Sistemas de Bases de Dados

2014

PostgreSQL

Docente: José Alferes

Grupo 03
David Gomes - 41647
João Costa - 41726
Mussa Amin - 35309

Índice

Conteúdo

1 Introdução

Neste trabalho foi feito um estudo acerca de um dos mais conhecidos Sistemas de Gestão de Bases de Dados, o PostgreSQL. Procurou-se fazer uma análise que abordasse as principais características e mecanismos que fazem parte deste SGBD – desde o modo como lida com armazenamento de dados; a capacidade de criar índices que melhoram a procura a partir de uma expressão SQL; o processamento e optimização dessas mesmas expressões; gestão de transações e o modo como o sistema gere acessos concorrentes aos dados; suporte a bases de dados distribuídas e a forma como a base de dados pode ser recuperada através de mecanismos de replicação.

No final do relatório, é também realizada uma breve comparação com um outro SBDG já nosso conhecido - o Oracle.

1.1 Introdução Histórica

O PostgreSQL, considerado um dos melhores SGBD, muito bem aceite no mercado e com um dos maiores conjuntos de utilizadores, começou for ser uma tentativa de corrigir alguns dos maiores problemas de um outro sistema, o Ingres.

O seu criador, Michael Stonebraker, havia concebido esse sistema inicial em 1982, na Universidade de Berkeley na Califórnia. Após uma tentativa de o comercializar, voltou mais tarde para o meio académico de forma a desenvolver uma versão evoluída que colmatasse os tais problemas. Daí surgiu o Postgres, que agora já lidava bem com o modelo de BD relacional e com tipos.

Mais à frente, o projecto Postgres foi abandonado, até que em 1994, Andre Yu e Jolly Chen, dois estudantes de Berkeley, desenvolveram e acrescentaram um interpretador de SQL, substituindo aquele que o Postgres continha. A partir desse momento, o novo sistema, já com o nome Postgres95, começou a ser partilhado pelos utilizadores da Internet em *open-source*, o que facilitava imenso o seu desenvolvimento. O PostgreSQL continua assim a evoluir e a ganhar novas formas de lidar com os complexos problemas que estão relacionados com as bases de dados.

2 Armazenamento e file structure

Neste capítulo iremos abordar o armazenamento e estrutura de ficheiros e organização dos registos.

2.1 Sistema de ficheiros

O armazenamento de dados utilizadas pelo PostgreSQL é feito na directoria "data" localizada na pasta de instalação do SGBD, ou seja, usa o sistema de ficheiros do sistema operativo, ficando dependente das estruturas nela implementadas mas com a vantagem de não ser necessário criar ferramentas de gestão de ficheiros, pois o próprio sistema operativo fornece. Difere do Oracle 11g na medida em que este usa o seu próprio sistema de ficheiros, tendo maior controlo nos dados inseridos e guardados na base de dados. Neste caso será preciso desenvolver mecanismos e ferramentas que permitam disponibilizar informação e gerir os ficheiros de forma correcta.

2.2 Organização de ficheiros

Cada tabela e índice são guardados num array de páginas com valor fixo, normalmente 8 kB, sendo que no caso particular dos índices a primeira página é reservada para guardar informações de controlo.

As *slotted pages* guardam os registos, sendo constituídas por um *Header* (armazenada a informação geral e apontadores para o espaço livre), *Item Id Data* (array de apontadores para os registos), *free space* (espaço liberto para novos dados), *Items* (registos já guardados) e *Special Space* (espaço para se efectuar acessos por índices).

2.3 Buffers

Ao criar uma base de dados com grande número de dados armazenados, os custos de acesso à memória e ao disco são um problema que importa minimizar o mais possível, e é com esse objectivo que o PostgreSQL implementa um sistema de Buffer Management. Este sistema comporta-se como uma cache, ou seja, as operações ao serem executadas escolhem um *buffer* de partida e movem-se circularmente no *array*. A navegação nos *buffers* é feita por um *clock-sweep*, que sequencialmente percorre o *array* até ao limite, voltando finalmente à posição 0, recomeçando a partir daí. Os *buffers* podem ser marcados de acordo com o seu estado, sendo que o estado **PINNED** indica que o buffer está a ser usado por um

processo, impedindo a sua utilização por outro, e **DIRTY** que indica que o conteúdo do *buffer* foi altera desde a última leitura de memória estável.

2.4 Partições

O PostgreSQL permite um mecanismo de partição de tabelas, dividindo tabelas de grandes dimensões em outras mais pequenas. Este mecanismo, embora aumente a complexidade das execuções das operações dado que o conteúdo de uma tabela fica desmembrado em várias, se for feito de forma correta traz vantagens aquando da consulta em tabelas de grande dimensão, especialmente quando os dados mais acedidos estão divididos numa partição.

Disto isto, há duas formas de particionamento suportadas pelo PostgreSQL:

Range Partitioning - A tabela é dividida por intervalos definidos por uma coluna chave ou conjunto de colunas, sem que os intervalos dos valores associados a diferentes partições se sobreponham. Um exemplo seria dividir de acordo com um intervalo de datas.

List Partitioning - A tabela é dividida indicando que valores chave aparecem em cada partição.

2.5 Table clustering

Table Clustering permite agrupar dados de uma tabela em espaços contíguos nas unidades de armazenamento, permitindo um ganho de eficiência já que o acesso aos mesmos dados é feito mais rapidamente.

O *cluster* de uma tabela é efectuado usando o seguinte comando: CLUSTER table_name [USING index_name]

Este método, implementado pelo PostgreSQL, reorganiza a tabela em disco tendo por base o índice inserido. O *multitable clustering*, apesar de em certas instâncias ser benéfico (nomeadamente *queries* que usam junções de tabelas com frequência), não é suportado pelo PostgreSQL.

3 Indexação e Hashing

Neste capitulo abordaremos os tipos e formas de indexação oferecidos pelo PostgreSQL, bem com os detalhes de implementação.

O correcto uso dos índices são uma forma de melhorar o desempenho das bases de dados de grandes dimensões, permitindo desta forma serem encontrados e retirados os registos de uma forma muito mais eficaz e eficiente.
No PostgreSQL a sintaxe para a criação de um índice é a seguinte:

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ]
  ON table [ USING method ]
    ( { column | ( expression ) } [ opclass ] [ ASC |
      DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace ]
    [ WHERE predicate ]
```

3.1 Tipos de Índices

O PostgreSQL , como a maioria dos SGBD suporta vários tipos de índices nomeadamente B-tree, Hash, GiST(Generalized Search Tree), SP-GiST (Space-Partitioned GiST) e GIN(Generalized Inverted *Index*).

O DBMS em estudo cria por omissão, através do comando CREATE INDEX os índices B-tree. As B-tree são uma estrutura de dados que apresentam bons resultados para consultas que envolvem a coluna indexada e os operadores de comparação (> >=, <, <=), podendo devolver também os resultados ordenados segundo um determinado atributo. Também são suportados operadores como BETWEEN, IN, IS NULL ou IS NOT NULL.

No PostgreSQL os índices de Hash são uma implementação do hashing linear, e assim sendo, apenas lidam com comparações simples de igualdade, ou seja, *o query planner* irá apenas considerar utilizar um índice de Hash quando a coluna indexada está evolvida numa comparação utilizando o operador =. O Hash é dinâmico, e por isso não possui função de rehashing.

Os índices GiST suportados pelo PostgreSQL não são um tipo singular de índices, mas sim uma infra-estrutura no qual são implementados diferentes

estratégias de indexação. Deste modo os operadores que vão resultar no uso dos índices GiST dependem da estratégia de indexação usada. Estes índices podem ser usados para optimizar as pesquisas "nearest-neighbor".

Os índices SP-GiST suportados pelo PostgreSQL são semelhantes aos índices GiST, oferecendo também uma infra-estrutura que suporta vários tipos de pesquisas.

O outro tipo de índices suportados pelo PostgreSQL são os índices GIN. Tal como os índices GiST e SP-GiST, os índices GIN suportam diferentes tipos de estratégias de indexação definidos pelo programador. Este tipo de índice serve para efectuar pesquisas em texto.

3.2 Outras Variações

Alem dos índices mais gerais acima referidos, para aumentar o desempenho das consultas, o PostgreSQL permite outras variações como índices multi-coluna, índices únicos, índices com expressões e índices parciais. Segue-se de seguida uma descrição de cada uma das variações.

- Índices multi-coluna - O PostgreSQL permite criar índices multi-coluna, sendo estes suportados pelos índices B-tree, GiST e GIN e tendo no máximo 32 colunas. Um índice B-tree com várias colunas pode ser utilizado com condições de consulta que envolvam qualquer subconjunto de colunas do índice, sendo mais eficiente o índice quando as restrições são sobre as colunas mais à esquerda.
- Índices únicos - Estes índices são apenas suportados pelos índices B-Tree. São usados para garantir a unicidade dos valores de cada coluna ou unicidade dos valores combinados de mais de uma coluna. Na criação de uma chave primária ou na criação de uma restrição, este tipo de índices são criados automaticamente pelo PostgreSQL.
- Índices por expressões – Este tipo de índices são uteis para obter o acesso mais rápido aos dados com base nos resultados de computações. No entanto estes índices são relativamente caros de manter uma vez que as expressões derivadas têm de ser computadas para cada linha da tabela no momento de inserção e modificação.
- Índices Parciais - Um índice parcial é construído sobre um subconjunto de uma determinada tabela, sendo este subconjunto definido pela expressão condicional designado por predicado do índice parcial. Este índice contém apenas entradas para os registos da tabela que satisfazem este predicado. Os índices parciais ajudam a reduzir significativamente o tamanho dos

índices, aumentando a velocidade de pesquisa. Normalmente estes índices são usados para evitar a indexação dos valores repetidos.

3.3 Estruturas Temporariamente Inconsistentes

No PostgreSQL é permitida a utilização de estruturas temporariamente inconsistentes. Para tal é utilizada a cláusula DEFFERABLE aquando da criação de uma tabela. Posteriormente durante a especificação das restrições é possível especificar o período de verificação de integridade dos índices, isto é, se se pretende que a verificação da integridade seja efectuada apenas no final da transacção ou se seja verificada logo no inicio após a introdução do comando (por definição o PostgreSQL faz a verificação logo após a introdução de qualquer comando).

4 Processamento e optimização de perguntas

Neste capitulo serão abordados os passos para o processamento de uma query.

4.1 Processamento da query

Parser - O parser verifica se existem erros de sintaxe numa dada query, e se a sintaxe estiver correta, o parser transforma a query numa parse tree. Uma parse tree é uma estrutura de dados que representa a consulta numa forma normal e não ambígua. Após este processo de parsing, a parse tree é deixada ao critério do planner/otimizador.

Planner - O planner é responsável por examinar a parse tree por forma a encontrar todos os planos possíveis para executar a query.

Executor - Após ser escolhido o plano de execução (o mais barato) , este é executado pelo executor devolvendo os dados mas relevantes da base de dados.

4.2 Operações Básicas

4.2.1 Seleção

Os algoritmos de selecção suportados pelo PostgreSQL são o sequential scan, índex scan, índex-only scan e bitmap índex scan.

De seguida são descritos cada um dos algoritmos.

Sequential Scan - O algoritmo de selecção mais básico é o sequential scan. É o algoritmo mais indicado para as consultas mais simples em tabelas de tamanho reduzido. Este algoritmo começa no inicio de uma tabela e avalia para cada registo as condições da query, e se estas fores satisfeitas o registo é adicionado ao resultado. Este algoritmo é escolhido por parte do otimizador se não existirem índices que possam ser usados para satisfazer a query.

Index Scan - Este algoritmo percorre os índices (primários ou secundários) e devolve as linhas da tabela por ordem do índice, e não pela ordem que estas se encontram no disco.

Index-Only Scan (a partir da versão 9.2) - Este algoritmo possui as mesmas propriedades que o índex scan, no entanto não necessita de aceder à tabela física para obter os dados.

Bitmap Index Scan - Este algoritmo é mais rápido para as consultas que utilizam dados estatísticos, dados que podem ser adicionados a um bitmap. O otimizador irá utilizar este algoritmo se existirem bitmaps associados à tabela que tornam a query mais rápida. Este algoritmo é mais usado para as consultas do tipo SELECT count(*).

4.2.2 Ordenação

Os algoritmos de ordenação implementados pelo PostgreSQL são o quicksort e o external sort.

Quicksort - Quando as tabelas cabem em memória, é usado o algoritmo quicksort.

External sort - Este algoritmo é usado quando as relações não cabem em memoria. Numa primeira fase as relações são partidas em pedaços por forma a que caibam em memória, sendo ordenados e armazenados em ficheiros temporários. Numa segunda fase estes ficheiros temporários são combinados num só.

4.2.3 Junção

Os algoritmos de junção implementados pelo PostgreSQL são o nested loop join, merge join, hash join e indexed nested loop join.

Nested loop join - Numa junção a relação da direita apenas é percorrida uma vez para cada registo da relação da esquerda.

Merge join - Algoritmo em que ambas as relações são ordenadas pelos atributos da condição de junção. Este algoritmo apenas percorre a relação uma vez.

Hash join - Este algoritmo é usado quando numa consulta a condição de junção é uma igualdade. Neste algoritmo a relação da direita é carregada para uma hashtable através do operador Hash, usando como chave o atributo da condição de junção. Por sua vez, a relação da esquerda é percorrida, e usando o atributo correspondente como chave, obtém os registos da hashtable.

4.3 Expressões complexas

Existe um mecanismo associado ao *pipelining* que é implementado pelo PostgreSQL, e que se baseia no processamento recursivo de planos, transmitidos do planner ao executor. Cada nó de plano que é chamado, ou avisa o executor de que já não envia mais tuplos ou devolve no mínimo um tuplo. Desta forma, para expressões que impliquem o uso de diversos planos, o método baseia-se no cálculo e retorno, por parte de cada nó, do próximo tuplo, sempre que é chamado.

Este mecanismo é usado para avaliar os quatro tipos básicos de uma pergunta de SQL – SELECT, que se limita a enviar cada tuplo resultante da árvore de planeamento para o cliente; INSERT, em que cada tuplo é inserido numa tabela especificada para a operação, através de um plano especial do tipo ModifyTable; UPDATE, em que cada nó que deve ser actualizado é apagado, e um novo, que mantém o ID do tuplo antigo, é criado com os valores mais recentes, a partir de um nó ModifyTable que contém esses mesmos valores; DELETE, cada tuplo que cujo ID seja igual ao que é passado ao nó MofidyTable é apagado.

4.4 Mecanismos para visualização de planos

O PostgreSQL permite que que sejam visualizados os planos de execução das consultas. Através do comando EXPLAIN, é mostrada a informação sobre que plano de execução será usado para a query dada como argumento. É possível também obter o custo real de uma query, através do comando ANALYSE (opcional). A sintaxe é a seguinte:

```
EXPLAIN [ANALYZE] [VERBOSE] query;
```

4.5 Estatísticas

Um componente muito importante dos sistemas de gestão de bases de dados são as estatísticas, visto que para gerar o plano de execução mais eficiente para uma dada query, o otimizador necessita de estimar o numero de linhas que são devolvidas pela query para depois proceder a escolha do plano a executar. As estatísticas são geradas através do comando ANALYSE, no entanto estas poderão ser diferentes para cada execução do comando sobre a mesma query pois este comando baseia-se em amostras aleatórias dos dados.

A sintaxe deste comando é a seguinte:

```
ANALYSE [VERBOSE] [ table_name [ (column_name [, …]) ] ]
```

5 Gestão de transações e controlo de concorrência

O PostgreSQL usa o modelo *Multiversion Concurrency Control* (MVCC). Este modelo multiversão permite que as *queries* de escrita de leitura não se bloqueiem mutuamente, permitindo um nível de concorrência mais eficaz. No entanto também tem as suas desvantagens, dado que o sistema guarda várias versões do mesmo tuplo (dado que uma *snapshot* consistente da base de dados é feita aquando duma transacção), logo consumindo muito espaço em disco. Para minimizar esta situação, o PostgreSQL tem o comando VACCUM que permite ao utilizador libertar espaço em disco.

Níveis de isolamento: No PostgreSQL, embora seja possível usar os quatro níveis de isolamento do *standart* do SQL, quando o nível sem isolamento (Read uncommited) é escolhido, na realidade o nível usado é o Read commited. Logo concretamente só existem três níveis de isolamento: **Read commited, Repeatable read** e **Serializable**. Para se definir o nível de isolamento de uma trasacção usa-se o comando SET TRANSACTION

Seguidamente explicamos cada tipo de nível de isolamento:

Read Committed - Este é o nível de isolamento por defeito do postgreSQL. Neste nível o sistema vê uma snapshot da base de dados cujas transacções fizeram commit aquando da execução do comando. Quer isto dizer que qualquer actualização numa transacção concorrente não é vista numa querry SELECT.

Repeatable Read - Este nível de isolamento diferencia-se do Read Committed pois o sistema apenas vê os dados *committed* antes da transacção e não durante a execução do comando. Este modo garante uma visão estável da base de dados.

Serializable - Sendo o nível mais restritivo de isolamento, as transacções apresentam um resultado sequencial, simulando escalonamentos serializáveis em vez de concorrentes. Este nível de escalonamento funciona tal como o Repeatable Read, no entanto as condições que poderão trazer alguma inconsistência às transacções são monitorizadas.

Locks explícitos - De forma a gerir acessos concorrentes aos dados das tabelas da base de dados, o PostgreSQL fornece vários modos de *locks*, sendo que a maioria

dos comandos neste SGBD automaticamente atribui os *locks* apropriados de modo a que não se faça *drop table* ou se modifique alguma tabela de forma inconsistente enquanto o comando é executado.

Locks de tabelas - Existem vários modos de *locks* e diferenciam-se entre si pelo conjunto de *locks* com que entram em conflito, sendo eles os seguintes:

ACCESS SHARE: Entra em conflito com ACCESS EXCLUSIVE e é adquirido usando o comando SELECT.

ROW SHARE: Entra em conflito com EXCLUSIVE e ACCESS EXCLUSIVE e é adquirido usando os comandos SELECT FOR UPDATE e SELECT FOR SHARE.

ROW EXCLUSIVE: Entra em conflito com SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE e é adquirido usando os comandos UPDATE, DELETE e INSERT.

SHARE UPDATE EXCLUSIVE: Entra em conflito com SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE e é adquirido usando os comandos VACUUM, ANALYZE, CREATE INDEX CONCURRENTLY.

SHARE: Entra em conflito com ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE e é adquirido usando o comando CREATE INDEX.

SHARE ROW EXCLUSIVE: Entra em conflito com ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE e este modo não é adquirido automaticamente por nenhum comando.

EXCLUSIVE: Entra em conflito com `ROW SHARE`, `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`,`SHARE ROW EXCLUSIVE`, `EXCLUSIVE` e `ACCESS EXCLUSIVE` e este modo não é adquirido automaticamente por nenhum comando.

ACESS EXCLUSIVE: Entra em conflito com todos os modos(`ACCESS SHARE`, `ROW SHARE`, `ROW EXCLUSIVE`,`SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` e `ACCESS EXCLUSIVE`)e é adquirido usando os comando ALTER TABLE, DROP TABLE, TRUNTATE, REINDEX, CLUSTER e VACUUM FULL.

Locks de tuplos - Relativamente aos tuplos das tabelas, os *locks* podem ser do tipo *exclusive* ou *shared*, sendo que o exclusive é automaticamente atribuído na actualização ou eliminação de um tuplo. O *lock* é mantido até a transacção fizer um *commit* ou *rollback*.

Deadlocks - O uso de *locks* explícitos aumenta a probabilidade da existência de *deadlocks*. Estes acontecem tipicamente quando duas transacções distintas tentam obter um *lock* de um tuplo ou tabela em ordem oposta. Por exemplo a transacção 1 não consegue actualizar a tabela B pois o *lock* pertence à transacção 2 que por sua vez não consegue actualizar a tabela A pois o *lock* desta pertece à transacção A. Para evitar um caso de *deadlock*, é chamado um algoritmo de detecção que cria um grafo de esperas baseado na informação de tabela de *locks* e procura por ciclos. Se encontrar algum faz *roll back* de uma transacção (que contribuiu para o *deadlock*) que fez *timeout*, tipicamente a que realizou menos trabalho.

Consistência - Em termos de consistência da base de dados, o nível de isolamento mais aconselhado é o *Serializable* pois as transacções que sofram violação de integridade são prontamente abortadas. A consistência da base de dados durante uma transacção pode ser verificada usando o seguinte comando:

SET CONSTRAINS { ALL | name [, …] }

{ DEFERRED | IMMEDIATE }

Sendo que *deferred* verifica a integridade no fim da transacção e *immediate* verifica entre operações.

6 Suporte para bases de dados distribuídas

A ideia de replicação de dados e consequente acesso a estes a partir de mais do que um servidor está presente no PostgreSQL (ainda que só recentemente). Assegurar a disponibilidade dos dados devido à existência de diversos servidores que se apoiam mutuamente no caso de falha é uma vantagem fulcral num sistema de gestão de bases de dados.

No entanto, não há implementações que por si só resolvam todos os problemas inerentes a estes casos (como por exemplo, a difícil tarefa de sincronizar dados entre vários servidores) pelo que haverá umas mais vantajosas dependendo da ocasião.

No que toca ao modo como essas mesmas implementações tentam lidar com o problema da sincronização, existem certos conceitos presentes, como o da existência de servidores síncronos, que não consideram válido o *commit* de uma transacção até todos os servidores o terem feito, assegurando que, no caso de existir uma falha, o processo de *failover* (onde um sistema secundário assume as funções de um sistema primário que falha ou está inacessível) não perde dados e que as *queries* continuam a retornar resultados consistentes.

Existem também implementações assíncronas, que estabelecem um período de atraso entre um *commit* e a propagação para outros servidores. Isto, embora resulte numa execução mais rápida em comparação com uma solução síncrona, poderá, em caso de falha, traduzir-se em perda de dados e na devolução de resultados não actualizados por parte de algumas *queries*.

Em seguida, serão apresentadas algumas das soluções.

6.1 *Failover* com partilha de disco

É guardada apenas uma cópia de toda a base de dados, sendo esta partilhada por diversos servidores. No caso de ocorrer uma falha por parte do servidor principal, um servidor secundário assume as suas funções. Dada a rápida intervenção, não deverá haver perda de dados. Se o servidor secundário responsável não puder ser iniciado, então é escolhido um novo servidor. Caso haja uma nova falha durante o período em que o servidor secundário funciona, então espera-se um certo tempo até que ele seja reiniciado, de modo a iniciar um processo de recuperação.

O PostgreSQL não sabe detectar falhas no servidor primário nem avisar o segundo de que terá de assumir as funções, pelo que é necessário recorrer a ferramentas externas ao sistema.

6.2 Log Shipping

Devido à actualização contínua de dados que é transmitida aos servidores secundários através dos WAL (Write-Ahead Logs), estes são capazes de se tornar servidores principais, em caso de falha, por terem quase toda a informação contida neles antes de ocorrer um problema. Tratando-se de um processo assíncrono, é possível perderem-se dados no caso do servidor principal falhar e não enviar as mudanças recentes para os servidores secundários.

Quando são enviadas *queries* de leitura para um servidor primário, este pode aproveitar-se do facto de haver um servidor secundário que contém dados idênticos aos seus, para efeitos de *backup* e recuperação, e pedir-lhe que efectue ele mesmo essas *queries*. A este processo dá-se o nome de *hot standby*.

Este processo poderá ser efectuado com base em *streaming*, que diminui o tempo de atraso entre um servidor principal e secundário, resultando numa menor probabilidade de haver perda de dados no caso de ocorrer um problema.

Também pode ser realizado com base em *cascading*, que resulta numa redução de peso para o servidor principal aquando do envio de alterações, dado que cada uma pode ser transmitida entre servidores secundários.

6.3 Replicação assíncrona com servidores *master*.

É relativamente complicado manter os dados consistentes entre servidores que nem sempre se encontram ligados, pelo que, através desta técnica, se assume que cada servidor funciona independentemente dos outros e que apenas se contactam aquando da existência de transacções conflituosas. Estes conflitos são resolvidos a partir de regras preestabelecidas.

Este processo sacrifica por vezes a completude dos dados (perda de dados) em troca de uma rápida execução.

6.4 Replicação síncrona com servidores *master*.

Com este tipo de replicação, existe já uma dependência entre os vários servidores. Em todas as modificações no servidor original, espera-se sempre que todos os outros se mantenham actualizados imediatamente antes de efectuarem *commit*. Como a escrita dos dados se traduz então num complexo sistema de *locking*, este processo poderá ser bastante lento, apesar de haver a garantia de que todos os servidores têm os dados completos e não houve perda destes pelo meio. Assim, a replicação assíncrona é indicada para operações de leitura.

7 Outras características

7.1 Tipo XML

O PostgreSQL suporta um tipo de dados específico para guardar dados XML. Esse tipo, *xml*, poderá então armazenar documentos que estejam *bem-formados*, isto é, sejam validados através de um conjunto de regras impostas pela especificação XML.

A criação de um valor XML (input) é dada pela função *xmlparse*, com a seguinte sintaxe:
XMLPARSE ({ DOCUMENT | CONTENT } value)

De modo a transformar um valor XML numa sequência de caracteres (output), usa-se a função *xmlserialize*, com a seguinte sintaxe (sendo *type* um caracter ou texto por exemplo):
XMLSERIALIZE ({ DOCUMENT | CONTENT } value AS type)

É necessário ter algum cuidado com o modo como os dados são geridos e armazenados, principalmente devido à diversidade de codificação de caracteres que possa existir num documento. O processamento destes dados seria ideal e imune a problemas se o modo como cada entidade que os gere usasse o mesmo tipo de codificação. Por isso, de modo a generalizar, é sugerida a codificação UTF-8.

É também importante referir que não existe forma de criar índices com base num atributo que seja deste tipo, tornando a procura mais lenta do que se possa desejar. Uma possível solução é realizar um *cast* à volta destes atributos de modo a que sejam temporariamente vistos como *strings* por exemplo, e criar os índices normalmente de modo a acelerar uma pesquisa.

7.2 Full-text search

Esta funcionalidade procura estabelecer ligações entre o que é pedido numa *query* e documentos que sejam representados em língua natural, devolvendo um conjunto destes que contenham termos que mais se assemelhem aos da *query*.

Para acelerar pesquisas, é feito um pré-processamento aos diversos documentos que os torna mais úteis na fase de indexação.

Esta fase de processamento analisa (efectua o *parsing*) *o* conteúdo de um documento e torna-o num conjunto de *tokens*, de modo que os tipos de dados presentes do documento possam ser mais facilmente processados; de seguida, converte esses *tokens* para *lexemes*, que são *strings* normalizadas, isto é, aquando de uma comparação, é possível determinar se as palavras representadas pelos *lexemes* são variações de uma mesma palavra, e serem associadas unicamente a essa mesma raíz. São também excluídas palavras que por serem bastante comuns na língua em questão, não trazem qualquer benefício para a pesquisa; ocorre depois o armazenamento de documentos que foram preprocessados e que estão optimizados para pesquisa.

Em suma, o documento é fragmentado, sendo depois esses fragmentos analisados lexicalmente de modo a poderem ser mais eficientemente usados pela indexação. Dois tipos de índice usados são os GiST e GIN.

7.2.1 Detecção de padrões:

Um documento é então um conjunto de valores de texto, que pode ser o resultado da união de vários atributos de uma ou mais tabelas, sobre o qual se pretendam obter tuplos que se associem a esse mesmo documento. Cada documento tem ainda de estar no formato de pré-processamento *tsvector*.

Dessa forma, cada documento passa a ser tratado como um valor *tsvector*, e o *matching* de texto e da *query* é feito usando o operador '@@'. O resultado de uma operação TSVECTOR @@ TSQUERY devolve *true* caso o documento contenha termos da query

Por exemplo:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat'::tsquery;
 ?column?
```

devolverá true, enquanto:

```
SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector;
 ?column?
```

devolverá false.

8 Comparação com o Oracle

O principal objectivo deste trabalho foi efectuar um estudo sobre um sistema de gestão de bases de dados e efectuar uma comparação com o sistema estudado nas aulas práticas, o Oracle. Decidimos estudar o PostgreSQL visto que é o SGBD mais parecido com o Oracle.

A nível de indexação e ao contrário do Oracle o PostgreSQL possui índices GiST, SP-GiST e GIN. Os dois SGBDs usam índices B+Tree. Apesar do Oracle não usar índices Hash, este suporta Hash estático. No entanto apesar do PostgreSQL suportar índices Hash, não é aconselhável a sua utilização.

Relativamente ao armazenamento e estrutura de ficheiros o PostgreSQL tem como vantagens o sistema de buffer implementado, permitindo minimizar os custos de acesso à memória de forma eficiente. Como este SGBD usa o sistema de ficheiros do sistema operativo, isso simplifica o acesso e a gestão dos ficheiros, não sendo necessário o trabalho adicional de criar mecanismos que lidem com os ficheiros, como é o caso do Oracle. Assim sendo, o Oracle embora tenha a desvantagem de criar estas mesmas ferramentas que permitem a gestão de ficheiros, tem a vantagem de ter o seu próprio sistema de ficheiros, que permite ter mais controlo sobre os dados inseridos na base de dados.

Relativamente à gestão de transacções, o PostgreSQL e o Oracle têm um funcionamento semelhante, dado que ambos usam o mecanismo de snapshot isolation, que traz benefícios a nível da solidez da base de dados dando consistência às transacções concorrentes, mas tem a desvantagem de ocupar muito espaço em disco. Ambos têm também quatro níveis de isolamento (embora para efeitos práticos só três são usados internamente), assim como ambos podem obter locks explicitamente.

No que toca ao suporte de bases distribuídas, o Oracle está mais bem preparado. Não há surpresas relativamente a isso dado que o suporte à distribuição de servidores é recente no PostgreSQL.

9 Bibliografia

1. ***Wikipedia.*** http://en.wikipedia.org/wiki/PostgreSQL,2014.

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

3. ***PostgreSQL 9.2 Documentation***. http://www.postgresql.org/docs/9.2/static/,2014.

4. ***PostgreSQL 9.3 Documentation.*** http://www.postgresql.org/docs/9.3/static/index.html,2014.