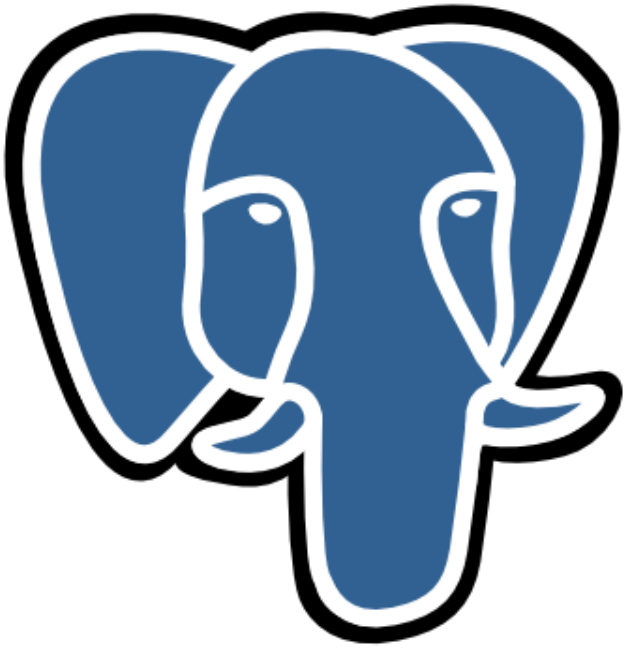
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Sistemas de Bases de Dados



PostgreSQL

Trabalho elaborado por:

- 41785, André Sardo
- 41938, Daniel Adelino
- 42077, Henrique Garcês

Grupo: 22

Ano lectivo: 2013/2014

Índice

1. Introdução

O presente relatório surge no âmbito da disciplina de Sistemas de Bases de Dados e o Sistema de Gestão de Bases de Dados escolhido foi o PostgreSQL. Este é maioritariamente escrito em C e é o SGBDs *open source* mais avançado e completo atualmente. A sua primeira versão data de Maio de 1995, tendo sido divulgado na internet em 1996, ano em que passou a ser *open source*. Hoje em dia o PostgreSQL é usado por centenas de empresas das mais diversas áreas como, por exemplo: na área da BioPharm; Comércio eletrónico; Educação (diversas universidades a nível mundial); Finanças; Jogos; Saúde; Projetos Open Source (Debian, LAMP, PostGIS, SourceForge); Tecnologia (Apple, Fijitsu, Red Hat, Sun Microsystems); Telecomunicações (Cisco, Skype).

O grupo escolheu o PostgreSQL como o SBGB a estudar para este trabalho por ser *open source* (logo o acesso a documentação do mesmo é facilitado), gratuito, estar em plena ascensão no mercado e ser bastante completo depois de mais de 15 anos de desenvolvimento. Neste relatório, a versão usada do PostgreSQL é a última versão estável disponível, a versão 9.3, e aqui irá ser explicado e mostrado como se comporta o PostgreSQL em cada um dos conceitos estudados ao longo do ano letivo, e para cada um destes será feita uma comparação com o SGBD estudado nas aulas, o Oracle 11g.

2. Armazenamento e File Structure

2.1 Buffer Management

O PostgreSQL usa um *buffer* de cache partilhado com o sistema operativo, com política *LRU* (Least Recently Used), ou seja, para decidir que páginas devem ser carregadas em memória e que páginas libertar, sempre que uma página é carregada em cache esta vai substituir a página anterior com o menor número de acessos. Claro que se a página substituída tiver sofrido alterações, estas são gravadas em disco. O *buffer* do PostgreSQL tem uma capacidade de 1000 páginas, com 8 kb cada, mas pode ser quanto maior o utilizador quiser (*shared_buffers(integer)*) , o que incrementa o desempenho do buffer, apesar de haver sempre a limitação relativa à memória do sistema operativo que se está a usar.

Este buffer possui as seguintes flags: *Pinned* (faz lock aos atributos a serem acedidos por um determinado processo, não podendo serem acedidos por outros processos); *Dirty* (significa que os dados foram modificados desde a última leitura e podem ser atualizados); *Counter* (número de acessos à página).

De modo a se verificar a consistência dos dados, este SGDB usa ainda uma sub-rotina *"Vacuum"*, que também recolhe dados estatísticos que registam os acessos dos utilizadores às bases de dados, que serão depois utilizados pelo *Query Planner* do PostgreSQL.

2.2 File Structure

O PostegreSQL guarda todos os dados e meta-dados numa diretoria chamada *"PGDATA"*, apesar de que a partir da versão 8.0 estes dados podem ser guardados onde o utilizador quiser. Este conjunto de dados forma um *database cluster* e é possível ter no mesmo computador várias instâncias do PostegreSQL a utilizarem diferentes *database clusters*.

No PostgreSQL as tabelas são guardadas em *heaps* e estes ficheiros são da forma "slotted-page". As tabelas e índices são guardadas noutro ficheiro em disco contendo cada um deles um *"free space map"* e um *"visibility map"*. O *"free space map"* dá a informação de quanto espaço livre ainda resta num *heap* ou *índice.* O *"visibility map"* é usado para se saber que páginas contêm tuplos visíveis para as transações ativas. Se os tuplos não estiverem visíveis, então não precisam de ser tratados pela rotina *"Vaccum"*.

Como as páginas têm no máximo 8 kb e não é possível dividir uma entrada por várias páginas, então para guardar campos de tamanho elevado o Postgre usa uma técnica denominada *TOAST* (The Oversized-Attribute Storage Technique). Esta técnica comprime os campos e divide-os, associando-lhes uma tabela TOAST, em que estes atributos com serão extraídos quando o resultado da consulta for apresentado ao utilizador. Nota: Esta técnica não pode ser usada em todos os atributos.

2.3 Partitioning

O particionamento de tabelas é basicamente dividir uma tabela que seja muito grande em pedações mais pequenos. Isto melhora os tempos de acesso e atualizações às tabelas. O PostegreSQL implementa o particionamento usando um mecanismo de herança de tabelas, isto é, cada partição é criada como filho de um único pai, a tabela original, que existe apenas para representar o conjunto dos dados. Existem dois tipos de partições no PostgreSQL: *"Range Partitioning"*, em que a tabela é particionada através de um intervalos de valores definidos por uma coluna ou conjuntos de colunas; *"List Partitioning"*, em que a tabela é particionada indicando explicitamente que valores entram em cada partição.

2.4 Clustering

O PostgreSQL permite a criação de *clusters* para tabelas. Isto faz com que as tabelas passem a estar ordenadas fisicamente de acordo com dado índice criado previamente. Com esta técnica, o número de procuras nas páginas é reduzido, tornando as consulta mais rápidas. O único problema do *clustering* é que ao se inserirem e/ou atualizarem

dados na tabela, estes não ficarão ordenados pelo índice, o que implica realizar novamente um *cluster* à tabela.

2.5 PostgreSQL VS Oracle

Em relação ao *buffer management*, o Oracle também implementa o seu o próprio *buffer*, mas este é mais sofisticado. A *file structure* do Oracle é um pouco diferente, já que este implementa totalmente o seu sistema de ficheiros, o que é uma vantagem em relação ao PostgreSQL que tem que se "adaptar" ao Sistema Operativo em uso, podendo ter um nível de desempenho menor. Quanto às tabelas, por defeito, tanto o PostgreSQL como o Oracle as organizam em *heaps*. No que diz respeito às partições, o Oracle possuí mais tipos de particionamento do que o PostgreSQL, como as partições por *hash*. A grande vantagem do Oracle em relação ao PostgreSQL relativamente ao clustering por tabelas, é que este permite *multitable clustering*.

3. Indexação e Hashing

Um índice é uma estrutura de dados auxiliar que quando corretamente utilizado, permite otimizar o desempenho na procura de dados. O índice é criado sobre uma ou mais colunas de uma tabela e é mais pequeno que esta tabela visto ter menos tuplos.

Em PostgreSQL um índice é criado com o seguinte comando:
CREATE [UNIQUE] INDEX name ON table [USING method]
({ column | (expression) } [opclass] [ASC | DESC]
[NULLS { FIRST | LAST }] [, ...])
[TABLESPACE tablespace]
[WHERE predicate]

O sistema tratará de atualizar um índice sempre que a tabela sobre a qual este foi criado for modificada. E o mesmo é utilizado nas *queries* em que o optimizador considerar necessário ao invés de percorrer a tabela sequencialmente.

3.1 Tipos de índices

Os tipos de índices que o PostgreSQL suporta na sua versão actual são: B-tree, Hash, GiST, SP-GiST e GIN. Os algoritmos utilizados em cada tipo de índice são diferentes, e são aplicados de acordo com o tipo de *query*, mas por omissão, um índice é criado como sendo uma B-tree, que é a melhor escolha na maioria das situações.

3.1.1 B-Tree

Um índice B-Tree suporta igualdades e *queries* de intervalos em dados que podem ser ordenados. Em PostgreSQL, o optimizador vai considerar utilizar este tipo de índice se a coluna indexada envolver uma comparação usando um dos operadores <, <=, =, >=, > ou construções equivalentes a combinações destes operadores como BETWEEN e IN. As condições IS NOT NULL e IS NULL também podem usar este índice.

3.1.2 Hash

Os índices baseados em funções hash apenas suportam comparações simples de igualdade. O algoritmo utilizado pelo PostgreSQL foi desenvolvido por W. Litwin. Esse hashing é dinâmico, ou seja, o não possui função de rehashing. Este tipo de índice não é melhor do que o B-tree, porque consome mais recursos e não é *WAL-logged*, isto é, pode ser preciso reconstruir os índices hash após uma falha da base de dados. Por isso, o uso de índices hash é desencorajado.

3.1.3 GiST, SP-GiST e GIN

Os índices GiST (Generalized Search Tree), SP-GiST (Space Partitioning Generalized Search Trees) e GIN(Generalized Inverted Index) não são um tipo singular de índice, mas na verdade uma infraestrutura na qual muitas estratégias de indexação diferentes podem ser implementadas. Os operadores particulares com que estes índices podem ser utilizados variam de acordo com a estratégia de indexação que é especificada pelo parâmetro *opclass* no método CREATE INDEX . Os índices GIN são índices invertidos que lidam com valores com mais de uma chave.

3.1.4 Índices Multi-Colunas

Em PostgreSQL é possível criar um índice de várias colunas de uma tabela. Mas apenas índices do tipo B-tree, GiST e GIN suportam indexação em múltiplas colunas, podendo ser especificadas no máximo 32 colunas. No entanto, este limite pode ser alterado, alterando o ficheiro src\include\pg_config_manual.h.
O comando para se criar um índice multi-colunas é o seguinte:
CREATE INDEX test_mm_idx ON test (a, b);

3.2 Índices para organização de ficheiros

O PostgreSQL utiliza um *heap* para organizar as tabelas em ficheiros. No entanto, com o comando CLUSTER é possível organizar os ficheiros com base em um índice criado anteriormente:
CLUSTER [VERBOSE] tablename [USING indexname]

3.4 Combinação de índices

Uma das limitações em fazer pesquisas com índices simples, é que a pesquisa pode apenas servir *queries* onde os atributos indexados são unidos pelo operador AND. Isto é, por exemplo a *query* WHERE a = 5 OR b = 6, não permite a utilização do índice, diretamente. Para superar esta limitação, o PostgreSQL não impõem nenhuma restrição na criação de vários índices sobre os mesmos atributos, e podemos até fazer múltiplo uso do mesmo índice.

Para combinar múltiplos índices, o sistema pesquisa cada índice necessário e cria para cada índice um bitmap em memória, contendo a localização dos tuplos que satisfazem as condições dos índices. Depois, estes bitmaps são unificados através das operações lógicas AND ou OR, de acordo com a *query.* Por fim, os tuplos da tabela representada são visitados em ordem física, visto que é nesta ordem que os bitmaps estão ordenados.

Caso a query inclua uma cláusula ORDER BY, é necessário um passo extra para realizar a ordenação. Por esta razão, tendo em conta as pesquisas de índice adicionais que podem eventualmente ser necessárias, o optimizador pode escolher utilizar um índice simples mesmo tendo outros índices disponíveis.

3.5 Inconsistência Temporária das Estruturas

O PostgreSQL permite que os índices mantenham temporariamente um estado de inconsistência, o que ocorre principalmente em operações de transação.

3.6 Comparação com o Oracle

Em termos de indexação, a principal diferença entre o PostgreSQL e o Oracle é que o PostgreSQL não suporta índices *bitmap*. Apesar disto suporta outros tipo de índices que são mais completos como o GiST ou GIN.

4. Processamento e otimização de perguntas

Estes são os passos do processamento de uma *query* até a obtenção do resultado:

1. Estabelece-se uma ligação da aplicação para um servidor do PostgreSQL. A aplicação envia a *query* e espera para receber de volta o resultado transmitido pelo servidor;
2. O *parser* verifica se a sintaxe da *query* transmitida pela aplicação está correta e cria uma *query tree*;
3. O *rewrite system* pega na *query tree* criada pelo *parser,* e procura por alguma regra a ser aplicada nesta árvore;
4. O planeador/optimizador recebe a árvore reescrita, e cria um plano de execução, estimando o caminho mais barato para se chegar ao resultado final, e este plano será a entrada do executor;
5. O executor caminha recursivamente através da árvore do plano, e retorna os tuplos na forma representada pelo plano. O executor faz uso do sistema de armazenamento para percorrer as relações, realiza classificações e junções, avalia as qualificações e envia de volta as linhas derivadas.

4.1 Ligação ao servidor

Em PostgreSQL as ligações ao servidor são estabelecidas através de um modelo cliente/servidor com um processo por utilizador. Como não se sabe quantas ligações serão realizadas, é utilizado um processo mestre que cria um novo processo servidor cada vez que uma ligação é requisitada. Este processo mestre denominado *postmaster*, atende os pedidos que chegam na porta TCP/IP especificada.

Sempre que é detetada a requisição de uma nova ligação, o processo *postmaster* cria um novo processo servidor chamado *postgres*. Para garantir a integridade dos dados, os processos *postgres* comunicam entre si utilizando semáforos e memória compartilhada. Assim que é estabelecida a ligação, o cliente pode enviar as queries para o servidor. O servidor analisa o comando, cria o plano de execução, executa o plano, e retorna os tuplos obtidos para o cliente.

4.2 Etapa de Parsing

4.2.1 Parse

O *parse* recebe a query, uma *String* (em texto ASCII) e verifica a sua validade em termos de sintaxe. Se estiver correta, uma *parse tree* é gerada, senão uma mensagem de erro é retornada. O *lexer*, definido no ficheiro *scan.l* é responsável por reconhecer identificadores, palavras-chaves do SQL, etc. Para cada palavra-chave ou identificador que é encontrado, um *token* é gerado e passado para o *parser.* O *parser* está definido no ficheiro *gram.y* e consiste num conjunto de regras gramaticais e ações que são executadas sempre que uma regra é "disparada". O código das ações (em linguagem C) é usado para construir a árvore.

4.2.2 Processo de transformação

No passo anterior foi criada uma *parse tree*, esta árvore é então passada para o *rewriter* (que é uma implementação do sistema de regras do PostgreSQL), que vai aplicar um conjunto de regras que vão transformar a *parse tree* numa *query tres*. Uma *query tree* é uma representação de uma query SQL onde todos os elementos são armazenados separadamente de acordo com a seguinte estrutura:

- Tipo de comando - indica qual comando produziu a *query tree* (SELECT, INSERT, UPDATE, DELETE).
- Tabela de abrangência – armazena a lista de relações usadas na *query.*
- Relação resultante - índice na tabela de abrangência que identifica a relação onde o resultado da *query* vai ter efeito.
- Lista de alvos – lista de expressões que definem o resultado da *query*. Operações do tipo DELETE não precisam duma lista de alvos, visto não produzirem resultado.
- Qualificação – é uma expressão com um valor booleano que indica se a operação (INSERT, UPDATE, DELETE ou SELECT) deve ou não ser executada para o resultado final. Corresponde à cláusula WHERE em uma expressão SQL.
- Árvore de junção - mostra a estrutura da cláusula FROM. Para *queries* simples, como por exemplo, SELECT FROM t1, t2, t3 a árvore de junção é uma lista de itens, porque a ordem da junção é irrelevante. Mas quando utilizamos o JOIN, a junção é feita segundo uma determinada ordem. As restrições associadas a

determinadas junções (utilizando expressões ON ou USING) são guardadas como expressões qualificativas associadas a estes nós da árvore de junção. É conveniente também associar a estes nós as expressões da cláusula WHERE como expressões de qualificação. Portanto, uma árvore de junção representa ambas as clausula, FROM e WHERE, de um SELECT.

- Outros - outras partes da query *tree*, como a clausula ORDER BY, por exemplo.

A estrutura da *query tree* resultante é semelhante à *parse tree* inicial. No entanto, há algumas diferenças que podem ser destacadas, por exemplo, um nó de função (FuncCall) na *parse tree* pode ser transformado num nó de expressão de função (FuncExpr) ou num nó de agregação de funções (Aggref) na *query tree*; os tipos de dados de colunas e os resultados da expressão são adicionados à *query tree*, ao contrário do que aconteceu com a *parse tree*. Estas árvores encontram-se no ficheiro log do servidor, e é possível vê-las a alterar os parâmetros de configuração *debug_print_parse*, *debug_print_rewritten* e *debug_print_pal*.

4.3 Planeador/Optimizador

O optimizador é responsável por elaborar o melhor plano de execução possível. Para isto, ele contempla todas as formas possíveis de se executar uma *query*, e seleciona a que se espera ser mais rápida. Quando não é possível examinar todas as alternativas possíveis de execução, o PostgreSQL utiliza um "*Genetic Query Optimizer"* que é um algoritmo usado para a otimização de *queries*. Este algoritmo é usado para reduzir o tempo do planeamento da execução de *queries* mais complexas. Usando o optimizador padrão, são gerados planos das relações individualmente e o plano da junção é gerado usando a abordagem genética.

4.3.1 Geração de Planos

Os planos gerados estão diretamente relacionados com os índices disponíveis. Se uma *query* tem uma restrição de atributo e esse atributo corresponde ao atributo-chave de um índice ou se há cláusulas de ordenação na *query*, então um plano de execução é criado para o respetivo índice. Em relação às junções, as estratégias disponíveis são:

- **Nested Loop Join** - a relação da direita é varrida uma vez para cada linha encontrada na relação à esquerda. Esta estratégia é fácil de implementar, mas pode demorar algum tempo. No entanto, se a relação da direita for varrida utilizando o índice, há uma melhora significativa. É possível usar os valores da linha atual da relação na esquerda como chaves para o varrimento do índice à direita.

- **Merge Join** - *no merge join*, cada tabela é ordenada de acordo com os atributos da junção. Esta ordenação pode ser explícita ou varrendo a relação na ordem apropriada utilizando um índice na chave de junção. Após a ordenação, a junção é feita varrendo as tabelas uma única vez e unindo de acordo com os atributos da junção.

- **Hash join** - no *hash join*, a relação da direita é colocada numa *hash table*, usando os atributos de junção como chave. Depois, a relação da esquerda é varrida e os valores correspondentes dos atributos de junção são usados como chave para encontrar correspondências na *hash table* inicialmente criada.

Se a *query* envolve mais de duas relações, o optimizador examina as diferentes sequências de junção possíveis, a fim de encontrar a melhor. Se houver menos de *geqo_threshold* (12 por padrão) relações, uma busca exaustiva é realizada, a fim de encontrar a melhor sequência de junção. O planeador considera preferencialmente as junções entre as relações para as quais existe a cláusula WHERE.

Por fim, a árvore do plano de execução consiste em varreduras sequenciais ou de índice das relações base, mais nós de *nested-loop*, *merge*, ou *hash join* de acordo com o necessário, mais os passos auxiliares, como nós de classificação ou nós de cálculo de

agregação de funções. A maioria desses tipos de nós têm a capacidade adicional para fazer a seleção (descartando as linhas que não verificam a uma condição booleana especificada) e projeção. Uma das responsabilidades do optimizador é anexar as condições de seleção da cláusula WHERE e calcular as expressões de saída requeridas para os nós mais adequadas da árvore do plano.

4.4 O executor

O executor processa recursivamente o plano gerado pelo optimizador para obter o conjunto de tuplos requeridos. Isto é feito através de um mecanismo de *demand-pull pipeline*, onde uma linha é devolvida cada vez que um nó do plano é chamado.

Se a query for do tipo SELECT, o executor simplesmente retorna todas as linhas entregues pelo optimizador; se for do tipo INSERT, o executor irá inserir cada linha entregue pelo optimizador na respetiva tabela, e se for UPDATE, o executor vai inserir nas respetivas tabelas as linhas retornadas pelo optimizador (que são os únicos a serem atualizados e vêm com um identificador de linha, o *TID*), marcando as linhas identificadas como excluídos. Por fim, se for do tipo DELETE, o optimizador retorna a coluna *TID* e o executor vai marcar as linhas de destino como excluído.

4.5 Algoritmos

Os algoritmos usados para a operação de seleção são *sequencial scan*, *bitmap scan* e *index scan*. Para ordenação é usado o *external merge sort*.

4.6 Materialização

Há situações em que é preciso voltar a obter todos, ou parte dos tuplos de uma relação. Dependendo do plano utilizado, isto pode ter um custo muito caro, por isso o PostgreSQL utiliza a materialização. Por padrão a materialização no PostgreSQL está ativa e é possível descativá-la com o comando:

```
set enable_material = off;
```

4.7 Transformações de perguntas

O PostgreSQL permite a definição de regras de reescrita de perguntas no servidor da base de dados.

A sintaxe usada é a seguinte:

CREATE RULE regra **AS ON { SELECT | INSERT | UPDATE | DELETE } TO**

tabela **[WHERE** predicado **] DO [INSTEAD]{ NOTHING |**

comando **| (** comando**;** comando **…)};**

As vistas são implementadas sobre o sistema de regras, sendo assim o comando:

CREATE VIEW vista **as SELECT * FROM** tabela**;**

Será então transformado para:

CREATE TABLE vista *(lista das colunas da tabela)* **CREATE RULE** "_RETURN" **AS**

ON SELECT TO vista **DO INSTEAD SELECT * FROM** tabela**;**

As regras **SELECT** são aplicadas como o último passo na transformação de perguntas, mesmo que os comandos SQL sejam: **INSERT**, **UPDATE** ou **DELETE**. Ao contrário das regras sobre operações de escrita, as **SELECT** modificam a árvore da interrogação localmente, em vez de criarem uma nova.

4.8 Estimativas

O PostgreSQL guarda dois tipos de estimativas que são Histogramas e Mapeamento lógico-físico.

Os Histogramas permitem ao optimizador determinar exatamente quantos tuplos de cada valor existe numa tabela ao invés de utilizar alguma heurística que pode sempre

conter uma margem de erro. Os Histogramas só podem ser aplicados a tablas que possuam algumas colunas que estão definidas com espaço finito de valores.

O Mapeamento lógico-físico permite uma melhoria no algoritmo de índex scan, pois efetua uma correlação entre a ordenação lógica e física das linhas de uma tabela.

4.8.1 Parametrização e construção de uso de estimativas

Existem alguns comandos que o seu principal objetivo não é alteração ou parametrização da construção ou uso de estimativas mas que indiretamente alteram as estimativas, tais como:

VACCUM – Limpa do disco tuplos já apagado, mas que continuavam em memória.

ANALYZE – Permite analisar e colecionar estatísticas de uma dada tabela e posteriormente guarda-las para futuras operações de otimização, importante ter em conta o facto de que repetidas operações do comando *analyze* podem originar estimativas diferentes visto que este comenda se baseia em amostras aleatórias dos dados.

CREATE INDEX – Através deste comando é possível posteriormente guardar estatísticas acerca da coluna para a qual foi criado o índex.

4.9 Planos

O PostgreSQL permite visualizar o plano que o optimizador constrói atrases do mecanismo *explain* e também é possível usar o comando *analyze* em conjunto para que a pergunta seja realizada e assim é visto o tempo estimado *vs* tempo efetivo.

4.10 PostgreSQL VS Oracle

A grande diferença nesta secção é que o PostgreSQL não suporta o mecanismo de *hints* enquanto que o Oracle suporta.

5-Gestão de transações e controlo de concorrência

O PostgreSQL dispõe de um grande conjunto de mecanismos de gestão de transações e controlo de concorrência. Isto é indispensável num SGBD, porque existe a necessidade de se garantir a integridade e consistência dos dados numa base de dados que sofre múltiplos acessos. Como foi estudado nas aulas, para que a integridade e consistência dos dados seja garantida, o SGBD respeita as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade).

5.1-Controlo de concorrência

5.1.1-MVCC (MultiVersion Concurrency Control)

Um dos dois mecanismos de controlo de concorrência de que o PostgreSQL dispõe é o *MVCC*, e este é usado por omissão pelo SGBD. Com este mecanismo, o SGBD mantém diferentes versões da base de dados guardadas e quando se efetuam *queries* sobre esta, cada transação vê apenas o *snapshot* de acordo com a sua visão da base de dados, impedindo assim que tenha acesso a dados inconsistentes que provêm de outras transações concorrentes.

5.1.2-Explicit Locking

O segundo mecanismo de controlo de concorrência é o *explicit locking*. Os *locks* são usados quando o *MVCC* não se comporta como desejado. Assim, através de *locks* é possível estabelecer regras de acesso a uma determinada zona da base de dados. No PostgreSQL os *locks* existentes são:

- **Table-level Locks:**

  LOCK [TABLE] name [, ...] [IN lockmode MODE] [NOWAIT]

  Onde *lockmode* pode ser:

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

Um *lock table* dura até ao final de uma transação. Mesmo assim há que ter alguns cuidados ao usar o *lock table*, já que podem existir *locks* que entrem em conflito na mesma transação. A documentação do PostgreSQL tem a seguinte tabela que mostra onde podem ocorrer estes conflitos entre *locks.*

Requested Lock Mode	Current Lock Mode							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

Definição dos *lock modes*:

ACCESS SHARE: Os acessos feitos às tabelas podem ser partilhados, desde que sejam apenas leituras;

ROW SHARE: É utilizado nas operações *SELECT FOR UPDATE* e *SELECT FOR SHARE;*

ROW EXCLUSIVE:É utilizado quando se pretendem modificar os dados numa tabela;

SHARE UPDATE EXCLUSIVE: Protege a tabela contra modificações concorrentes do *schema* e contra a rotina *vacuum*;

***SHARE*:** Fornece proteção contra alterações concorrentes de dados;

***EXCLUSIVE*:** Apenas podem existir existir queries em paralelo numa tabela;

***ACCESS EXCLUSIVE*:** Garante que apenas o dono do *lock* acede à tabela. É o *lock* usado por omissão caso nenhum seja especificado.

- **Tuple-level Locks:**

  Os *locks* de tuplos podem ser *shared* ou *exclusive.* Aqui, o *lock* também só dura até ao final de uma transação, ou caso seja feito um *commit* ou *rollback.* Os *locks* de tuplos não afetam as *queries*, apenas afetam *writes* sobre os mesmos tuplos.

Um *shared lock* é obtido através do comando "*select for share.* Com este tipo de *lock*, qualquer transação pode obter o *lock*, mas não será possível que outra transação altere ou apague um *lock* num tuplo sobre o qual já existe um *shared lock*. Se isto acontecer, a transação fica trancada até que o *shared lock* seja libertado.

Um *exclusive lock* é obtido quando um tuplo é atualizado ou apgado. Para se obter um *lock* deste tipo sem se modificar um tuplo, é necessário usar o comando *"select for update".*

- **Advisory Locks:**
  Os *advisory locks* são um mecanismo usao pelo PostgreSQL que permite criar *locks* a nível de uma aplicação. Uma vez feito um *advisory lock*, este é mantido até que seja expressamente *unlocked* ou que se termine a aplicação. Este tipo de *locks* pode ser usado várias vezes pelo mesmo processo, mas no entanto para cada pedido de *lock* tem que haver um pedido de *unlock* antes de ser atribuído um novo *lock.*

5.1.3-Deadlocks

O uso de *explicit locking* eleva a possibilidade de ocorrerem *deadlocks*. Estes ocorrem quando, por exemplo, uma transação quer obter um *lock* que outra transação já possui, e essa por sua vez quer obter o *lock* da primeira. Para contornar esta situação, o PostgreSQL implementa uma deteção automática de *deadlocks*, fazendo com uma das transações seja abortada para que a outra possa terminar.

É importante referir que os *deadlocks* podem ocorrer mesmo não se usando *explicit lock*. Como se pode verificar pelo exemplo seguinte, retirado da documentação do PostgreSQL, em que duas transações tentam modificar uma tabela, e a primeira executa o seguinte comando:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;
```

Este comando efetua um *lock* a nível de tuplos com o número de conta (11111). A segunda transação executa o comando:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;

UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;
```

O primeiro *update* efetua um *lock* ao nível dos tuplos na linha especificada, por isso esse tuplo é atualizado. Mas no segundo *update*, este verifica que a linha que está a tentar atualizar já está *locked*, ficando então à espera. Entretando a primeira transação executa:

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

Assim, a primeira transação tenta fazer *lock* aos tuplos, mas não sucede porque a segunda já detém esse *lock*, ficando bloqueada à espera que a segunda transação termine, o que nunca irá acontecer já que esta também se encontra bloqueada. Quando isto acontece, o PostgreSQL simplesmente aborta uma das transações.

A melhor defesa contra *deadlock*s, é o utilizador assegurar-se de que todas as aplicações que usam uma base de dados obtêm *locks* numa ordem consistente. No exemplo referido, se as duas transações tivessem atualizado as linhas seguindo a mesma ordem, não teria ocorrido *deadlock.* Em suma, se nenhum *deadlock* for detetado, uma transação que procure um *lock* a nível de tuplos ou tabelas, vai esperar indefinidamente que os conflitos entre *locks* sejam resolvidos, ou seja, as aplicações deixam transações abertas indefinidamente, o que consume muitos recursos.

5.2-Locking e índices

O PostgreSQL implementa diferentes políticas de tratamento de acessos não bloqueados aos dados duma tabela, dependendo do índice em uso.

- **B-Tree e GiST (Generalized Search Tree):** Os *locks* são libertos após cada inserção ou consulta de um tuplo. Este índice é o que oferece maior capacidade de concorrência sem ocorrerem *deadlocks*.

- **Hash:** Neste tipo de índice, os *locks* são libertos após o processamento de todos os *buckets*. Estes índices oferecem uma boa capacidade de concorrência, apesar

da preservação dos *locks* ser maior do que o índice anterior, e a não ocorrência de *deadlocks* não é totalmente garantida.

- **GIN (Generalized Invertex Index):** Aqui, os *locks* são libertos após cada inserção ou consulta de um tuplo. Este tipo de índice poderia ser tão bom como o primeiro aqui apresentado, mas no entanto, as inserções de valores numa estrutura *GIN* resultam na inserção de várias chaves por tuplo, o que resulta em mais trabalho para apenas uma inserção.

5.3-Gestão de Transações

O SQL define quatro níveis de isolamento de transações, de modo a lidar com situações em que o isolamento possa estar comprometido. Como é o caso dos *dirt reads* (uma transação lê dados de uma transação concorrente que ainda não fez *commit*), dos *nonrepeatable reads* (uma transação faz uma leitura diferente dos dados em leituras subsequentes, o que indica que outra transação que já fez *commit* modificou esses dados), e dos *phantom reads* (uma *query* devolve um conjunto de tuplos diferentes do já devolvido anteriormente, o que significa que outra transação concorrente alterou os tuplos e fez *commit* entretanto):

Os quatro níveis de isolamento para contornar estes erros são:

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Com o comando ilustrado a seguir, o utilizador pode decidir que nível de isolamento usar nas suas transações.

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

where transaction_mode is one of:

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

De referir que dos quatro níveis de isolamento existentes o PostgreSQL apenas implementa o "*read commited"* e o "*seriazable"*. Isto é justificado pelo fato de estes serem os dois níveis de isolamento que funcionam corretamente com o mecanismo MVCC. O "*read commited"* é o nível de isolamento usado por omissão no PostgreSQL. Neste nível de isolamento, uma *query* "SELECT" (sem as cláusulas FOR, UPDATE, SHARE) apenas lê os dados que foram *commited* no ínicio da operação, ou seja, nunca lê dados que não foram commit por outras transações concorrentes. Para as restantes cláusulas, o comportamento é semelhante, à exceção de que os tuplos alvos da *query* já podem ter sido atualizados. O nível "*serializable*" é considerado o nível de isolamento mais estrito, na medida em que são feitos *savepoints* após cada instrução, de modo a tentar simular a serialização de possíveis acessos. Assim sucessivas instruções SELECT veem sempre os mesmos dados, independentemente dos acessos concorrentes de outras transações.

5.3.1- Savepoints

O PostgreSQL suporta *Nested* Transactions através da utilização de savepoints. Este mecanismo possibilita reverter uma transação a um determinado estado anterior. Um "*savepoint*" representa um determinado local numa transação que permite que todos os comandos executados depois da sua definição possam ser revertidos. É ainda possível anular todas as modificações efetuadas após um *savepoint*, utilizando o comando "*rollback to savepoint"*, bem como libertar um *savepoint* através do comando "*release savepoint*".

Exemplo de como estabelecer um *savepoint* e posteriormente desfazer os efeitos de todos os comandos executados após a sua criação (retirado da documentação do PostgreSQL):

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

Esta transição insere os valores 1 e 3, mas não o valor 2.

Exemplo de como estabelecer um *savepoint* e mais tarde destruí-lo:

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

Esta transição insere os tuplos 3 e 4.

5.4-Write Ahead-Log

O PostgreSQL usufrui ainda um mecanismo que garante a integridade dos dados em disco. O *WAL* guarda todas as transações que foram executadas e ainda não estão armazenadas em memória persistente. Este mecanismo aumenta a performance do sistema por evitar que todas as transações sejam armazenadas em disco, e também garante a integridade dos dados, porque caso aconteça algum problema com o sistema e todas as operações ainda não tiverem sido escritas em disco, há sempre a possibilidade de se recorrer ao *log* para atualizar a memória persistente. Apesar do *log* também ter que ser gravada em disco, a complexidade desta operação é menor do que gravar todas as transações em disco de imediato. Se o utilizador (desde que possua privilégios) desejar, pode forçar o armazenamento do *log* em memória usando o comando "*checkpoint*".

5.5-PostgreSQL VS Oracle

.A diferença entre os PostegreSQL e o Oracle prende-se aos níveis de isolamento de transações, em que o Oracle usa três níveis (*serializable*, *read only* e *read commited*), enquanto que o PostgreSQL apenas usa dois níveis (*serializable* e *read commited*). Isto não faz com que o PostgreSQL fique muito atrás do Oracle, já que cada sistema tem as suas vantagens e desvantagens, e neste capítulo estas são mínimas.

6. Suporte para bases de dados distribuídas

O PostgreSQL tem suporte para bases de dados distribuídas homogéneas, através de replicação de dados por vários servidores.

Nos dias de hoje é cada vez mais natural as bases de dados serem distribuídas não só para segurança dos dados mas também para acessos concorrentes e mais rápidos pelo mundo fora. Mas para manter uma base de dados distribuída são gerados alguns problemas que têm que ser resolvidos pelo sistema de base de dados.

6.1 Streaming Replication

O PostgreSQL contém uma ferramenta chamada "*Streaming Replication"* que embora não seja uma ferramenta para bases de dados distribuídas permite a replicação de dados.

O *Streaming Replication* funciona da seguinte forma, existe um servidor primário (master) e existem outros servidores secundários (slaves). A replicação dos dados é feita através de *streams* de registos WAL (Write-Ahead Logs) de modo assíncrono, não existindo *overhead* no servidor primário e os servidores secundários podem executar perguntas. Como é assíncrono existe uma maior velocidade nas transações mas é possível ocorrer perda de dados caso a base de dados tenha alguma falha.

6.2 Cascading Replication

A *Cascading Replication* permite que os servidores secundários também possam enviar as alterações do *WAL* para outros servidores secundários que estejam desatualizados para assim evitar uma sobrecarga no servidor primário.

6.3 Synchronous Replication

A replicação síncrona permite que se confirme que todas as alterações efetuadas por uma transação foram transferidas para um servidor secundário.

Quando se fala em replicação síncrona, cada *commit* de uma transação de escrita irá esperar até que tenha a confirmação de que o *commit* foi escrito no *log de transações* de ambos os servidores (primário e secundário).

Para transações apenas de leitura e *rollbacks* de uma transação não necessitam de esperar por repostas por parte dos servidores secundários.

Todas as sub-transações não esperam por *commits* apenas a transação de nível superior, assim como as transações de longa duração não esperam por nenhum *commit* somente para o ultimo. Todas as ações do protocolo *two-phase commit* requerem que se espere por *commits*.

6.4 Failover

Neste método no caso do servidor primário falhar um dos servidores secundários é promovido a primário e tem que efetuar as ações de *failover*.

Caso um servidor secundário falhe simplesmente tem que efetuar as ações de recuperação, no caso de não ser possível a sua recuperação então é necessário a substituição desse servidor secundário.

6.5 PostgreSQL VS Oracle

O Oracle permite a criação de uma base de dados distribuídas e replicação enquanto o PostgreSQL apenas permite a replicação de dados.

Os termos base de dados distribuídas e replicação de base de dados estão relacionados mas são distintos. Num sistema de base de dados distribuída pura o sistema gere apenas uma copia de todos os dados enquanto na replicação refere-se a copias de uma mesma base de dados por locais distintos.

7-Outras caraterísticas do PostgreSQL

7.1-Suporte XML

A grande vantagem de um SGBD ter suporte XML é a facilidade que este fornece no que respeita ao armazenamento de dados. Para se transformar um documento ou conteúdo num XML usa-se o seguinte comando:

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

Exemplo:

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

Também se pode transformar um tipo de dados XML em outro tipo, através do seguinte comando, em que *type* é *character*, *character varying* ou *text*.

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

Uma desvantagem de se usar XML é que não existe maneira de comparar valores deste tipo, o que impossibilita resolver *queries* sobre este tipo de dados. Como não existe comparação de dados em XML é também impossível criar índices sobre colunas deste tipo de dados, mas este problema pode ser contornado se se transformar os dados XML em texto e fazer indexação do valor que se pretende ou fazer índices sobre expressões XPath.

Para se usarem consultas XPath, XSLT e outras, basta instalar o módulo "xml2" do PostgreSQL. Com isto, pode-se usar a função "*x-path_table*" permite construir uma tabela com consultas *Xpath* sobre a uma tabela.

xpath_table(text key, text document, text relation, text xpaths, text criteria) returns setof record

Em que *key* é o nome do campo a ser utilizado como chave, *document* o nome do atributo que contem o documento xml a ser processado, *relation* o nome da tabela ou view que contém o atributo *document*, *xpaths* contém as expressões xpath separadas por "|" e *criteria* o conteúdo da cláusula *where.* Exemplo:

```
CREATE TABLE test (
    id int PRIMARY KEY,
    xml text
);

INSERT INTO test VALUES (1, '<doc num="C1">
<line num="L1"><a>1</a><b>2</b><c>3</c></line>
<line num="L2"><a>11</a><b>22</b><c>33</c></line>
</doc>');

INSERT INTO test VALUES (2, '<doc num="C2">
<line num="L1"><a>111</a><b>222</b><c>333</c></line>
<line num="L2"><a>111</a><b>222</b><c>333</c></line>
</doc>');

SELECT * FROM
  xpath_table('id','xml','test',

'/doc/@num|/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
              'true')
  AS t(id int, doc_num varchar(10), line_num varchar(10), val1 int,
val2 int, val3 int)
WHERE id = 1 ORDER BY doc_num, line_num
```

O exemplo produz uma tabela com os seguintes valores:

```
id | doc num | line num | val1 | val2 | val3
---+---------+----------+------+------+------
 1 | C1      | L1       |    1 |    2 |    3
 1 |         | L2       |   11 |   22 |   33
```

7.2-Segurança e Autenticação

O PostgreSQL oferece mecanismos de proteção do SGBD. Para autenticação de utilizadores que se pretendam conectar a uma base de dados tem-se: *Trust, Password, GSSAPI, SSPI, Kerberos, Ident-based, LDAP, RADIUS, Certificate, PAM*, entre alguns outros. Cada um destes mecanismos possuí uma forma diferente de configuração, e o caso de existirem tantos mecanismos de autenticação prende-se com o facto do PostgreSQL ser *open source*.

7.3 Mecanismos object/relational

O PostgreSQL sendo uma base de dados Object-Relational, significa que, ainda tem suporte para armazenar os dados em forma de objetos.

7.4 Ferramentas

O PostgreSQL tem várias ferramentas ao dispor, entre elas são:

pgAdmin e PGAccess

Uma ferramenta de administração para o PosgreSQL é a pgAdmin III. Esta ferramenta oferece uma interface gráfica que permite ao utilizador gerir graficamente todos os aspetos relacionados com a base de dados, desde a sua criação até ao processamento de perguntas.

Uma alternativa à interface gráfica é a ferramenta *psql*, que permite realizar as mesmas operações que o pgAdmin III, mas através de uma linha de comandos.

Database Master e DBTools Manager

Estas ferramentas permitem que se efetue *reports* da base de dados, mas o nível de reporting oferecido varia entre todas as ferramentas deste tipo. Algumas possibilitam apenas relatórios de dados ou metadados enquanto outras também oferecem a possibilidade de visualizar a informação em gráficos e exportar dados para PDF ou XML.