para a maioria dos tipos de dado TOAST.

- EXTERNAL - permite armazenamento *out-of-line*, mas não a compressão. A utilização de EXTERNAL faz com que as operações em subcadeias de caracteres nas colunas com tipos de dado "*texto*" e *bytea* sejam mais rápidas otimizando a manipulação de dados com custo na utilização de mais espaço em disco;

- MAIN - permite compressão, mas somente efetua armazenamento *out-of-line* em situações particulares

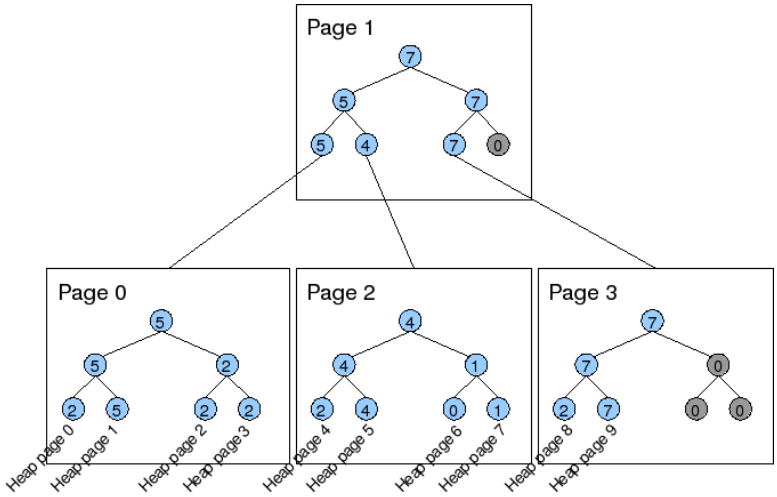
Cada tipo de dado TOAST define uma estratégia padrão para as colunas desse tipo de dado, no entanto essa estratégia pode ser alterada para uma certa coluna da tabela usando o comando ALTER TABLE SET STORAGE.

2.5. Free Space Map e Visibility Map

A estrutura de dados *Free Space Map* está presente em todas as *heaps* e índices de relações, excepto nas tabelas com índices de *hash* de forma a mapear o espaço disponível em cada relação. A *Free Space Map* , tal como está representada na figura abaixo, contém no seu interior uma árvore binária, armazenada num vetor com um byte por nó.

Cada nó folha representa uma página *heap* ou uma página *FSM* de nível superior. Em cada nó que não é folha é guardado o valor mais alto dos seus filhos. Assim, o valor máximo dos nós folha é guardado na raiz da árvore.

Na *Free Space Map* apesar de para a leitura dos atributos ser necessário percorrer toda a página a organização em *heap* elimina a necessidade de realizar cálculos para saber onde colocar o tuplo.



A estrutura de dados *Visibility Map* mapeia as páginas que se encontram visíveis para todas as transações a ocorrer em determinado instante. A estrutura utiliza somente um bit de cada página para realizar esse mapeamento, sendo que se o bit estiver a "um" significa que todos os tuplos dessa página estão visíveis para um determinado conjunto de transações.

2.6. Table e multitable clustering

A técnica de *clustering* pode ser definida como a junção de dados em espaços contínuos dos componentes de armazenamento, ou seja, os dados mais requisitados serão guardados em disco numa mesma região do mesmo, tornando assim as operações de acesso à informação mais rápidas. No *PostgreSQL* só é possível utilizar *table clustering* no entanto, terá de ser explicitamente definido pelo administrador do SGBD. O *multitable clustering* permite armazenar registos de diversas relações no mesmo ficheiro, mas tal não é suportado pelo *PostgreSQL*.

O *table clustering* é efetuado fazendo a reorganização de uma tabela tendo como base um índice escolhido da própria tabela, no entanto se for efetuado alguma alteração na tabela posteriormente, a reorganização da mesma de acordo com o índice não é feita automaticamente, ou seja, terá de ser efetuada manualmente.
Seguem alguns exemplos de como o utilizador pode utilizar o *clustering*:

- Utilizar o *cluster* na tabela *employees* com base no índice *employees_ind*:

CLUSTER employees USING employees_ind;

- Utilizar o *"cluster"* sobre o mesmo índice utilizado anteriormente:

  CLUSTER employees;

- Utilizar o *cluster* em todas as tabelas que já tenham sido *clustered*:

  CLUSTER;

2.7. Particionamento

O *PostgreSQL* permite particionar uma tabela em diversas sub-tabelas, passando a existir uma tabela lógica que representa a tabela completa e um conjunto de sub-tabelas físicas onde estão armazenados os dados. A realização de um particionamento de uma tabela, aumenta de forma significativa a complexidade da execução das operações, o que leva por vezes a um aumento do custo temporal referente à execução de operações.

O sistema de particionamento suporta um mecanismo de herança, que consiste na criação de uma partição esta será denominada como "filha" de uma outra tabela, normalmente vazia, que existe apenas para representar um conjunto inteiro de dados. O particionamento pode ser realizado de duas formas distintas:

- *Range Partitioning* - A tabela é particionada em intervalos definidos por uma coluna chave ou por um conjunto de colunas, sem que exista sobreposição entre os valores de range, associados a cada partição diferente.

- *List Partitioning* - A tabela é particionada listando explicitamente que valores chave irão aparecer em cada partição.

A implementação do particionamento é descrita detalhadamente na documentação do *PostgreSQL*.
Em comparação com o Oracle são de realçar algumas diferenças nomeadamente no que diz respeito à utilização de um sistema de ficheiros próprio, particionamento, e *clustering*. O *PostgreSQL* não implementa o seu próprio sistema de ficheiros, o que exige um esforço adicional no desenvolvimento do sistema, pois é necessário desenvolver ferramentas para gerir todos os detalhes de gestão de ficheiros que por norma o sistema operativo já oferece.
Em relação ao particionamento o Oracle fornece uma forma adicional de o realizar chamado *Hash Partitioning* que é baseado numa função de *hashing* para particionar os dados. No que diz respeito ao *clustering* da base de dados o *PostgreSQL* não suporta *multitable clustering*, o que em relação ao Oracle é uma clara desvantagem pois consegue realizar *clustering* de várias tabelas, conseguindo uma maior eficiência em tabelas que utilizam junções.

3. Indexação e Hashing

O mecanismo de indexação permite obter melhorias significativas na *performance* do acesso a informação na base de dados. Com este método é possível extrair os tuplos requeridos de uma forma mais eficiente quando comparado ao método de pesquisa linear o que poderá originar um grande impacto no processamento de *queries* especialmente se a base de dados em causa contiver um volume, relativamente grande, de informação.
Em *PostgreSQL* a sintaxe para a criação de um índice é definida da seguinte forma:

```
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 ]
```

Por definição, o *PostgreSQL* cria índices que serão *B-tree* através do comando:

```
CREATE INDEX name ON table ( column )
```

Contrariamente, caso se pretenda remover um índice deve-se executar o comando:

```
DROP INDEX name
```

Para além de B-*tree*, outros tipos de índices são também suportados em *PostgreSQL* como é o caso de *Hash*, *GiST*, *SP-GiST* e *GIN*.
Segue-se uma descrição mais aprofundada de cada um destes tipos de indexação.

3.1. Tipos de índices

3.1.1. Índices B-tree

Tal como referido anteriormente este é o método de indexação utilizado por omissão no *PostgreSQL* visto que suporta alta concorrência de operações. A acrescentar tem-se que este tipo de índices pode ser usado sobre valores inteiros, texto ou NULL sendo que apresenta bons resultados para *queries* que envolvam operações de igualdade ou desigualdade (*range queries*) sobre dados que podem ser ordenados. Desta forma, *queries* que envolvam os operadores de comparação <, <=, =, >=, > assim como operadores equivalentes aos anteriores como BETWEEN e IN ou condições IS NULL ou IS NOT NULL tendem a obter resultados satisfatórios a nível

de *performance*. O otimizador pode também usar as *B-trees* para a indexação de *queries* que envolvam o operador LIKE caso o padrão seja uma constante e esteja ancorado no início da *string*. Para além das vantagens referidas, este tipo de árvores permite que as leituras sejam não bloqueantes.
Tal como em *PostgreSQL*, o sistema de base de dados *Oracle* suporta igualmente índices B-Tree.

3.1.2. Índices de Hashing

Este tipo de índices é usado apenas em comparações de igualdade simples. O *planner* irá considerar o uso de índices *hash* sempre que uma coluna indexada esteja envolvida numa comparação que contenha o operador = . Este tipo de índices usa um algoritmo de *hashing* dinâmico pelo que se evita assim a necessidade de efetuar um *rehash* completo no momento de expansão da tabela. Em termos de *performance* os índices de *Hashing* são semelhantes aos índices *B-tree*, no entanto requerem custos de manutenção significativamente maiores pelo que as *B-trees* são tidas como a opção mais viável na maioria dos casos. Tal como referido na documentação, as operações de indexação *hash* não são *WAL-logged* o que pode originar problemas quando a base de dados falha e existem dados por escrever.
O seguinte comando permite a criação de uma índice *hash* em *PostgreSQL*:

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

Contrariamente ao *PostgreSQL*, o sistema de base de dados *Oracle* não suporta índices de *hash*.

3.1.3. Índices GiST

Generalized Search Tree é uma infraestrutura dentro da qual várias formas de indexação podem ser implementadas. Este tipo de indexação é uma árvore balanceada que serve como base para a implementação de outros esquemas de indexação como as *B-tree* ou *R-tree*. Uma das vantagens do *GiST* é o facto de permitir o desenvolvimento de tipos de dados, customizados com métodos de acesso apropriados, por um perito no domínio de tipos de dados sem ter necessidade de ser um perito em bases de dados. Desta forma, é possível estender as funcionalidades daquilo que é implementado dentro do *GiST* com outras novas funcionalidades. A distribuição *standard* do *PostgreSQL* inclui classes de operadores *GiST* para vários tipos de dados geométricos bidimensionais que suportam *queries* indexadas usando os operadores:
<<, &<, &>, >>, <<|, &<|, |&>, |>>, @>, <@, ~=, &&
O seguinte comando permite a criação de uma índice *GiST* em *PostgreSQL*:

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

Contrariamente ao *PostgreSQL*, o sistema de base de dados *Oracle* não suporta índices de *GiST*.

3.1.4. Índices SP-GiST

Este tipo de índices é bastante semelhante aos índices *GiST* sendo que também oferecem uma infraestrutura que suporta vários tipos de procura. A característica principal deste tipo de indexação é que estas árvores permitem uma divisão repetida do espaço de procura em partições que não necessitam de ter tamanho igual o que pode originar procuras com resultados bastante eficientes. Estas estruturas foram concebidas inicialmente para uso em memória sendo desenhadas como um conjunto de nós dinamicamente alocados e ligados por apontadores. No entanto, quando usada para acessos a disco deve-se minimizar o número de acessos de forma a obter a informação pretendida com eficiência satisfatória.

3.1.5. Índices GIN

Generalized Inverted Index são índices invertidos que podem manusear valores compostos com mais do que uma chave (ex.: *arrays*). Tal como o *GiST* e o *SP-GiST*, os índices *GIN* também suportam várias estratégias de indexação sendo que os operadores em que um índice *GIN* pode ser usado dependem destas estratégias. Este tipo de índices são usados frequentemente para a pesquisa em texto sendo que uma dada palavra é a chave e a sua localização é o valor. Uma das vantagens dos índices *GIN* é o facto de, à semelhança dos índices *GiST*, permitirem o desenvolvimento de tipos de dados, customizados com métodos de acesso apropriados, por um perito no domínio de tipos de dados sem ter necessidade de ser um perito em bases de dados. A distribuição *standard* do *PostgreSQL* inclui classes de operadores *GIN* para *arrays* unidimensionais que suportam os operadores:
<@, @>, =, &&
O seguinte comando permite a criação de uma índice *GIN* em *PostgreSQL*:

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

Contrariamente ao *PostgreSQL*, o sistema de base de dados *Oracle* não suporta índices de *GIN*.

3.1.6. Índices Multicoluna

Correntemente apenas os índices *B-tree*, *GiST*, e *GIN* suportam índices multicoluna.
Um índice multicoluna *B-tree* pode ser usado em condições de *queries* que envolvam um subconjunto de colunas do índice mas o índice é mais eficiente quando há restrições sobre as colunas mais à esquerda. A ideia chave é o índice utilizar restrições de igualdade nas colunas mais à esquerda em conjunção com quaisquer restrições de desigualdade na primeira coluna que não tenha uma restrição de igualdade para assim limitar o intervalo do índice percorrido. Restrições sobre colunas à direita destas colunas são verificadas no índice e desta forma minimizam acessos à tabela mas não minimizam a porção de índice percorrido.
Um índice multicoluna *GiST* pode ser utilizado em condições de *queries* que envolvam um subconjunto de índices de colunas. Condições em colunas adicionais restringem as entradas devolvidas pelo índice mas a condição na primeira coluna é a mais importante para determinar a porção de índice que necessita de ser percorrida. Um índice *GiST* vai ser relativamente ineficiente se a sua primeira coluna contiver poucos valores distintos, mesmo que hajam muitos valores distintos nas colunas adicionais.
Um índice multicoluna *GIN* pode também ser utilizado em condições de *queries* que envolvam um subconjunto de índices de colunas. Ao contrário das *B-tree* ou *GiST*, a eficiência das procuras será igual independentemente das colunas indexadas usadas nas condições da *query*.
Na grande maioria dos casos, um índice referente a uma coluna é suficiente e consequentemente poupa-se tempo e espaço. Índices com mais de 3 colunas dificilmente ajudam a não ser que o uso da tabela seja extremamente estilizado.
O sistema de base de dados *Oracle* suporta igualmente índices multicoluna.

3.1.7. Índices e Ordenação

A acrescentar a simplesmente se devolver o resultado de uma *query*, um índice pode devolver este resultado já ordenado. Caso uma *query* especifique a operação ORDER BY o resultado vem ordenado sem se ter a necessidade de se realizar um passo de ordenação adicional. O *planner* irá realizar a operação ORDER BY percorrendo o índice disponível que satisfaça a especificação ou percorrendo a tabela pela sua ordem física e ordenando explicitamente. Se se necessitar de percorrer uma grande porção da tabela uma ordenação explícita pode ser mais eficiente do que a utilização de um índice visto que requer menos acessos de I/O. Um caso especial ocorre quando a operação ORDER BY é combinada com o LIMIT n, neste caso uma ordenação explícita terá de processar todos os dados para identificar os primeiros n tuplos mas se existir um índice que satisfaça o ORDER BY, os primeiros n tuplos são devolvidos tornando-se desnecessário percorrer o resto dos dados.
Por definição, os índices *B-tree* armazenam as suas entradas em ordem ascendente com *nulls* no fim. Isto implica que o percorrer de um índice de uma coluna devolva os resultados de forma ordenada satisfazendo assim a operação ORDER BY.

3.1.8. Índices múltiplos combinados

O percorrer de um índice único pode usar apenas cláusulas de *queries* que utilizem índices de colunas com operadores da sua classe de operadores e que são unidos com a operação AND. Como exemplo, num índice em (a,b) numa condição como WHERE a=5 AND b=6 pode-se usar o índice único mas numa condição como WHERE a=5 OR b=6 já não se pode utilizar este índice.

O *PostgreSQL* tem a habilidade de combinar índices múltiplos para lidar com casos que não podem ser implementados com um *scan* de um índice único. O sistema poderá formar condições AND e OR percorrendo vários índices. Para combinar múltiplos índices, o sistema percorre cada índice e prepara um *bitmap* em memória dando a localização dos tuplos da tabela que satisfazem as condições em causa. Os *bitmaps* são unidos a partir de condições AND e OR sendo de seguida devolvidos. Os tuplos são percorridos sequencialmente, pela sua ordem física, pelo que se houver a necessidade de ordenar o resultado, esta ordenação deverá ser realizada num passo adicional. Visto que o percorrer de cada índice adiciona tempo o *planner* algumas vezes opta por percorrer um índice simples.

O sistema de base de dados *Oracle* suporta igualmente a combinação de índices múltiplos. No entanto, enquanto que em *Oracle* é possível definir uma forma de indexação com *bitmaps*, em *PostgreSQL* estes são usados apenas internamente sendo que o utilizador não pode definir um tipo de indexação com *bitmaps*.

3.1.9. Índices únicos

Os índices podem também ser usados para fortalecer a unicidade do valor de uma coluna ou a unicidade do combinado de várias colunas. Apenas os índices *B-tree* podem ser declarados únicos com o seguinte comando:

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

Quando um índice é declarado como sendo único, tuplos de várias tabelas com o mesmo valor indexado deixam de ser permitidos. Um índice multicoluna único só rejeitará casos onde todas as colunas indexadas sejam iguais em múltiplos tuplos.

O *PostgreSQL* cria automaticamente um índice único quando uma restrição única ou chave primária é definida para uma tabela. O índice cobre as colunas que compõe a chave primária ou a restrição única e é o mecanismo que garante a restrição.

O sistema de bases de dados *Oracle* suporta também o mecanismo de índices únicos.

3.1.10. Índices sobre expressões

Uma índice, em vez de ser sobre uma coluna da tabela, pode ser também uma função ou expressão escalar computada sobre uma ou mais colunas da tabela. Esta

característica é útil para obter um acesso mais rápido a tabelas baseadas no resultado de computações.
Como exemplo, uma forma de avaliar comparações *case-sensitive* é usando a função *lower*

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

Esta *query* pode usar um índice, caso este tenha sido definido, sobre o resultado da função lower(col1) através da expressão:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

Caso se declare um índice UNIQUE este irá evitar a criação de tuplos com valores de *column* diferentes em maiúsculas/minúsculas assim como irá prevenir a criação de tuplos com valor de *column* idênticos. Desta forma, pode ser feito uso de índices sobre expressões em restrições que não sejam definíveis como restrições únicas.
Índices sobre expressões são relativamente custosos de manter porque as expressões derivadas devem ser calculadas a cada inserção, ou atualização, de um tuplo. No entanto, as expressões sobre índices não são re-calculadas durante a procura indexada visto que já se encontram armazenadas no índice. Desta forma, índices sobre expressões tornam-se úteis quando a rapidez de extração é mais importante do que a rapidez de inserção ou atualização.
No sistema de base de dados *Oracle* pode-se também fazer uso de índices sobre expressões.

3.1.11. Índices parciais

Este tipo de índices é construído sobre um subconjunto de uma tabela, este subconjunto é definido por uma expressão condicional (predicado do índice parcial). O índice contém entradas apenas para os tuplos da tabela que satisfaçam o predicado. Uma das maiores razões para o uso de índices parciais é o facto de estes evitarem a indexação de valores comuns (valores que aparecem mais do que uma dada percentagem de vezes em tuplos da tabela). Visto que uma *query* que use estes valores não vai usar o índice de qualquer forma, este método tem como consequência a redução do espaço ocupado pelo índice assim como o aumentar da rapidez de processamento de *queries* e de operações de atualização na tabela.
Este índices podem também ser definidos como *UNIQUE*.
Contrariamente ao *PostgreSQL*, o sistema de base de dados *Oracle* não suporta índices parciais.

3.2. Estruturas temporariamente inconsistentes

Em certas alturas é possível encontrar-se estruturas temporariamente inconsistentes no *PostgreSQL*. O comando SET CONSTRAINTS define a forma como a verificação de restrições é realizada na transação corrente. Restrições IMMEDIATE são verificadas no final de cada ação executada enquanto que restrições DEFERRED são

verificadas após o *commit* final. O comando que permite definir a forma de verificação de restrições em *PostgreSQL* é o seguinte:

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

Tanto o *PostgreSQL* como o sistema de bases de dados *Oracle* permitem estruturas temporariamente inconsistentes.

4. Processamento e optimização de perguntas

A execução de uma pergunta *SQL* envolve várias etapas, desde o momento em que o utilizador introduz a *query* na aplicação cliente até ao momento em que o resultado é apresentado ao utilizador. Sucintamente:

- É estabelecida uma ligação entre a aplicação cliente e o SGBD transmitindo uma *query* pela qual se espera a resposta do resultado.
- O *parser* analisa a *query* introduzida e verifica se esta tem a sintaxe correta. Em caso afirmativo, é criada a *query tree* correspondente.
- O sistema de reescrita procura por regras na *query tree* aplicando-lhes as transformações necessárias com base nos catálogos do sistema.
- A partir da *query tree*, o planeador/otimizador cria um *query plan*. Para isso, são criados os diversos planos possíveis que devolvem o mesmo resultado estimando o custo de execução de cada plano. O que tiver menor custo será o escolhido.
- O executor executa o plano de execução devolvendo os tuplos que pertencem ao resultado final.

4.1. Algoritmos de seleção

No *PostgreSQL*, a operação de seleção pode ser efetuada utilizando um dos seguintes algoritmos: *Sequential scan*, *Index scan* e *Bitmap index scan*.

4.1.1. Sequential scan

Este algoritmo consiste em percorrer a tabela completa, de forma sequencial, verificando para cada tuplo a condição de seleção (cláusula *WHERE*). Os tuplos que satisfizerem a condição de seleção contribuirão para o resultado final.

4.1.2. Index scan

Este algoritmo permite a pesquisa a partir de tabelas de índice. Os tuplos, que contribuem para o resultado final, são devolvidos pela ordem do índice e não pela ordem em que se encontram armazenados em disco.

4.1.3. Bitmap index scan

Este algoritmo implica a utilização de estruturas auxiliares, conhecidas por *bitmap*, que são simplesmente *arrays* de *bits*. Aplicável a cada valor possível de cada atributo da relação, caso o domínio seja discreto, estes *arrays* possuem o valor 1 para indicar que o registo da relação, cujo índice é igual ao índice onde ocorre esse valor no *array*, possui como valor do atributo o valor representado pelo *bitmap*. O valor 0 no *bitmap* indica que o registo da relação, cujo índice é igual ao índice onde ocorre esse valor no *array*, não tem como valor do atributo o valor representado pelo *bitmap*. A utilização de *bitmaps* é especialmente vantajosa para operações de agregação como é

o caso das operações *AVG* e *COUNT* pois não é necessário aceder às relações, apenas aos *bitmaps*, para retornar o resultado final.

4.2. Algoritmos de junção

O *PostgreSQL* implementa alguns algoritmos de junção como é o caso do *Hash Join*, do *Merge Join* e do *Nested Loop Join*.

4.2.1. Hash Join

Com este algoritmo, é efetuado inicialmente um *scan* à relação da direita de modo a particionar a relação, através da aplicação de uma função de *hash* aos atributos de junção, num conjunto de *buckets* formando, assim, uma *hash table*. De seguida, é efetuado um *scan* à relação da esquerda e aplicada a mesma função de *hash* aos atributos de junção de cada tuplo. Consequentemente, estes tuplos serão mapeados nos *buckets* da tabela de *hash* anteriormente construída e, assim, cada *bucket* incluirá os pares de tuplos, um de cada tabela, que irão contribuir para o resultado final.

4.2.2. Merge Join

A utilização deste algoritmo implica que ambas as relações estejam ordenadas pelos atributos de junção. Após esta condição se verificar, é efetuado um *scan* sobre cada relação, em paralelo, e os pares de tuplos que satisfizerem a condição de junção contribuirão para o resultado final. Apesar de poder existir um custo inicial acrescido devido à ordenação das tabelas pelos atributos de junção, esta estratégia poderá revelar-se interessante visto que para cada relação apenas é efetuado um *scan*.

4.2.3. Nested Loop Join

Este algoritmo consiste em efetuar um *scan* sobre a tabela do lado direito para cada tuplo existente na tabela do lado esquerdo. Para cada par de tuplos é verificada a condição de junção. Visto ser necessário analisar todos os pares de tuplos possíveis, constituídos por um tuplo de cada tabela, este algoritmo pode tornar-se custoso. No entanto, caso o *scan* efetuado sobre a tabela do lado direito seja um i*ndex scan* esta estratégia poderá revelar-se benéfica.

4.3. Algoritmos de ordenação

O *PostgreSQL* implementa dois algoritmos de ordenação – o *External Merge Sort* e o *Quicksort*.

4.3.1. External Merge Sort

Este algoritmo é utilizado quando as tabelas não cabem em memória. Assim, particiona-se a tabela em blocos que cabem em memória e ordena-se cada um desses blocos de forma independente. Posteriormente, estes blocos vão sendo combinados até perfazer a tabela que constitui o resultado final.

4.3.2. Quicksort

Este algoritmo é utilizado quando as tabelas cabem em memória devido à sua eficiência.

4.4. Mecanismo de suporte para expressões complexas

O *PostgreSQL* possui mecanismos de suporte para expressões complexas como é o caso da materialização, *pipelining* e paralelização.

4.4.1. Materialização

A avaliação de expressões pode ser efetuada com o auxílio de materializações que permitem guardar resultados intermédios. Esta abordagem permite que um plano de execução, que utilize por exemplo o algoritmo *Merge Join*, seja mais eficiente se tiver que percorrer diversas vezes a mesma tabela. No entanto, poderá ser necessário um considerável espaço de armazenamento para guardar os resultados intermédios e os custos de escrever e voltar a ler esses resultados pode ser elevado.

4.4.2. Pipelining

Esta abordagem permite que à medida que os resultados parciais de uma expressão sejam obtidos, estes sejam imediatamente enviados para outra expressão no nível acima do plano de execução tornando, assim, mais rápida a execução. No entanto, em operações de ordenação o *pipelining* não pode ser utilizado pois nesta situação é necessário ter acesso a todos os dados *a priori*. Aqui, o *pipelining* é executado em modo *demand driven*, ou seja, os dados são enviados para o nível acima do plano de execução à medida que vão sendo requeridos podendo acontecer o efeito de cascata.

4.4.3. Paralelização

É possível o processamento de perguntas de forma paralela, quer seja distribuindo a computação por diversos processadores ou através da utilização de vários discos em simultâneo para as operações I/O. Por exemplo, o algoritmo *Hash Join* pode ser paralelizável estando cada processador a computar um *hash* diferente de forma simultânea.

4.5. Estimativas

O *PostgreSQL* permite a utilização do comando EXPLAIN para ver o plano de execução de uma determinada *query*. A sintaxe é a seguinte:

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

Onde o *option* poderá ser uma das seguintes opções:

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

A execução deste comando permite a visualização da seguinte informação:

- Tipo de operação utilizada para percorrer a(s) tabela(s) referenciada(s) pela *query* (por exemplo, pesquisa sequencial ou pesquisa por índice).
- Estimativa do custo inicial.
- Estimativa do custo total.
- Estimativa do número de tuplos devolvidos pelo plano de execução.
- No caso de existirem várias tabelas referenciadas pela *query* também é visualizado os algoritmos de junção utilizados.

Caso a opção ANALYSE seja ativada, a expressão avaliada será de facto executada, e não apenas planeada, sendo assim possível visualizar ainda o custo real da *query*.

4.6. Estatísticas

O comando ANALYSE coleciona estatísticas acerca do conteúdo das tabelas da base de dados, guardando o resultado no catálogo do sistema *pg_statistic*. Consequentemente, o *query planner* utiliza estas estatísticas para ajudar a determinar o plano de execução mais eficiente para as *queries*. A sintaxe do comando é a seguinte:

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

4.7. Parametrização

Existe um conjunto de parâmetros de configuração que influenciam os planos escolhidos pelo otimizador. Se o plano *default* escolhido pelo otimizador para uma *query* em particular não for ótimo, uma possível solução consiste em utilizar um

destes parâmetros de configuração de modo a forçar o otimizador a escolher outro plano. Eis os parâmetros:

- **enable_bitmapscan(boolean)**: ativa a utilização de planos do tipo *bitmap-scan* por parte do *query planner*.
- **enable_hashjoin(boolean)**: ativa a utilização de planos do tipo *hash-join* por parte do *query planner*.
- **enable_indexscan(boolean)**: ativa a utilização de planos do tipo *index-scan* por parte do *query planner*.
- **enable_indexonlyscan(boolean)**: ativa a utilização de planos do tipo *index-only-scan* por parte do *query planner*.
- **enable_material(boolean)**: ativa a utilização de materialização por parte do *query planner*. Não é possível desativar a materialização inteiramente.
- **enable_mergejoin(boolean)**: ativa a utilização de planos do tipo *merge-join* por parte do *query planner*.
- **enable_nestloop(boolean)**: ativa a utilização de planos do tipo *nested-loop* por parte do *query planner*. Não é possível desativar o *nested-loop* inteiramente.
- **enable_seqscan(boolean)**: ativa a utilização de planos do tipo *seq-scan* por parte do *query planner*. Não é possível desativar a pesquisa sequencial inteiramente.
- **enable_sort(boolean)**: ativa a utilização de ordenação por parte do *query planner*. Não é possível desativar a ordenação inteiramente.

Tudo o que foi apresentado neste capítulo também existe no Oracle.

5. Bases de dados distribuídas

As bases de dados distribuídas podem ser de dois tipos, homogéneas ou heterogéneas. Nas bases de dados homogéneas os servidores são compostos por bases de dados do mesmo tipo, já nas bases de dados heterogéneas os servidores que as compõem possuem mais de um tipo de base de dados. A sincronização dos diversos servidores é um problema fundamental e a não existência de uma única solução para resolver o problema, abriu portas à criação de diversas soluções, sendo que cada uma tenta minimizar os impactos da sincronização de maneiras diferentes.
A replicação dos dados pode se dar de maneira síncrona ou assíncrona. No caso de replicação síncrona, cada transação é dada como concluída quando todos os nós confirmam que a transação local foi bem sucedida *commited*. Na replicação assíncrona, o nó principal executa a transação enviando confirmação ao solicitante e então encaminha a transação aos demais nós. Somente nas versões mais recentes é que o *PostgreSQL* permite implementar bases de dados distribuídas fornecendo alta disponibilidade e desempenho. Existem também algumas extensões ao *PostgreSQL*, que foram surgindo por o sistema ser *open source*, e que permitem outras funcionalidades, tal como o *Slony-I*,
Nas seções seguintes apenas vão ser abordadas as técnicas implementadas pelo sistema de modo a garantir a sincronização.

5.1. LOG SHIPPING

Na replicação da base de dados feita por *Log-Shipping* o servidor primário e o servidor *standby* (servidor pronto para tomar o controlo das operações caso o servidor primário falhe) trabalham em conjunto para evitar a inconsistência no sistema. Dessa forma o servidor primário trabalha de modo contínuo enquanto que o servidor primário trabalha em modo de recuperação, lendo os ficheiros WAL (Write-Ahead Logs) do servidor primário. O *Log-Shipping* é um modo de replicação assíncrono pelo que os registos WAL só são enviados depois de uma determinada transação ter feito *commit*. Assim, existe uma janela para dados perdidos que previne que os dados que ainda não foram enviados após as transações se possam perder caso o servidor primário sofra uma falha. O tamanho da janela de perda de dados pode ser limitado através do parâmetro *archive_timeout*. No entanto se o *timeout* for muito pequeno pode aumentar de forma significativa a largura de banda necessária para enviar os registos para o servidor primário.
A implementação desta funcionalidade não necessita de quaisquer alterações nas tabelas da base de dados e tem pouco impacto na performance do servidor primário. No *PostgreSQL* esta funcionalidade é implementada transferindo de tempos a tempos ficheiros WAL de 16Mb para o servidor primário, sendo que se conseguem transmitir a baixo custo mesmo para sistemas em sítios distantes. Como veremos mais à frente neste capítulo o mesmo não acontece com a replicação por *streaming* visto que os WAL vão mudando de tamanho ao longo da ligação de rede.

5.2. Streaming replication

A replicação por *streaming* oferece a possibilidade de o servidor *standby* ser atualizado mais rapidamente do que usando o método *Log-Shipping*, pois ao invés de se aguardar que um ficheiro WAL seja completamente preenchido, o servidor *standby* liga-se ao servidor primário e gera um *stream* contínuo de registos WAL em direção ao servidor primário.
Este modelo de replicação é assíncrono por defeito, pelo que existe um pequeno atraso entre fazer *commit* de uma transação no servidor primário e essas alterações ficarem visíveis no servidor *standby*. Dessa forma não é necessário utilizar o parâmetro *achive_timeout* pois o atraso da atualização é muito mais pequeno que no *Log-Shipping*.

5.3. Cascading replication

A funcionalidade de replicação em cascata permite que um servidor *standby* aceite ligações de replicação e consiga enviar um *stream* de registos WAL para outros servidores *standby* atuando como transmissor. Este modo de replicação permite reduzir o *overhead* de ligações que são efetuadas para o servidor primário e minimizar as larguras de banda entre os *sites*. Cada réplica pode-se ligar a várias outras réplicas para as quais envia as atualizações recebidas mas que apenas se pode ligar a uma réplica, ou diretamente ao servidor primário, para receber as atualizações. Quando uma réplica que serve como intermediária para outras é promovida a primária, esta termina imediatamente as ligações que tem com outras réplicas para quem reencaminha as atualizações.

5.4. Synchronous replication

A replicação síncrona oferece a possibilidade de obter confirmação de que todas as atualizações feitas por uma transação foram transferidas para um servidor síncrono *standby*. Quando é utilizada a replicação síncrona, cada *commit* de uma transação de escrita tem de aguardar até que tenha recebido confirmação, tal que o *commit* tenha sido escrito no ficheiro de log em disco no servidor primário e *standby*. Dessa forma a única maneira que existe de serem perdidos dados é quando ambos o servidor primário e *standby* falharem ao mesmo tempo. Assim consegue-se fornecer um alto nível de durabilidade. De outro ponto de vista as transações que apenas leiam o estado da base de dados ou que sejam abortadas não têm de esperar, não existindo necessidade de esperar que as alterações sejam efectuadas. Assim não é necessário garantir que não existe perca de dados porque as alterações efectuadas terão de ser desfeitas e como a réplica nunca receberá a informação de *commit* da transação nunca tornará visíveis as suas alterações.

Em comparação com o Oracle, que também suporta replicação de dados à que realçar que no *PostgreSQL* esta replicação apenas permite que um só servidor funcione como primário. Devido ao facto de a introdução de suporte para bases de dados distribuídas somente ter acontecido em versões mais recentes, torna o *PostgreSQL* um pouco limitado em relação ao Oracle, sendo que no Oracle é possível realizar a fragmentação dos dados.

6. Gestão de transações e controlo de concorrência

O conceito de controlo de concorrência em sistemas de bases de dados é usado para endereçar conflitos de acessos simultâneos a dados de por parte de múltiplos utilizadores. Quando aplicado na prática este deve coordenar transações concorrentes preservando a integridade dos dados. O *PostgreSQL* permite definir explicitamente transações sendo que cada transação é iniciada pelo comando BEGIN e finalizada pelo comando END. No entanto o *PostgreSQL* analisa cada comando como sendo uma transação, ou seja, como estando envolvido dentro dos comandos BEGIN e END.

O *PostgreSQL* não impõe limite na duração das transações mas é aconselhável que estas não sejam demasiado longas devido à possibilidade de uma transação ficar muito tempo bloqueada num *lock* ou de se ter de fazer *rollback* sobre a mesma.

O *PostgreSQL* não implementa *nested transactions*, no entanto oferece um mecanismo de *savepoints* que permite a criação de pontos de restauro sendo que se houver a necessidade de se fazer *rollback* depois de uma dada instrução a base de dados retorna ao estado em que foi executado o último *savepoint*.

6.1. Multiversion Concurrency Control

Internamente, o *PostgreSQL* mantém a consistência dos dados usando o modelo MVCC, isto é, durante o processo de *querying* cada transação tem acesso a uma dada versão, consistente, da base de dados independentemente do estado atual da mesma. Isto impossibilita que uma transação tenha acesso a dados inconsistentes providenciando assim o necessário mecanismo de isolação. Este processo tem, no entanto, um consumo elevado de recursos devido ao elevado número de versões que guarda pelo que o sistema tem como responsabilidade eliminar versões que deixam de ser necessárias.

A principal vantagem do modelo MVCC em relação a um modelo baseado em *locks* é o facto de este permitir que operações de leitura não entrem em conflitos com operações de escrita tornando-se consequentemente mais eficiente.

6.2. Níveis de isolamento

O *PostgreSQL*, tal como o *SQL Standard*, oferece 4 níveis de isolamento: *Read Committed*, *Read Uncommitted*, *Repetable Read* e *Serializable*. No entanto ao se escolher o nível de isolamento *Read Uncommitted*, internamente, está-se a realizar um *Read Comitted* pelo que, na prática, o *PostgreSQL* oferece 3 níveis de isolamento. Pode-se explicitar o nível de isolamento pretendido com o seguinte comando:

```
SET TRANSACTION { SERIALIZABLE | REPEATABLE READ |
READ COMMITTED | READ UNCOMMITTED } READ WRITE |
```

READ ONLY [NOT] DEFERRABLE

6.2.1. Read Committed

É o nível de isolamento por defeito nas transações do *PostgreSQL*. Quando uma transação usa este nível de isolamento, uma *query* SELECT apenas vê dados que estão *committed* até ao instante em que a *query* deu início. Consequentemente, se uma outra transação concorrente atualizar os dados em causa, esta atualização não será vista pela *query* SELECT. No entanto, a operação SELECT pode ver dados que ainda não estão *committed* desde que a atualização destes dados tenha sido realizada na mesma transação onde é executada a operação SELECT. De referir também que dois comandos SELECT sucessivos, dentro da mesma transação, podem obter resultados diferentes casos os dados tenham sido atualizados, e *committed*, entretanto por uma transação concorrente originando o fenómeno de *nonrepetable read*.
Operações de atualização de dados como UPDATE, DELETE, SELECT FOR UPDATE e SELECT FOR SHARE comportam-se da mesma forma que a operação SELECT na procura de tuplos. Caso duas transação concorrentes pretendam executar operações de atualização sobre os mesmos dados, a última terá de esperar que a primeira faça *commit* ou *rollback*.
No caso da primeira transação fazer *rollback* não são feitas alterações à base de dados pelo que a operação da segunda transação pode executar a sua *query* sobre os dados que estavam originalmente na base de dados.
No caso da primeira transação fazer *commit* com uma operação de remoção de tuplo então a segunda transação irá ignorar o tuplo em questão, caso contrário a primeira transação irá usar o valor atualizado na execução da sua *query* de atualização.

6.2.2. Repetable Read

Este nível de isolamento apenas vê um *snapshot* de dados *committed* antes da transação dar início. No entanto, tal como o nível de isolamento anteriormente descrito, uma transação pode ver atualizações ainda não *committed* desde que as mesmas tenham sido executadas nessa transação.
Desta forma, *queries* SELECT sucessivas numa transação obtêm sempre repostas iguais em qualquer caso não existindo assim a possibilidade de ocorrer o fenómeno de *nonrepetable read*.
Operações de atualização de dados como UPDATE, DELETE, SELECT FOR UPDATE e SELECT FOR SHARE comportam-se da mesma forma que a operação SELECT na procura de tuplos sendo que os dados que obtêm são sempre os dados *committed* antes da transação se iniciar. Caso existam várias transações concorrentes a tentar atualizar os mesmos dados apenas uma tem acesso a estes dados. Caso esta transação faça *commit* então as restantes transações devem fazer *rollback* e apresentarão uma mensagem de erro. Isto deve-se ao facto de uma transação não poder modificar ou fazer *lock* em tuplos alterados por outra transação. Caso

contrário, em que a primeira transação faz *rollback*, então uma das restantes pode continuar com a sua execução da *query*.

6.2.3. Serializable

Representa o nível de isolamento mais restrito. Este nível garante o resultado como se as transações fossem executadas sequencialmente e não concorrentemente. A implementação deste nível de isolamento é bastante semelhante à implementação do *Repetable Read* sendo que caso várias transações pretendam aceder ao mesmo recurso há uma grande probabilidade de se ter de fazer *rollback*. São também monitorizadas condições para detectar conflitos. Esta monitorização e deteção de conflitos pode provocar falhas de serialização.
Para garantir uma verdadeira serialização o *PostgreSQL* usa *predicate locking*, isto é, mantém *locks* para determinar que operações de escrita podem ter impacto sobre o resultado de operações de leitura previamente executadas. Este tipo de *locks* não causa nenhum tipo de bloqueio sobre as transações pelo que não pode ser responsável por causar *deadlocks* sendo o seu principal objectivo identificar e assinalar dependências entre transações serializáveis concorrentes que podem levar a falhas de serialização. Em contraste, numa transação que opera em *Read Committed* ou *Repetable Read* e que pretenda manter a consistência dos dados poderá ter de fazer um *lock* sobre a tabela toda não permitindo assim acessos a essa tabela enquanto o *lock* estiver acionado.

6.3. Locking explícito

O *PostgreSQL* providência vários tipos de *locks* para controlar acesso a tabelas. O *locking* explícito pode ser usado em situações em que o MVCC não tem o comportamento desejado. Este sistema de *locks* tem 2 níveis de granularidade: *locks* executados sobre tabelas e *locks* executados sobre tuplos. Todos os *locks* são mantidos num tabela, *pg_locks*, sendo que cada transação mantém um *lock* até a mesma fazer *commit* ou *rollback*.

6.3.1. Locking em tabelas

Existem vários modos de *locks* sobre tabelas sendo que estes afetam a tabela inteira. De seguida é apresentado, para cada modo de *lock*, a instrução que o pode adquirir e os seus conflitos com outros modos *lock*.

- ACCESS SHARE
  - Conflito com o modo ACCESS EXCLUSIVE
  - Comando SELECT adquire este *lock* sendo que, em geral, qualquer *query* que apenas contenha operações de leitura adquire este lock
- ROW SHARE

    - Conflito com os modos de *lock* EXCLUSIVE e ACCESS EXCLUSIVE
    - Os comandos SELECT FOR UPDATE e SELECT FOR SHARE adquirem este *lock* para as suas tabelas alvo
- ROW EXCLUSIVE
    - Conflito com SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE
    - Os comandos UPDATE, DELETE e INSERT adquirem este *lock* para as suas tabelas alvo
- SHARE UPDATE EXCLUSIVE
    - Conflito com SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE
    - Os comandos VACUUM (sem ser *FULL*), ANALYSE, CREATE INDEX CONCURRENTLY e outras formas de ALTER TABLE adquirem este modo de *lock*
- SHARE
    - Conflito com ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE
    - O comando CREATE INDEX (sem CONCURRENTLY) adquire este modo de *lock*
- SHARE ROW EXCLUSIVE
    - Conflito com ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE
- EXCLUSIVE
    - Conflito com ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE
- ACCESS EXCLUSIVE
    - Conflito com todos os modos anteriormente descritos
    - Os comandos ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER e VACCUM FULL adquirem este modo de *lock*

6.3.2. Locking em tuplos

Este tipo de *locks* pode ser exclusivo ou partilhado. Um lock exclusivo sobre um tuplo é adquirido automaticamente quando este é atualizado ou apagado. Tal como no caso anterior, neste tipo de *locking* cada *lock* é também mantido até a respectiva transação fazer *commit* ou *rollback*. Desta forma, caso se pretenda obter um *lock* exclusivo sobre um tuplo deve-se executar o comando SELECT FOR UPDATE enquanto que caso se pretenda obter um *lock* partilhado deve-se executar o comando SELECT FOR SHARE, no entanto a transação em que este último comando é executado não permite que qualquer outra transação possa alterar o tuplo em questão.

6.4. Deadlocks

O uso de *locks* explícitos pode originar situações de *deadlocks* em que uma (ou mais) transações mantêm *locks* sobre recursos a que outra transação pretende aceder. Por exemplo, se a transação T1 adquire um *lock* exclusivo sobre a tabela A e de seguida tenta obter um *lock* exclusivo sobre a tabela B sendo que existe uma transação T2 que já contém um *lock* exclusivo sobre a tabela B, se de seguida T2 pretender obter um *lock* exclusivo sobre a tabela A verifica-se uma situação em que nenhuma transação pode avançar com as suas operações pelo que se está na presença de um cenário de *deadlock*.

O *PostgreSQL* detecta automaticamente situações de *deadlock* e resolve-as abortando uma das transações permitindo às restantes continuarem com as suas operações. Para que tal seja possível o *PostgreSQL* usa um algoritmo baseado em *timeouts* que monitoriza o tempo de uma transação e caso uma transação demore mais do que um segundo (tempo por definição) para realizar as suas operações então é executado o algoritmo de *deadlocks* que constrói um grafo de espera entre transações. Neste grafo, quando uma transação está à espera que outra liberte um *lock* existe um arco da primeira transação dirigido para a segunda. Após o grafo estar completo o algoritmo verifica se existe uma situação de *deadlock* e caso tal se verifique é executado o *rollback* sobre uma das transações de forma as que as restantes possam continuar com as suas operações.

O *PostgreSQL* não faz nenhum tipo de optimização dinâmica sobre o parâmetro de *timeout* deixando a cargo do administrador da base de dados a definição deste valor.

6.5. Consistência

Tanto o nível de isolamento *Read Committed* como o nível de isolamento *Repetable Read* podem levar à violação da integridade da base de dados. Desta forma, para garantir a consistência no acesso a informação da base de dados o melhor nível de isolamento a usar-se deve ser o *Serializable* visto que se houver algum tipo de violação de integridade uma das transações responsáveis é abortada não se correndo assim o risco de obter leituras inconsistentes. Ainda assim, o utilizador pode definir o modo como a verificação de integridade numa transação é realizada. Com o comando DEFERRED a integridade da base de dados é verificada no final da execução de cada transação enquanto que com o comando IMMEDIATE esta mesma verificação é realizada no final de cada operação realizada.

As características referenciadas neste subcapítulo são bastante semelhantes às do sistema de base de dados *Oracle*.

7. Outras características do sistema

7.1. XML

O tipo de dados XML é suportado pelo *PostgreSQL* e pode ser utilizado para armazenamento de dados. A vantagem em seguir esta abordagem prende-se com o facto de a informação ser verificada de modo a garantir que os documentos são bem formados. Existem ainda funções de suporte sobre dados XML para executar operações *type-safe*. No entanto, o *PostgreSQL* não permite a validação de dados XML com DTD ou com XML Schema, mesmo que este seja referenciado.

Para utilizar este tipo de dados é necessário fazer a instalação com o comando:

configure-with-libxml.

Produzir valores do tipo XML, a partir de texto, implica a execução do seguinte comando:

XMLPARSE ({DOCUMENT | CONTENT} *value*)

Por outro lado, produzir texto a partir de valores XML é possível com o comando:

XMLSERIALIZABLE ({DOCUMENT | CONTENT } *value* AS *type*)

É possível verificar se determinado documento é do tipo XML através da execução do comando:

value IS DOCUMENT

A expressão avaliada devolve *true* se o argumento *value* corresponde a um documento XML, devolve *false* caso contrário ou *null* se o argumento for *null*.

O *PostgreSQL* permite ainda a avaliação de expressões *XPath*. Por exemplo, a seguinte função devolve *true* se a expressão XPath *text* devolve algum nó ou *false* caso contrário, com base no argumento XML *xml*:

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

7.2. Autenticação

O *PostgreSQL* oferece um número diverso de métodos de autenticação do cliente.
A autenticação do cliente é controlada por um ficheiro de configuração, *pg_hba.conf*, que guarda um conjunto de registos, um por cliente, especificando o tipo de conexão, o intervalo de endereços IP do cliente se for relevante, o nome da base de dados, o nome do utilizador e o método de autenticação a ser usado em conexões que utilizem estes parâmetros.
Por exemplo, o seguinte comando indica que o utilizador *usertest* tem acesso à base de dados *dbtest* através de uma conexão TCP/IP sobre SSL utilizando *password*:

hostssl dbtest usertest 255.255.255.255 *password*

Os vários métodos de autenticação disponíveis são os seguintes:

- Trust
- Password
- GSSAPI
- SSPI
- Kerberos
- Ident
- Peer
- LDAP
- RADIUS
- Certificate
- PAM

7.3. Triggers

O *PostgreSQL* permite a criação de *triggers* para auxiliar na preservação da consistência de uma base de dados.
Os *triggers* podem ser escritos na maioria das linguagens procedimentais disponíveis, incluindo PL/pgSQL, PL/Tcl, PL/Perl, PL/Python. Podem ainda ser escritos em C.
A sintaxe para a criação de um *trigger* é a seguinte:

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

Onde o *event* poderá ser uma das seguintes opções:

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

Tal como no *PostgreSQL*, o *Oracle* também permite a definição de *triggers* para manter a consistência da base de dados. No Oracle, o XML pode ser usado para transferência de dados. No entanto, ao contrário do *PostgreSQL*, um documento XML pode ser validado com um DTD ou XML Schema. Também são permitidas

consultas em XPath e XQuery. No *Oracle*, também existem diversas formas de autenticação de modo a permitir apenas certos utilizadores de aceder, processar e alterar dados. Para cada utilizador existem ainda vários tipos de privilégios concedidos que limitam, de forma diferente, o seu acesso e as suas ações sobre a base de dados.

8. Referências

[1] *PostgreSQL 9.3.4 documentation*.
[2] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S. *Database System Concepts*, 6th ed. McGraw-Hill, 2010.