PostgreSQL 9.3

Sistemas de bases de dados, Grupo 14
Universidade Nova de Lisboa - FCT

1 de junho de 2014

Professor: José Alferes

Bruno Grácio, 41981
Fábio Valente, 42177
Pedro Pinto, 41974

***Resumo**: No âmbito da cadeira de sistemas de base de dados foi desenvolvido um trabalho sobre o sistema PostgreSQL, estudando algumas das suas principais características. Este trabalho foi dividido em oito secções: Introdução Histórica, Armazenamento e File-structure, Indexação e Hashing, Processamento e Otimização de queries, Gestão de transações e controlo de concorrência, Suporte para bases de dados distribuídas e Outras Características deste sistema e Comparações com o Oracle 11g.*

Índice

1. Introdução histórica

O *PostgreSQL* (conhecido anteriormente como *Postgres95*) derivou do projeto *POSTGRES* da universidade de *Berkley*. A implementação do projeto *POSTGRES* iniciou-se em 1986 e em 1987 tornou-se operacional, a primeira versão lançada para o público externo foi em 1989. Em 1991 foi lançada uma nova versão, com melhorias no executor de consultas e algumas partes do código foram reescritas. As versões subsequentes, até ao *Postgres95*, foram focadas em confiabilidade e portabilidade.

O *POSTGRES* foi utilizado para diversos sistemas de pesquisa e de produção, uma aplicação de análise financeira, um banco com rotas de asteroides, e diversos sistemas de informações geográficas. O código do *POSTGRES* foi aproveitado para um produto comercializado pela *Illustra Information Technologies* (posteriormente incorporada à *Informix*, que agora pertence à *IBM*).

A versão seguinte, o *Postgres95*, teve mudanças radicais em relação ao projeto original. O seu código foi totalmente revisto e o tamanho das fontes foi reduzido em 25%, a linguagem *SQL* foi implementada como interface padrão. A performance foi consideravelmente melhorada e vários recursos foram adicionados. Em 1996 o nome *Postgres95* tornou-se inadequado, o projeto foi rebatizado "*PostgreSQL*", para enfatizar a relação do *POSTGRES* original com a linguagem *SQL*, a primeira versão oficial do *PostgreSQL* foi a 6.0.

No *Postgres95* o foco era sobretudo na correção de falhas e otimização de código, enquanto no *PostgreSQL* o foco foi sobretudo na melhoria de recursos existentes e implementação de novos recursos, sempre seguindo os padrões de *SQL* anteriormente estabelecidos.

2. Armazenamento e *file-structure*

2.1 *Buffer*

O *PostgreSQL* implementa o seu próprio gestor de *buffers*. Uma desvantagem desta implementação é a possibilidade de existir conflito com o sistema de gestão de *buffers* do Sistema operativo. Este sistema usa dois tipos de *buffers*:

- O ***shared_buffers***, define a quantidade de memória que a base de dados usa para os *buffers* de memória partilhada. Este sistema usa como valor padrão 32MB, mas o valor pode ser menor;
- O ***temp_buffers***, define a quantidade Máxima de *buffers* temporários utilizados em cada sessão da base de dados. Estes *buffers* são usados apenas para o acesso a tabelas temporárias. Este sistema usa como valor padrão 8MB, mas pode ser alterado.

2.2 *File System*

O *PostgreSQL* usa o sistema de ficheiros do Sistema Operativo. Tradicionalmente, os arquivos de configuração usados pelas bases de dados são armazenados no *PGDATA*. Uma local comum para o *PGDATA* é */var*, */lib*, */pgsql*, */data* e contém várias subdiretorias, como podemos ver na tabela 1.

Podem existir diversos *clusters* numa só máquina, geridos por diferentes instâncias do servidor, sendo que para cada uma das bases de dados no *cluster* existe uma subdiretoria contida em *PGDATA/base* com o nome do *Object ID*. É possível consultar a lista de identificadores no ficheiro *pg_database*.

Quando a tabela ou índices são superiores a 1GB esta é dividida em segmentos. Estes segmentos são guardados com o nome *ficheiro.1, ficheiro.2, ..., ficheiro.N*. Este método evita problemas em plataformas que têm limitações de tamanho nos ficheiros.

Item	Descrição
PG_VERSION	Um arquivo que contém o número da versão principal do PostgreSQL
base	Subdiretório contendo subdiretórios por banco de dados
global	Subdiretório que contém tabelas de todo o agrupamento, como pg_database
pg_clog	Subdiretório contendo transação comprometer dados de status
pg_multixact	Subdiretório que contém os dados de status multitransaction (usado para bloqueios de linhas compartilhadas)
pg_notify	Subdiretório contendo escutar / notificar dados de status
pg_serial	Subdiretório que contém informações sobre transações serializáveis cometidos
pg_snapshots	Subdiretório contendo instantâneos exportados
pg_stat_tmp	Subdiretório que contém os arquivos temporários para o subsistema de estatísticas
pg_subtrans	Subdiretório que contém os dados de status subtransação
pg_tblspc	Subdiretório que contém links simbólicos para os espaços de tabela
pg_twophase	Subdiretório que contém os arquivos de estado para transações preparadas
pg_xlog	Subdiretório contendo WAL (Write Ahead Log) arquivos
postmaster.opts	Um arquivo de gravar as opções de linha de comando, o servidor foi iniciado pela última vez com
postmaster.pid	Um arquivo de bloqueio gravação do ID do processo postmaster (PID), caminho do diretório de dados do agrupamento, postmaster começar timestamp, número de porta, Unix-domain caminho do diretório socket (vazia no Windows), listen_address primeiro válido (endereço IP ou * , ou vazio se não escuta em TCP), e ID segmento de memória compartilhada (este arquivo não estiver presente após o desligamento do servidor)

Tabela 1: Conteúdo do PGDATA

2.3 Partições

O *PostgreSQL* suporta o particionamento de tabelas básicas através da herança de tabelas. Para configurar uma tabela de partição devemos:

1. Criar a tabela principal das quais as partições vão herdar;
2. Criar as tabelas filho que vão herdar da tabela principal. Podem ser tabelas normais, mas vamos consulta-las como partições;
3. Adicionar restrições para as tabelas filho, para definirmos os valores chave permitidos para cada partição.

```
CHECK (x = 1)
CHECK (condado IN ('Oxfordshire', 'Buckinghamshire', 'Warwickshire'))
CHECK (outletID> = 100 E outletID <200)
```

Devemos ter alguma atenção a definir estas restrições para não gerar sobreposições entre valores, como o exemplo abaixo.

```
CHECK (outletID entre 100 e 200)
CHECK (outletID entre 200 e 300)
```

4. Para cada partição criar um índice;
5. Opcionalmente, podemos definir um *trigger* para redirecionar os dados da tabela principal para as partições apropriadas.

2.4 Organização

No *PostgreSQL* é usado umo formato de *slotted-page* de tamanho fixo (normalmente 8KB). Estas páginas têm a seguinte estrutura:

Item	Descrição
PageHeaderData	24 bytes de comprimento. Contém informações gerais sobre a página, incluindo ponteiros espaço livre.
ItemIdData	Array of (offset, comprimento) pares apontando para os itens reais. 4 bytes por item.
Espaço livre	O espaço não alocado. Novos ponteiros de itens são alocados a partir do início desta área, novos itens a partir do final.
Itens	Os itens reais próprios.
Espaço especial	Dados específicos do método de acesso do índice. Diferentes métodos de armazenamento de dados diferentes. Esvaziar em tabelas normais.

Tabela 2: Estrutura geral das páginas

Como o tamanho é fixo, é difícil armazenar grandes quantidades de tuplos. Para resolver esta questão o *PostgreSQL* recorre a uma implementação *TOAST*. *The Oversize Attribute Storage Techique*, *TOAST*, recebe essas grandes quantidades de tuplos e comprime ou divide-os em múltiplas linhas físicas de forma transparente para o utilizador.

Existem quatro estratégias diferentes para armazenar entradas *TOAST*:

- ***PLAIN***: impede a compressão ou o armazenamento fora de linha. É a única opção para colunas com tipos de dados que não necessitam de *TOAST*;
- ***EXTENDED***: permite a compressão e armazenamento fora-de-linha. Este é o padrão para a maioria das técnicas *TOAST*.
- ***EXTERNAL***: permite o armazenamento fora-de-linha, mas não de compressão. Uso desta estratégia otimiza a manipulação dos dados com o custo de utilizar mais espaço livre.
- ***MAIN***: permite o armazenamento de compressão, mas não fora-de-linha.

3. Indexação e hashing

3.1 Introdução

O índice de uma base de dados é uma estrutura de dados que ajuda a melhorar o tempo de acesso à informação. Nos próximos exemplos, vamos considerar que temos uma tabela `Pessoa(id,nome)` e que é feita a seguinte pergunta: `SELECT nome FROM pessoa WHERE id = X`.

Supondo que não houve nenhum tipo de preparação inicial, o sistema para responder à pergunta efetuada teria que, percorrer toda a tabela até conseguir encontrar alguma linha que correspondesse à condição `id = X`. Este método é claramente ineficiente, no entanto, se tivéssemos indicado ao sistema para manter um índice sobre o atributo, esta operação teria sido efetuada muito mais eficientemente.

Ao correr o seguinte comando vamos criar um índice sobre o atributo id:

```
CREATE INDEX índice_id_pessoa ON pessoa(id)
```

Assumindo agora que foi criado um índice sobre o atributo id, o sistema irá automaticamente atualizar o índice sempre que alterámos a tabela e utilizá-lo sempre que se pretender realizar uma pergunta à tabela Pessoa. Este método é agora mais eficiente que uma pesquisa sequencial às linhas da tabela.

Criar um índice de uma tabela grande pode ser demorado, no entanto, o *PostgreSQL* permite efetuar perguntas à tabela em paralelo com a criação do índice. Quando efetuamos inserções, atualizações ou remoções, estas operações são bloqueadas até que o índice esteja totalmente criado.

É possível permitir que as inserções ocorram em simultâneo com a criação do índice, mas temos que ter atenção aos problemas que tal permissão possa gerar. Este método pode ser invocado através da opção `CONCURRENTLY` do comando `CREATE INDEX`.

Quando ativamos esta opção, vamos consumir recursos extra do *CPU* e vamos impor um maior número de operações *I/O*, o que pode atrasar as operações.

3.2 Estruturas de dados suportadas

O *PostgreSQL* disponibiliza vários tipos de índice: ***B-Tree***, ***Hash***, ***GIST***, ***SP-GIST*** e ***GIN***. Se não indicarmos o tipo de índice que queremos quando efetuamos o comando `CREATE INDEX`, o indicie *B-Tree* é escolhido porque é o que se melhor adapta à maioria das situações.

O índice ***B-Tree***, vai ser considerado sempre que haja uma comparação utilizando: `<`, `<=`, `=`, `>=` e o `>`. Este índice também suporta condições como: `BETWEEN`, `IN`, `IS NULL` e `IS NOT NULL`. Uma das utilizações muito comuns para este índice é para obtenção da informação por uma certa ordem.

O índice ***Hash***, só consegue lidar com comparações de igualdade simples. Este índice vai ser considerado sempre que o operador `=` estiver envolvido na comparação. Para criar índices deste tipo utilizamos o comando `CREATE INDEX nome_indice ON nome_tabela USING hash (coluna)`. Devemos ter atenção que estes índices precisam de ser reconstruídos com o comando `REINDEX` depois de uma falha na base dados se existir informação por escrever. O *hash* é dinâmico, logo não possui função de *rehashing*.

O índice ***GIST***, não é apenas um índice, mas sim uma infraestrutura com muitas estratégias diferentes de indexação. O *PostgreSQL* inclui classes de operadores *GIST* para alguns tipos geométricos 2D. É capaz também de otimizar a pesquisa *nearest-neighbor* para encontrar os lugares mais próximos de um ponto principal.

O índice ***SP-GIST***, tal como o índice *GIST*, oferece uma infraestrutura que suporta vários tipos de pesquisa. Este índice permite a implementação de uma vasta gama de diferentes estruturas de dados não balanceadas, tais como: *quadtrees*, árvores *kd* e árvores *radix*.

O índice ***GIN***, pode lidar com valores que contenham mais do que uma chave, como por exemplo: matrizes. Tal como as anteriores, o *GIN* suporta diferentes estratégias de indexação. Este índice, tal como *GIST*, são usados quando efetuamos pesquisas por texto.

3.3 Índice multi-coluna

Um índice pode ser definido como multi-coluna se contiver mais do que uma coluna de uma tabela. Por exemplo ter uma tabela do género `Pessoa(id,nif,nome)` e ser realizado uma pergunta `SELECT nome FROM pessoa WHRE id=X AND nif=Y`.

Ao correr o seguinte comando vamos criar um índice sobre os atributos id e nif:

```
CREATE INDEX índice_id_nif_pessoa ON pessoa(id,nif)
```

Conhecemos três tipos de índices capazes de lidar com esta situação, *B-Tree, GIST e GIN*. O *PostgreSQL* tem um limite de 32 colunas, mas este limite pode ser alterado manualmente. O índice multi-coluna *B-Tree*, pode ser usado numa pergunta que envolva qualquer subgrupo das colunas indexada, mas é mais eficiente quando existem restrições nas primeiras colunas indexadas. Devemos ter atenção, porque cada coluna deve ser usada com os operadores adequados para o tipo de índice.

Este tipo de índices deve ser usado com moderação. Na maioria das vezes um índice de uma só coluna é suficiente, o que economiza tempo e espaço. Índices com mais de três colunas, são mais suscetíveis de não serem usadas, a menos que a tabela esteja extremamente alterada.

3.4 Índices e ordenação

Um índice é capaz de devolver um conjunto de linhas de uma tabela por ordem. Isto é bastante bom, porque assim a consulta `ORDER BY`, não precisa de um paço extra para a ordenação. Dos índices suportados pelo *PostgreSQL*, só a *B-Tree* produz saídas de valores ordenadas.

Se uma consulta pedir uma leitura de uma larga fração de uma tabela, a ordenação explícita será provavelmente a técnica mais eficiente usando o índice. Isto porque requer menos acessos a disco durante os padrões de acesso sequencial. Os índices são mais úteis quando as consultas são sobre uma fração menor de uma tabela.

Um caso especial do *PostgreSQL* é usar o `ORDER BY` em conjunto com o `LIMIT N`. Isto permite uma ordenação explícita processe apenas as primeiras `N` linhas, no entanto, se existir um índice correspondente ao `ORDER BY`, as primeiras `N` linhas podem ser obtidas diretamente. Podemos a qualquer momento ajustar a ordem de um índice *B-Tree*, incluindo as opções: `ASC`, `DESC`, `NULLS FIRST` e `NULLS LAST`.

3.5 Multi-índices

Um índice único consegue apenas usar as cláusulas que usem colunas do índice que estão unidos pela operação `AND`. O *PostgreSQL* tem a capacidade de combinar vários índices para lidar com estes casos. O sistema consegue realizar condições de `AND` e `OR` em várias pesquisas no índice. Por exemplo uma pergunta do género `WHERE x = 42 OR x = 47 OR x = 53 OR x = 99` pode ser separada em quatro pesquisas.

Para combinar índices múltiplos, o sistema verifica cada índice necessário e prepara o bitmap em memória dando a localização das linhas da tabela. É aplicado depois no bitmap as funções `AND` e `OR` juntas como necessário para a pergunta. Finalmente as linhas atuais da tabela são visitadas e retornadas.

Existem várias combinações de índices que podem ser úteis. No entanto o programador da base de dados deve realizar *trade-offs* para decidir quais índices fornecer. Às vezes, índices de várias colunas são os melhores, mas á vezes é melhor criar índices separados.

3.6 Índices para organização de ficheiros

Quando uma tabela é *cluster*, é fisicamente reordenada com base na informação de um índice. No entanto, este comando só executa uma vez. Quando a tabela é atualizada, as alterações não são agrupadas, isto é, não é feita nenhuma tentativa para armazenar as

linhas novas na ordem em que está o índice. Logo, se pretendermos reordenar a tabela, temos que voltar a correr o comando. A criação da tabela com o parâmetro `FILLFACTOR`, pode ajudar a preservar a ordem do *cluster* durante as atualizações.

```
CLUSTER [VERBOSE] nome_tabela [USING nome_indice]
```

3.7 Estruturas temporariamente inconsistentes

O acesso de estruturas de índice, no *PostgreSQL* tem que ser efetuado controladamente através de um *LOCK*. Apesar de o *PostgreSQL* controlar o acesso aos índices, as atualizações podem criar inconsistências entre uma tabela e o índice. Isto deve-se ao facto do sistema ter uma separação lógica sobre acessos a tabelas e acessos a índices.
Uma vez que o *PostgreSQL* permite diferir a verificação de restrições de integridade para o final das transações, é também necessário permitir a existência de estruturas inconsistentes no sistema temporariamente.

4. Processamento e otimização de perguntas

4.1 Introdução

O objetivo do processamento e otimização de perguntas é criar um plano de execução ótimo para uma dada pergunta *SQL*. Ou seja, dada uma pergunta *SQL*, esta pode ser executada por uma variedade de planos de execução em que cada um desses planos produz o mesmo conjunto de resultados. Caso a pergunta seja computacionalmente viável, são examinados todos os planos de execução onde é selecionado o plano que executa mais rapidamente a pergunta. Mas em algumas situações não é possível verificar todos os planos de execução devido ao gasto de tempo em processamento a verificar qual seria o melhor plano a ser executado, isto acontece geralmente quando as perguntas têm muitas operações de *joins*. Então para uma pergunta *SQL* ser executada o mais rapidamente possível pelo sistema de gestão de base de dados tem que seguir um conjunto de transformações:

1. A aplicação envia uma pergunta ao *SGBD* e espera pela resposta;
2. O *parser* verifica a sintaxe da pergunta e constrói uma *query tree* da pergunta;
3. O sistema de reescrita procura na árvore e substitui pelo seu valor real;
4. O optimizador analisa a *query tree* e cria todos os caminhos possíveis do resultado final, analisa o custo estimado de cada caminho e escolhe o de menor;
5. O executor executa o plano escolhido e envia a resposta a quem fez o pedido.

Parser

O *parser* tem duas fases de execução. Primeiro o *parser* recebe uma pergunta e verifica se cada palavra da pergunta tem a sintaxe do SQL, após a verificação da sintaxe de toda a pergunta é criado um *parser tree*. Este *parser tree* consiste em representar a pergunta *SQL* numa linguagem mais formal e sem ambiguidades. Na segunda fase de execução, o *parser* analisa o *parser tree* criado anteriormente, verificando-o semanticamente e criando uma *query tree*. Uma *query tree* é uma representação interna de uma pergunta *SQL*.

Sistema de reescrita

O sistema de reescrita de uma pergunta é uma fase que está entre a análise da pergunta (*parser*) e o *planner/optimizer*. Processa a *query tree* devolvida pelo *parser* (que representa a pergunta feita pelo utilizador) e se existir alguma regra que foi pré-definida pelo utilizador, esta tem de ser aplicada e uma nova *query tree* é escrita de forma alternativa.

Planner/Optimizer

A tarefa do *planner/optimizer* é criar um plano de execução ótimo. Dada uma pergunta *SQL*, e consequentemente a sua *query tree* criada pelo *parser*, esta pode ser executada de diferentes formas. Mas cada uma delas produz o mesmo conjunto de resultados. O optimizador normal do *PostgreSQL* executa uma pesquisa quase exaustiva sobre o espaço de estratégias alternativas. Este algoritmo, introduzido pela primeira vez na base de dados *System R* da *IBM*, produz uma ordem quase ótima das junções, mas pode tomar uma enorme quantidade de tempo e de espaço em memória quando o número de junções de tabelas de uma pergunta é grande. O que faz com que o optimizador normal de perguntas do *PostgreSQL* seja inadequado para perguntas que unem uma grande quantidade de tabelas. Então de modo a escolher um plano de execução razoável (que pode não ser o plano ótimo) para ser executado, o *PostgreSQL* utiliza o *Genetic Query Optimizer* quando o número de junções excede um determinado *threshold*.

Genetic Query Optimizer

De todos os operadores relacionais o mais difícil de processar e otimizar é a junção, o número de possíveis planos de consulta cresce exponencialmente com o número de junções da pergunta. Além disso, o esforço que o optimizador tem para otimizar uma pergunta é ainda maior devido à variedade de métodos de junção (*nested loop join, hash join e merge join*) e à diversidade de índices (*B-tree, hash, GIST e GIN*) como caminhos de acesso para as relações.

Para tentar resolver este problema o *PostgreSQL* utiliza um outro optimizador chamado o *Genetic Query Optimizer (GEQO)*. O *GEQO* aborda o problema de otimização como se fosse o problema do caixeiro-viajante (*TSP - Traveling Salesman Problem*). Os planos de

execução possíveis são codificados como *strings* de inteiros e cada sequência representa a ordem de junção de uma relação para a próxima.

O processo de planeamento do *GEQO* utiliza um *standard planner* para gerar planos para fazer leituras individuais sobre relações. De seguida, os planos de junção são criados utilizando uma abordagem genética (utilizando um algoritmo genético). Na fase inicial, o *GEQO* gera algumas possíveis sequências de junção ao acaso e para cada sequência de junção gerada, o *standard planner* é invocado para estimar o custo de execução da pergunta utilizando aquela sequência de junção. As sequências de junção consideradas mais aptas são aquelas que têm um custo de execução menor comparando com as sequências de junção que têm um custo de execução maior e dessa comparação o algoritmo genético descarta os candidatos menos aptos. De seguida, são geradas novas sequências de junção através da combinação de genes dos candidatos mais aptos, ou seja, utilizando porções aleatoriamente escolhidas das sequências de junção com menor custo selecionadas para criar novas sequências a considerar. Este processo é repetido até um número pré-estabelecido de sequências de junção forem consideradas. E no final, o candidato mais apto (sequência de junção com menor custo) é escolhido e utilizado para gerar o plano de execução a ser utilizado.

Executor

O executor recebe o plano de execução criado pelo *planner/optimizer* e recursivamente processa-o para extrair um conjunto necessário de tuplos. Este é um essencialmente um mecanismo de *pipeline demand-pull*, em que cada vez que um nó do plano é chamado, o executor entrega mais tuplos ao nó pai ou indica que já não tem mais tuplos para entregar.

4.2 Operação básica (Seleção)

No *PostgreSQL* existem três tipos de implementação do operador de seleção e são eles, *sequencial scan, index scan* e *bitmap index scan*:

- ***Sequencial Scan***: É o algoritmo de seleção que funciona para todas as pesquisas e consiste em percorrer todos os tuplos de uma tabela, verificando se cada um respeita as condições de seleção descritas na clausula `WHERE` respetivas a tabela que está a ser percorrida;
- ***Index Scan***: Este algoritmo permite uma pesquisa por índices, tanto por índices primários como secundários. É possível que o index scan não percorra todos os tuplos de uma tabela, caso as pesquisas tenham operações de menor, maior, menor ou igual e maior ou igual. Consequência deste tipo de scan leva também a devolver os tuplos pela ordem dos índices em vez de ser pela ordem que estão guardados na tabela;
- ***Bitmap Index Scan***: Este algoritmo utiliza uma estrutura de dados guardada em memória chamada *bitmap* que guarda apenas um conjunto de *bits* que representam o valor de um determinado atributo de uma tabela, significa que um bitmap tem tantos bits quantos registos existam na tabela. Os índices *bitmap* são particularmente interessantes para perguntas que contam o número de tuplos (`SELECT COUNT(*)`) porque não é necessário aceder aos dados em disco. Mas para muitos atributos os *bitmaps* podem não ser uma boa solução visto que é preciso um *bitmap* para cada valor do atributo da tabela em questão.

4.3 Operação básica (Junção)

No *PostgreSQL* existem três tipos de implementação do operador de junção e são eles, *nested loop join, merge join e hash join*:

- ***Nested Loop Join***: Consiste em fazer um scan da relação do lado direito para todos os tuplos da tabela do lado esquerdo. Este tipo de implementação é muito fácil de implementar mas pode ser muito dispendioso executá-la. Contudo, caso a tabela do lado direito possa ser lida através de um índice, esta é uma boa estratégia porque é possível usar os valores a partir do tuplo atual da relação da esquerda como chave para a verificação do índice da direita.

- ***Merge Join***: Esta implementação consiste na ordenação de cada relação sobre o atributo de junção antes de a operação *join* ser executada. De seguida as duas relações são percorridas em paralelo verificando as condições de junção. Este tipo de junção é bastante atrativo porque cada relação pode ser percorrida apenas uma vez, mas a sua complexidade é acrescida pois é feito primeiro uma ordenação.
- ***Hash Join***: Esta implementação consiste na leitura da relação do lado direito carregando-a para uma *hash table*, usando os seus atributos de junção como *hash keys*. De seguida, a relação do lado esquerdo é percorrida e os atributos de junção de cada tuplo são utilizados como *hash keys* para encontrar os tuplos correspondentes na *hash table*, caso os valores das *hash keys* tiverem na *hash table* é feita uma junção com todos esses tuplos.

4.4 Operação básica (Ordenação)

No *PostgreSQL* existem dois tipos de implementação para executar ordenação, são eles, o *quicksort* e o *external merge sort*. Os operadores de ordenação também são utilizados para remover duplicados no operador de união e no operador `DISTINCT`:

- ***Quicksort***: O *quicksort* só é executado quando as relações cabem na totalidade em memória;
- ***External merge sort***: O *external merge sort* é utilizado sempre que alguma das relações não cabe em memória. A sua execução é feita em duas fases, primeiro, as relações são partidas em blocos que caibam em memória, são ordenados e depois escritos em ficheiros temporários. E depois de todos os blocos estarem ordenados e escritos em ficheiros temporários, estes fazem um *merge* originando um único ficheiro completamente ordenado.

4.5 Mecanismos de suporte a expressões complexas

- **Materialização**: Como os algoritmos *Nested Loop Join e Merge Join* são algoritmos que normalmente percorrem várias vezes as tabelas sobre as quais estão a operar, tendem a executar várias vezes os mesmos resultados, por isso, os planos de execução que utilizam este tipo de algoritmos utilizam a

materialização para reduzir o tempo de computação. A materialização permite guardar resultados intermédios em memória possibilitando a sua reutilização em iterações seguintes, tornando o tempo de execução das perguntas mais rápido.

- ***Pipelining***: O modelo de *pipelining* implementado é o *demand-driven* que permite gerar tuplos à medida que são pedidos pelas operações superiores do *pipeline*. No entanto o *pipelining* não é possível para operações de ordenação ou de *hash join* porque é preciso ter conhecimento de todos os tuplos para executar os algoritmos referidos e no caso do *pipelining* os tuplos vão chegando á medida que são pedidos não tendo conhecimento de todos os tuplos numa só vez.
- **Paralelização**: O *PostgreSQL* suporta paralelismo completo do lado cliente. As aplicações podem abrir múltiplas conexões com a base de dados e gerem-nas de forma assíncrona ou por via de *threads*. Do lado servidor, existe algum paralelismo para conseguir receber os pedidos dos clientes que chegam paralelamente e também para executar perguntas, no caso de perguntas que executem o algoritmo *Hash Join*, este pode ser executado em vários processadores paralelamente para computar diferentes *hash keys*.

4.6 Verificação de planos de perguntas

O *PostgreSQL* concebe um plano de execução para cada pergunta que recebe. Escolher o plano certo para corresponder à estrutura da consulta e as propriedades dos dados é crucial para um bom desempenho, por isso, o sistema inclui um *planner* complexo que tenta escolher bons planos. Para ver o plano de execução que o *planner* criou para uma pergunta, utiliza-se o comando `EXPLAIN`. O output da pergunta com o comando `EXPLAIN` é uma linha para cada nó do plano de execução (nó da *query tree*), mostrando o tipo do nó mais as estimativas de custo que o *planner* fez para a execução desse nó do plano, assim como, qual o tipo de *scan* que é feito as tabelas referidas na pergunta e quais os algoritmos de junção que serão usados para juntar os tuplos de cada tabela. O comando `EXPLAIN` permite ainda que o seu resultado seja mostrado em vários formatos como, *XML, JSON* ou *YAML*.

Exemplo:

```
EXPLAIN SELECT * FROM city where city.name = 'Lisbon';
```

Output:

```
"Index Scan using citykey on city (cost=0.28..8.30 rows=1 width=40)"
"Index Cond: ((name)::text = 'Lisbon'::text)"
```

4.7 Estimativas

É possível verificar a precisão das estimativas do *planner*, utilizando o comando `EXPLAIN ANALYZE option`. Com esta opção o comando `EXPLAIN` executa a pergunta, e mostra as contagens reais das linhas e o tempo real acumulado dentro de cada nó do plano, juntamente com as mesmas estimativas que o comando `EXPLAIN` mostra.

Exemplo:

```
EXPLAIN ANALYZE SELECT * FROM city where city.name = 'Lisbon';
```

Output:

```
"Index Scan using citykey on city (cost=0.28..8.30 rows=1 width=40)
(actual time=0.050..0.052 rows=1 loops=1)"
"Index Cond: ((name)::text = 'Lisbon'::text)"
"Total runtime: 0.086 ms"
```

O *planner* do *PostgreSQL* depende de informações estatísticas sobre o conteúdo das tabelas por forma a gerar bons planos de execução das perguntas. Estas estatísticas podem ser atualizadas com o comando `ANALYZE`, que pode ser invocado por si só ou como um passo adicional no *VACUUM*. É importante ter estatísticas razoavelmente precisas, de outra forma a escolha de um plano de execução "fraco" pode prejudicar o desempenho da base de dados. Das estatísticas que são obtidas, as mais importantes são os histogramas que dão informação sobre o número de tuplos que um determinado valor numa tabela tem, e o mapeamento lógico-físico que é a relação entre a ordenação física e lógica dos tuplos das tabelas.

4.8 Parametrização

Estes parâmetros de configuração influenciam os planos de execução escolhidos pelo optimizador. Se o plano por *default* escolhido pelo optimizador para uma pergunta em particular, não é o ideal, uma solução temporária é usar alguns dos seguintes parâmetros de configuração para forçar o optimizador a escolher um pano diferente.

- **enable_bitmapscan (boolean)**: Ativa ou descativa o uso de bitmaps. Por *default* o parâmetro está *on*;
- **enable_hashjoin (boolean)**: Ativa ou descativa o uso de *hash joins*. Por *default* o parâmetro está *on*;
- **enable_indexscan (boolean)**: Ativa ou descativa o uso de *index scan*. Por *default* o parâmetro está *on*;
- **enable_material (boolean)**: Ativa ou descativa o uso de materialização. É impossível descativar a materialização inteiramente, mas tornando a variável do parâmetro *off* previne que o optimizador considere alguns planos. Por *default* o parâmetro está *on*;
- **enable_mergejoin (boolean)**: Ativa ou descativa o uso de *merge joins*. Por *default* o parâmetro está *on*;
- **enable_nestloop (boolean)**: Ativa ou descativa o uso de *nested loop joins*. É impossível suprimir inteiramente a opção de *nested loop join*, mas ao descativar este método desencoraja o optimizador a considerar alguns planos se tiver outros métodos disponíveis. Por *default* o parâmetro está *on*;
- **enable_seqscan (boolean)**: Ativa ou descativa o uso de *sequence scan*. É impossível suprimir inteiramente a opção de *sequence scan*, mas ao descativar este método desencoraja o optimizador a considerar alguns planos se tiver outros métodos disponíveis. Por *default* o parâmetro está *on*;
- **enable_sort (boolean)**: Ativa ou descativa o uso de ordenação. É impossível suprimir inteiramente o uso desta operação, mas descativando-a desencoraja o uso deste método levando o optimizador a usar outros métodos disponíveis;

Também existem parâmetros para configurar o *GEQO*, *Genetic Query Optimizer*. O *GEQO* é um algoritmo que faz planos de execução de perguntas utilizando uma heurística de pesquisa. Isto reduz o tempo de planeamento de perguntas complexas (perguntas que tenham muitas junções), com custo de elaboração de planos que são por vezes inferiores aos encontrados pelo algoritmo de pesquisa exaustiva normal.

- **geqo (boolean)**: Ativa ou descativa o algoritmo genético de otimização. Por *default* está ativo e é aconselhável não descativar este parâmetro em produção.
- **geqo_threshold (integer)**: Atribui um novo valor para o limite da pesquisa exaustiva normal, ou seja, caso o valor introduzido neste parâmetro for 8, o número de junções permitidas para fazer pesquisas de planos de execução correndo o algoritmo de pesquisa exaustiva é de 8. A partir de 8 junções o optimizador utiliza o algoritmo *GEQO* para procurar um plano de execução razoável/opimo para a pergunta. O valor por *default* é de 12.

5. Gestão de transações e controlo de concorrência

5.1 Introdução

O *PostgreSQL* interpreta cada comando SQL como uma transação, todos os comandos apresentam implicitamente as *keywords BEGIN* e *END* à sua volta. Este sistema não suporta transações do tipo *nested transactions* mas fornece um mecanismo baseado em *savepoints* que permite simular este comportamento, neste mecanismo é possível definir dentro de uma transação pontos para os quais se pode fazer um *ROLLBACK* para tratar de eventuais erros sem que seja necessário executar a transação desde o início.

Internamente, para garantir a consistência dos dados quando estes são acedidos em simultâneo, o *PostgreSQL* usa um modelo multi-versão de controlo de concorrência (*MVCC*). Este modelo protege cada transação e permite a várias transações, que estejam a aceder aos mesmos dados, uma visão de dados consistente. Estes dados podem tornar-se inconsistentes quando duas ou mais transações atualizam, os mesmos dados, simultaneamente.

O mecanismo usado para implementar o *MVCC* é o inovador *Snapshot Isolation Serializable (SSI)*. No *SSI* os *locks* adquiridos para fazer leitura de dados não entram em conflito com os *locks* adquiridos para fazer operações de escrita, assim uma leitura nunca bloqueia uma escrita e uma escrita nunca bloqueia uma leitura.

Apesar de o *PostgreSQL* manter esta garantia mesmo quando é fornecido o nível mais rigoroso de isolamento da transação, ele também implementa um mecanismo baseado em *locks*. Para aplicações que não necessitem da isolação total da transação o *lock* pode ser feito ao nível da tabela ou mesmo ao nível da linha, fazendo o *lock* aos pontos críticos da transação.

No entanto, o uso adequado de *Snapshot Isolation Serializable* geralmente produz um desempenho melhor que o sistema baseado em *locks*.

5.2 Isolamento de transações

O *standard* do *SQL* define quatro níveis de isolamento de uma transação. O nível mais restrito é o da serialização, qualquer execução simultânea de um conjunto de transações serializáveis produz o mesmo efeito, que executar uma de cada vez numa determinada ordem. Os outros três níveis de isolamento resultam da interação entre transações concorrentes. Fenómenos proibidos aos vários níveis:

- **Dirty Read**: A transação lê dados escritos por uma transação concorrente que ainda não fez *commit*;
- **Repeatable Read**: A transação volta a ler os dados anteriormente já lidos e descobre que os dados foram modificados por outra transação (que fez *commit* depois da leitura inicial);
- **Phantom Read**: A transação volta a executar uma consulta retornando o resultado que satisfaz a *query* de procura e descobre que o resultado que satisfaz a condição mudou devido ao *commit* de uma outra transação.

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

Tabela 3:Níveis de isolamento e fenômenos

5.3 Nível de isolamento (*Read Committed*)

Read Committed é o nível de isolamento *standard* do *PostgreSQL*. Quando uma transação utiliza este nível de isolamento, o comando `SELECT` (sem a cláusula `FOR UPDATE/SHARE`) consulta apenas os dados que já foram *commited* antes da consulta começar, nunca lê dados que não tenham sido *commited* ou dados intermédios que ainda estejam a ser processados por uma transação concorrente.

No entanto o comando `SELECT` vê as atualizações de resultados intermédios feitos pela sua própria transação, mesmo que estes ainda não tenham sido *commit*. Dois comandos `SELECT` feitos de forma sucessiva podem ver dados diferentes, mesmo estando dentro de

uma única transação, caso a transação faça *commit* das alterações realizadas aquando da execução do primeiro comando `SELECT`. Os comandos `UPDATE`, `DELETE`, `SELECT FOR UPDATE` e `SELECT FOR SHARE`, comportam-se da mesma maneira que o comando `SELECT`.

5.4 Nível de isolamento (*Repeatable Read*)

Neste nível de isolamento os dados dentro de uma transação apenas são vistos quando é realizado o commit antes dessa transação começar, não são vistos os dados uncommited nem as mudanças ocorridas enquanto uma transação concorrente esta a ser executada.

A grande diferença deste nível para o nível Read Committed é a forma como o snapshot dos dados é realizado. No Repeatable Read o snapshot dos dados dá-se no início da transação, ao contrário do Read Commited cujo snapshot dos dados ocorre na execução do respetivo comando, este nível é mais restrito o que obriga as transações a estarem preparadas para lidar com falhas. Apenas as transações que alterem dados poderão ter a necessidade de ser abortadas, as que apenas leem os dados da base de dados nunca terão conflitos de serialização.

5.5 Nível de isolamento (*Serializable*)

O nível de isolamento *Serializable* é o nível de isolamento mais rigoroso usado pelo *PostgreSQL*. Neste nível as transações são executadas uma após a outra, em serie, ao invés de simultaneamente como nos outros níveis, no entanto, como no nível anterior as aplicações que usarem este nível de isolamento devem de estar preparadas para lidar com falhas e para recomeçar a execução de uma transição que tenha abortado.

Para detetar conflitos neste nível é usado um sistema de monotorização que controla as condições que podem provocar a execução de transações concorrentes de forma inconsciente com qualquer serialização possível entre essas transações. Este controlo não é bloqueante para além do existente do nível anterior, contudo a monotorização de condições de falha sobrecarrega o sistema.

Para garantir uma completa serialização o *PostgreSQL* usa predicado de *locking*, o que significa que mantém *locks* que permitem determinar quando uma escrita tem impacto no resultado de uma leitura feita anteriormente por uma transação concorrente.

Este tipo de *locks* não causam nenhum tipo de bloqueio nas transações e por isso não podem provocar *deadlocks*. Estes *locks* são usados para identificar e assinalar dependências entre transações serializáveis concorrentes que, em certas combinações, podem levar a falhas de serialização.

A monitorização de dependências entre leituras e escritas tem um custo associado mas equilibrado com o custo e a possibilidade de bloqueio associada ao uso explícito de *locks*, o uso de transações serializáveis são a escolha mais eficiente para certos ambientes de execução. Para modificar o nível de isolamento de uma pergunta é preciso indicar através do comando `SET TRANSACTION`.

5.6 *Locks*

São vários os tipos de *lock* usados pelo *PostgreSQL* para controlar o acesso concorrente aos dados presentes nas tabelas. As aplicações que não precisam do total isolamento entre transações fornecido pelo sistema de multi-versão e preferem gerir explicitamente os pontos de conflito. Os *locks* são obtidos implicitamente de tipos apropriados para garantir que tabelas referenciadas não são apagadas ou modificadas duma forma incompatível enquanto o comando executa.

Existem dois tipos de *lock* tendo em conta a granularidade: ao nível da tabela e ao nível da linha (tuplos), quando uma transação realiza o *lock*, o mesmo é mantido até ao final, mas se o *lock* for obtido após ser definido um *save-point* o *lock* é libertado caso a transação faça *roll-back* para o *save-point*, ou seja, se a transação for abortada todas as alterações feitas pela mesma são anuladas. Se os *locks* em questão não forem incompatíveis entre si, é possível existirem varias transações em simultâneo com *locks* sobre a mesma tabela.

Ocorrem algumas incompatibilidades entre tipos de *lock* sobre uma tabela, Tabela 4, não permitindo que duas transações possam ter ao mesmo tempo dois tipos de *locks* incompatíveis sobre essa tabela.

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

Tabela 4: Locks*, modos de conflito*

Vários modos de *lock*:

- **ACCESS SHARE**: O comando `SELECT` obtém este tipo de *lock* nas tabelas referenciadas. Em geral, qualquer pergunta que apenas lê uma tabela mas não a modifica obtém este tipo de lock;
- **ROW SHARE**: Os comandos `SELECT FOR UPDATE` e `SELECT FOR SHARE` obtém este tipo de *lock* nas tabelas às quais são aplicadas os comandos;
- **ROW EXCLUSIVE**: Os comandos `UPDATE`, `DELETE` e `INSERT` obtém este tipo de *lock* nas tabelas às quais são aplicadas os comandos. Geralmente este tipo de *lock* é obtido por qualquer comando que modifique uma tabela;
- **SHARE UPDATE EXCLUSIVE**: Este tipo protege uma tabela contra alterações de esquema concorrentes e de execuções do programa *VACUUM*. Este tipo é obtido pelos comandos `VACUUM` (sem `FULL`), `ANALYZE`, `CREATE INDEX CONCURRENTLY`, e alguns tipos de `ALTER TABLE`;
- **SHARE**: Este tipo protege uma tabela contra alterações de dados concorrentes. Este tipo é obtido pelo comando `CREATE INDEX` (sem `CONCURRENTLY`);
- **SHARE ROW EXCLUSIVE**: Protege uma tabela contra alterações de dados concorrentes e é autoexclusivo para que apenas uma sessão possa ter o *lock*.
- **EXCLUSIVE**: Este tipo permite apenas que possam ser executadas em paralelo leituras à tabela;
- **ACCESS EXCLUSIVE**: Este tipo garante que apenas a transação que tem o *lock* e que pode aceder à tabela. Este tipo é obtido pelos comandos `ALTER TABLE`, `DROP`

TABLE, TRUNCATE, REINDEX, CLUSTER e VACUUM FULL. É também o tipo de *lock* obtido pelo comando LOCK TABLE quando não é especificado nenhum tipo. Este é o único tipo que pode bloquear o comando SELECT (sem FOR UPDATE/SHARE).

Mesmo com o melhor controlo e com algumas políticas para evitar *deadlocks*, esses ainda podem acontecer. Assim que o sistema deteta um *deadlock* uma das transações é abortada e a outra continua.

6. Suporte para bases de dados distribuídas

6.1 Introdução

Servidores de base de dados podem trabalhar em conjunto para permitir que um segundo servidor possa assumir o controlo rapidamente, caso o primeiro servidor tenha uma falha, ou para permitir que vários computadores se sirvam dos mesmos dados. Idealmente, os servidores de bases de dados poderiam trabalhar juntos sem problemas.

Os servidores de bases de dados apenas *read-only* são bastante fáceis de serem combinados, mas infelizmente a maioria dos servidores de bases de dados têm um misto de pedidos de leitura e de escrita de dados e os servidores *read/write* são muito mais difíceis de serem combinados. Isto, devido ao facto de, os dados *read-only* precisarem de serem enviados para todos os servidores uma única vez, a escrita feita num dos servidores tem de ser propagada por todos os servidores para que no futuro os pedidos de leitura desses dados possam ser retornados de forma consistente. Este problema de sincronização é a principal dificuldade para os servidores que trabalham em conjunto porque não existe uma solução única que elimina o impacto do problema de sincronização para todos os casos de uso. Existem várias soluções possíveis e cada solução aborda o mesmo problema de maneira diferente e minimiza o seu impacto para uma carga de trabalho específica.

Algumas soluções lidam com o problema de sincronização permitindo apenas que um único servidor modifique os dados. Os servidores que podem modificar os dados são chamados de *read/write servers*, *master servers* ou *primary servers*. Servidores que esperam por alterações da parte do master chamam-se *standby servers* ou *slave servers*.

Algumas soluções são síncronas, o que significa que uma transação de modificação de dados não é considerada como feita até todos os servidores efetuarem a modificação. Isso garante que caso haja uma falha em algum servidor, todos os servidores sabem que aquela transação falhou e fazem *roll back* à transação. Assim quando houver leitura daqueles dados a informação retornada estará consistente. Pelo contrário, nas soluções

assíncronas é permitido algum atraso entre o tempo da operação ser executada e a sua propagação para os outros servidores, o que possibilita que sejam retornados dados inconsistentes.

6.2 *Log-Shipping*

O mecanismo de *log shipping* consiste em transferir os registos *WAL* de um servidor de bases de dados primário para um servidor de bases de dados secundário. No *PostgreSQL* a implementação é *file-based*, o que significa que os registos *WAL* são transferidos um de cada vez. Os ficheiros *WAL* podem ser enviados de forma fácil para qualquer sistema, seja ele adjacente, o mesmo sistema ou outro sistema do outro lado do mundo. A largura de banda necessária para esta técnica varia de acordo com a taxa de transação do servidor primário, sendo que quanto mais transações houver maior terá de ser a largura de banda já que o *WAL* é atualizado várias vezes. É de notar que o *log shipping* é executado de forma assíncrona e como resultado disso, existe uma janela para perda de dados no caso de o servidor primário sofrer uma falha. As transações que ainda não foram enviadas serão perdidas.

6.3 Replicação por *streaming*

Este método de replicação consiste em os *standby servers* estarem sempre atualizados, isto significa que recebem as alterações do *WAL* assim que são geradas, sem terem de ficar à espera que o ficheiro *WAL* seja preenchido.

6.4 Replicação em cascada

Este método de replicação permite que os *standby servers* recebam as atualizações do *WAL* vindas de outro *standby server*, isto é, alguns *standby servers* recebem as atualizações e propagam a informação por outros *standby servers*, fazendo-se passar por reencaminhadores da informação, e com isto permite reduzir o número de conexões realizadas ao servidor primário.

6.5 Replicação por sincronização

O *PostgreSQL* por defeito utiliza uma abordagem assíncrona, significa que quando o servidor primário falha e algumas transações que foram *commited* podem não ter sido replicadas para o *standby server*, causando perca de dados. A quantidade de perca de dados é proporcional ao atraso de replicação no momento da falha. A abordagem síncrona oferece a possibilidade de confirmar se todas as alterações que foram feitas pela transação, foram executadas de forma síncrona pelo *standby server*.

Quando for requisitada a replicação síncrona, cada *commit* de uma transação de escrita irá esperar até que chegue a confirmação de que o *commit* foi escrito no seu *log* e armazenado no disco dos dois servidores, servidor primário e *stadby server*. A única possibilidade de os dados se perderem é se o servidor primário e o *standby server* sofrerem um crash ao mesmo tempo. Assim, a abordagem síncrona oferece ao utilizador uma maior confiança ao garantir que as alterações feitas não serão perdidas no caso de um servidor falhar, um ponto a desfavor desta abordagem é o facto de haver um acréscimo no tempo de espera que o servidor secundário execute as operações. As transações *read-only* e transações de *rollback* não necessitam de espera pela resposta vinda dos *standby servers*. Só é permitida uma abordagem síncrona aos servidores que estejam ligados diretamente ao servidor primário.

6.6 Outras alternativas

Como o *PostgreSQL* há alguns anos atrás não implementava nenhuma forma de bases de dados distribuída e como é um sistema *open source*, foram aparecendo algumas alternativas de utilização do *PostgreSQL* de forma distribuída. Essas alternativas serão mostradas no quadro seguinte:

Program	License	Maturity	Replication Method	Sync	Connection Pooling	Load Balancing	Query Partitioning
PgCluster	BSD	Stalled	Master-Master	Synchronous	No	Yes	No
pgpool-I	BSD	Stable	Statement-Based Middleware	Synchronous	Yes	Yes	No
Pgpool-II	BSD	Recent release	Statement-Based Middleware	Synchronous	Yes	Yes	Yes
slony	BSD	Stable	Master-Slave	Asynchronous	No	No	No
Bucardo	BSD	Stable	Master-Master, Master-Slave	Asynchronous	No	No	No
Londiste	BSD	Stable	Master-Slave	Asynchronous	No	No	No
Mammoth	BSD	Stalled	Master-Slave	Asynchronous	No	No	No
rubyrep	MIT	Stalled	Master-Master, Master-Slave	Asynchronous	No	No	No

Tabela 5: Alternativas de utilização do PostgreSQL *de forma distribuída*

7. Outras características do sistema estudado

7.1 *XML*

O *PostgreSQL* suporta documentos *XML*, estes documentos tem a capacidade de armazenar dados de forma estruturada. Para usar o tipo de dados *XML*, o utilizador tem, durante a instalação tem de executar o comando: `configure --with-libxml`.

De forma a melhorar o desempenho e garantir que as funções funcionem corretamente, todos os ficheiros *XML* devem utilizar o formato *UTF-8*. Algumas funções *XML* podem não funcionar em dados que não estejam no formato *ASCII* quando a codificação ao servidor não é o *UTF-8*.

O *XML* não obriga o armazenamento de apenas documentos neste formato, sendo que também é possível armazenar fragmentos. Para verificar se um determinado documento é um documento *XML* usamos o comando `IS DOCUMENT`, e o nome do ficheiro que queremos verificar: `xmldoc IS DOCUMENT`.

Para criar um documento *XML* a partir de uma *string* utiliza-se o comando:

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

Para fazer o inverso utiliza-se o comando:

```
XMLSERIALIZE ({DOCUMENT | CONTENT} value AS { character | character varying | texto })
```

O *PostgreSQL* suporta a linguagem de interrogação *XML XPath*, um subconjunto do *XQuery*. A função *XPATH* avalia a expressão *xpath* (um valor de texto) em relação ao valor *xml*. Esta retorna uma matriz com os valores *XML* correspondentes ao conjunto de nós produzidos pela expressão *XPATH*.

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

7.2 Autenticação

Através da autenticação é possível identificar o utilizador e definir os seus privilégios de acesso. O *PostgreSQL* a fim de garantir a segurança no acesso a bases de dados, suporta vários mecanismos de autenticação: *Trust, Password, GSSAPI, SSPI, Kerberos, Ident, LDAP, RADIUS, Certificate* e *PAM*.

7.3 *Triggers*

Os *triggers* mantém a consistência de uma base de dados. Estes podem ser escritos em várias linguagens procedimentos: *PL/pgSQL, PL/Tcl, PL/Perl, PL/Python*. Ou até mesmo em C. A criação de um *trigger* em *PostgreSQL* é realizada através do comando:

```
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* pode ser:

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

8. Comparação com o Oracle 11g

8.1 Indexação e *hashing*

Em termos de indexação e *hashing*, o *PostgreSQL* é bastante parecido ao *Oracle*. No entanto, a grande diferença é o facto de o *PostgreSQL* suportar tipos de indexação mais aprimorados, como o *GIST* e o *GIN*.

8.2 Processamento e otimização de perguntas

O *PostgreSQL* é um sistema muito complexo e com uma grande variedade de algoritmos e de mecanismos que o *Oracle* não tem, isto porque o *Oracle* lida com bases de dados de grandes dimensões e às vezes ter muitos algoritmos e mecanismos para processar perguntas torna-se desvantajoso. Posto isto, verificamos que o sistema *Oracle* tem algumas limitações no processamento de perguntas muito complexas devido às heurísticas utilizadas. Outra diferença é que ao nível de processamento de perguntas que tenham muitas junções, o *PostgreSQL* pode utilizar mais algoritmos de otimização do que o *Oracle*, o que pode prejudicar em termos de tempo de resposta.

O *Oracle* possui mecanismos de suporte a expressões complexas, assim como o *PostgreSQL* e todos os sistemas de base de dados complexos. Mas estes não possuem maneira de definir explicitamente através de comandos quais os mecanismos a serem utilizados. Os sistemas internamente é que avaliam e decidem qual o melhor mecanismo a utilizar.

Ao nível da verificação dos planos de execução escolhidos, o sistema *Oracle* assim como o *PostgreSQL* implementam o comando `EXPLAIN`. Contudo o *PostgreSQL* também permite utilizar o comando `ANALIZE` em conjunto com o comando `EXPLAIN` fazendo com que a pergunta seja mesmo executada podendo analisar nos resultados obtidos as estimativas feitas para a pergunta.

8.3 Gestão de transações e controlo de concorrência

Em termos de gestão de transações e controlo de concorrência, o *PostgreSQL* é bastante semelhante ao *Oracle*, pois utiliza os mesmo mecanismos que o *Oracle* e mais alguns.

8.4 Suporte para bases de dados distribuídas

O sistema *Oracle* suporta bases de dados distribuídas, possibilitante a replicação e fragmentação dos dados pelos vários servidores, assim como possibilita ter mais do que um servidor como servidor primário. Por outro lado, o sistema *PostgreSQL* até há uns anos atrás não tinha suporte para bases de dados distribuídas, mas nas últimas versões começou a implementar esse suporte, tendo atualmente apenas suporte a replicação de dados usando um servidor como primário e os restantes como servidores secundários.

9. Referências

1. Welcome to the PostgreSQL Wiki,
   https://wiki.postgresql.org/wiki/Main_Page

2. PostgreSQL 9.3.4 Documentation,
   http://www.postgresql.org/docs/9.3/interactive/index.html

3. Database System Concepts, 4th Edition,
   Avi Silberschatz, Henry F. Korth e S. Sudarshan

4. PHP: PostgreSQL – Manual,
   http://www.php.net/manual/pt_BR/book.pgsql.php