Análise PostgreSQL

Disciplina: Sistemas de Base de Dados
Docente: José Alferes
Turno: P2
Grupo: 25
Autores: Ana Carvalho nº 42183
Cláudio Borges nº 42554
Eduardo Beja Martins nº 42028

Índice

Introdução

No âmbito da cadeira de Sistemas de Bases de Dados, o presente trabalho pretende comparar dois sistemas de bases de dados: o Oracle 11g e o PostgreSQL. Primeiramente iremos apresentar as características do primeiro sistema para só depois comparar com o segundo. Isto deve -se ao facto de termos ao longo da cadeira direccionado o estudo para o Oracle. Para caracterizar o sistema PostgreSQL analisamos as seguintes temáticas: a gestão de armazenamento, quais as estruturas de indexação suportadas, mecanismos para processamento e optimização de perguntas, e ainda a gestão de transacções e controlo de concorrência.

1 - Origem do PostgreSQL e Aplicabilidade

O sistema PostgreSQL tem origem num projecto da University of California at Berkeley, e incialmente tinha o nome Postgres. O grande responsável por este sistema é Michael Stonebraker que começou a trabalhar nele em 1986 e nesta altura usava a linguagem de query POSTQUEL. O nome Postgres vem de “After Ingres” um sistema de BD que seguia a teoria clássica de um sistema de gestão de BD relacional, também desenvolvido na UCB.
Foi desenvolvido de 1986 a 1994 e era um projecto que tinha como intenção abrir novos rumos no conceito de base dados sendo inclusivé de início comercializado como Ilustra sendo mais tarde comprado pela Informix, empresa que mais tarde foi comprada pela IBM.

Em 1995, o nome foi mudado para Postgres95 e passou a usar SQL ao invés de POSTQUEL. Um ano mais tarde o Postgres 95 deixou de estar centrado na universidade e passou a estar disponível como material open-source começando a ser também desenvolvido por pessoas fora da Universidade. Neste desenvolvimento muitas foram as novas funcionalidades desenvolvidas e assim continuou a sua revolução no mundo dos Sistemas de Gestão de Bases de Dados. Foi nesta altura que surgiu o nome PostgreSQL que que até aos dias de hoje se mantém. Actualmente é utilizada a versão 9 e corre nos Sistemas Operativos principais. Para se comprovar que PostgreSQL é de facto um sistema com capacidades basta perceber que muitas empresas de renome e que até algumas entidades governamentais

americanas o usam. Exemplos disso são: CISCO, NTT Data, NOAA, Research In Motion, US Forestry Service , e, ainda o The American Chemical Society.

2 - Armazenamento e file structure

2.1 Sistema de ficheiros e Armazenamento

O sistema PostgreSQL usa uma organização sequencial de ficheiros, ou seja, cada relação é guardada num ficheiro. Exactamente por cada tabela ser guardada num ficheiro diferente é que este SGBD pode confiar no sistema Operativo, trabalhando assim com ele. PGDATA é a designação da directoria com toda a informação do cluster. No entanto, podem coexistir diversos clusters numa máquina graças à hierarquia usada pelo sistema. Para aceder à localização das bases de dados, basta aceder à directoria /base que se encontra dentro da directoria representada por PGDATA. Antes de perceber como está organizada a hierarquia abaixo de PGDATA/base é necessário entender o conceito OID - Object identifier. O PostgreSQL usa internamente OIDs utilizar em tabelas do sistema. Em geral, estes identificadores não são usados em tabelas criadas pelo utilizador a não ser caso seja pedido. Estes identifcadores, que na verdade são um inteiro, são por exemplo, para o sistema encontrar as bases de dados existentes no cluster. Para encontrar as bases de dados do sistema basta consultar a tabela existente na variável pg_database pela coluna OID, e, caso pretendido também por datname que representa o nome da tabela. Esta questão é importante porque no sistema de ficheiros as bases de dados não são guardadas em directórios com o seu nome, mas sim com o se OID. Ou seja, dentro do caminho PGDATA/base existem directorias com o nome igual ao valor do OID correspondente. Imaginando que existe uma base de dados chamada “SBD” com OID “20”, esta reside no sistema de ficheiros em PGDATA/base/20. No entanto, os OIDs são mais que identificadores de Bases de Dados. Se continuarmos a descer na hierarquia, encontramos os ficheiros da BD acedida, e novamente estes são ficheiros com nomes números, isto é com OIDs. No entanto, neste caso já não se referem a BDs mas sim a tabelas. O OID das tabelas de um BD pode ser obtido através de uma query como : SELECT relname, oid, relpages, reltuples FROM pg_class. Pg_class é a tabela de sistema que guarda informações sobre as relações, e como se pode verificar, nela existem informações como o OID, o número de túplos, número de páginas usadas etc. Resumindo, a hierarquia usada pelo PostgreSQL segue a seguinte lógica: PGDATA/base/[OID_Base_de_Dados]/[OIDS_Tabela]. Associado a cada ficheiro está associado um Free Space Map que guarda a informação sobre o espaço livre na relação.

2.1.1 TOAST

O sistema analisado não permite que registos ocupem mais que uma página (8k no PostgreSQL). Uma implicação directa desta abordagem é que não se permitem atributos de grandes dimensões. É aqui entra o TOAST - The oversized Attribute Storage Technique - uma técnica que faz com que caso haja um tuplo que ocupe mais que uma página, este é guardado numa tabela TOAST que fica associado à tabela do tuplo. Para se saber se existem tabelas que usam tabelas TOAST basta correr o comando:
SELECT relname, reltoastrelid, relpages FROM pg_class;

Ao se pedir o atributo reltoastrelid, obtém-se o OID da tabela TOAST associada à tabela retornada por relname. Caso o valor de reltoastrelid seja 0, então não existe nenhuma tabela TOAST associada.

2.2 . Organização dos tuplos e tabelas

As tabelas são guardadas de forma não ordenada, num heap seguindo a estrutura de uma slotted-page. Esta slotted-page tem a seguinte estrutura: cabeçalho, um array de apontadores que guarda para cada o endereço de memória correspondente e o tamanho do tuplo, e , por fim os tuplos por ordem inversa ao dos apontadores desde o fim da página (slotted-page). Ao usar este tipo de armazenamento, são admissíveis registos de tamanho variável.

2.3 Clustering

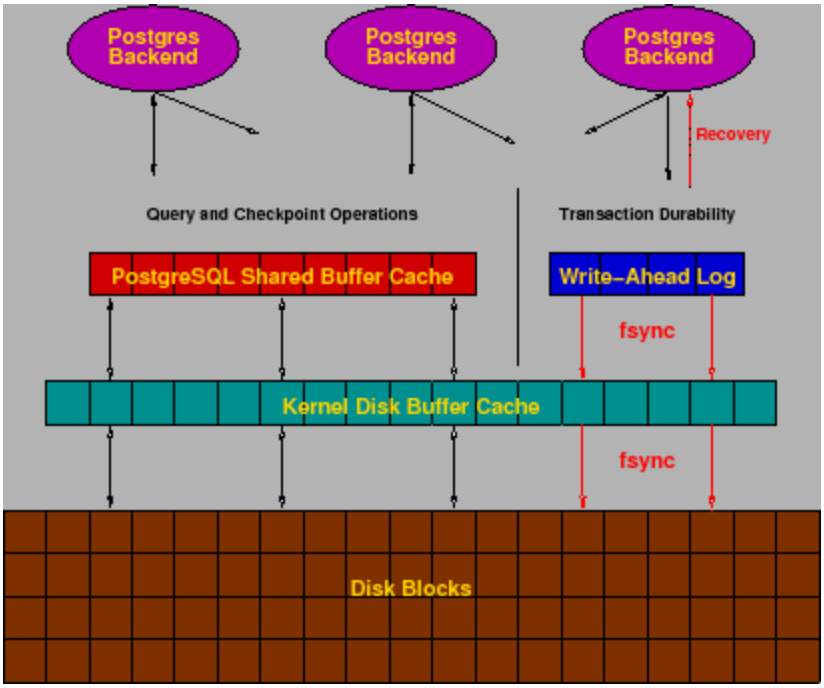
Este sistema não lida com multitable clustering. No entanto é possível fazer com que uma tabela seja ordenada segundo um index. Porém esta trata-se de uma acção que não é actualizada ao longo do tempo. Isto é, a tabela apenas fica ordenada caso não sejam adicionados tuplos ou modificados. Caso se pretenda manter a tabela organizada por aquele index, basta repetir o comando para ordenar. Para o fazer é necessário que exista o index associado àquela tabela. O comando para ordenar uma tabela é:

CLUSTER table_name USING index_name

onde, table_name equivale ao nome da tabela e index_name criado para essa tabela.

2.4 Buffer Management

O PostgreSQL utiliza o seu próprio buffer, pelo que podemos dizer que se sucede um fenómeno de *double buffering* uma vez que os blocos lidos do disco passam para um buffer do SO e depois para um próprio do PostgreSQL e só aí é que são lidos. O inverso acontece quando se trata de uma escrita: os blocos passam pelo buffer do SBD e só depois para o buffer do SO. É consequência imediata desta abordagem a perda de performance, uma vez que implica mais uma leitura/escrita por operação. A imagem seguinte retrata a estratégia apresentada anteriormente.



2.5 Partições

Em termos de particionamento, o PostgreSQL consegue lidar com este tipo de armazenamento caso o utilizador o deseje. Para o fazer, este utiliza herança de tabelas. Isto é, cada partição tem de ser criada como tabela filha de uma tabela pai que está sempre vazia. Trata-se de uma tabela vazia porque apenas existe para representar o conjunto de tabelas filhas.

O sistema PostgreSQL apenas suporta dois tipos de particionamento: por range e por lista. O primeiro refere-se a limitar as partições por intervalos de valor, como por exemplo datas ou valores de salários, etc. Já o seguinte, refere-se a atribuir uma série de valores a cada partição como por exemplo atribuir a cada partição um conjunto de distritos de um país. Para o utilizador poder utilizar partições deve fazê-lo da seguinte forma:

1. Criar a tabela pai vazia. Não se deve adicionar qualquer constraint a não ser que se queira que seja aplicada a todas as partições. Também não faz sentido criar indexes ou restrições unique.

   ```
   CREATE TABLE measurement (
      city_id         int not null,
      logdate         date not null,
      peaktemp        int,
      unitsales       int
   );
   ```

2. Criar diversas tabelas filhas - tantas quanto se precise - que herdam da tabela pai, isto é a tabela criada em 1. Estas tabelas filhas, são tabelas normais, mas a que designaremos desde então por partições. É aqui que se devem adicionar as restrições a cada partição atribuindo os valores de chaves permitidas por cada uma.

   ```
   CREATE TABLE measurement_y2006m02 (
      CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
   ) INHERITS (measurement);

   CREATE TABLE measurement_y2006m03 (
      CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE '2006-04-01' )
   ) INHERITS (measurement);
   ```

   Neste exemplo criaram-se tabelas para cada mês do ano 2006. Como se pode ver a key INHERITS é a que confere a herança à tabela filha.

   Uma aplicação deste exemplo, é que imagine-se que apenas se quer saber os dados dos últimos 12 meses. Então a cada mês que passa não se quer mais saber do mês mais antigo na Base de Dados. Então basta apagar a tabela em questão com um simples DROP TABLE. Não é muito comum apagar-se dados de Bases de Dados, mas de facto pode ser o pretendido. Percebe-se ainda que neste exemplo se faz particionamento por range (1/MÊS/ANO - 30, 31 ou 28 /MÊS/ANO).

3 - Indexação e hashing

3.1 Tipos de Indexação no PostgreSQL

3.1.1 Indexação B+Tree

O PostgreSQL usa B+Trees, criando-as e gerindo-as com algoritmo de Lehman e Yao [10].

Seja k um parametro da árvore tal que 2k seja o número máximo de elementos num nó (à excepção da alta chave).

Este algoritmo caracteriza-se por criar folhas de igual profundidade e garantir que cada nó (excepto a raiz e nós folha) tenha entre k/2 a k filhos, (as folhas têm entre (k-1)/2 e k-1 valores).

O algormo original usa chaves de tamanho fixo e ligações unidireccionais nó-folha, contudo o PosgreSQL implementa-o de forma a poder utilizar chaves de tamanho variável e ligações bi-direccionais.

As B+Trees apresentam bons resultados para *queries* de intervalo e de igualdade, em dados que podem ser ordenados segundo um determinado atributo. O optimizador do PostgreSQL considera usar uma B+Tree sempre que uma coluna indexada se apresenta numa comparação envolvendo um dos seguintes operadores [2]: <, <=, =, >= e > ou construções semelhantes utilizando uma procura por índice B+Tree (por exemplo, BETWEEN e IN) (nota: IS NULL não é equivalente a = e não é indexável).

O optimizador também pode usar índices B+Tree para queries que utilizem operadores de procura de semelhanças por padrões, como o LIKE, ILIKE, ~ e ~* [3], mas apenas no caso de o padrão ser constante e corresponder ao início da cadeia de caracteres. Como exemplo, o operador LIKE poderia ser usado do seguinte modo:

LIKE exemplo%

Mas não

LIKE %exemplo%

3.1.2 Indexação R-Tree

Os índices R-Tree são usados para queries de dados espaciais de duas dimensões. Utilizam o algoritmo de divisão quadrática de Guttman [4], que tenta encontrar a divisão mais pequena num conjunto.

Podemos criar um índice deste género da seguinte forma:

CREATE INDEX nome ON tabela USING rtree (coluna);

onde nome é o nome do índice criado, tabela, tabela de origem e coluna a coluna referenciada.

O optimizador do PostgreSQL considera a utilização de um índice R-Tree sempre que uma coluna indexada se apresenta numa comparação envolvendo um dos seguintes operadores [14]: <<, &<, &>, >>, <<|, &<|, |&>, |>>, ~, @ e ~=.

3.1.3 Indexação por hashes

O PostgreSQL usa índices baseados em funções de *hash* mas apenas suporta comparações simples de igualdade, utilizando o algoritmo de *hashing* Linear de W. Litwin [5], originalmente desenhado para sistemas baseados em discos com um processador. A indexação é feita pela chave primária.

O optimizador do PostgreSQL considera usar este tipo de índice sempre que uma coluna indexada estiver envolvida numa comparação que utilize o operador de igualdade. Um comando semelhante ao seguinte criará um índice baseado em funções de *hash*:

CREATE INDEX nome ON tabela USING hash (coluna);

onde *nome* é o nome do índice, *tabela* a a tabela a aplicar, *hash* a *hash* a utilizar e *coluna* a coluna de origem da chave.

3.1.4 Indexação GiST

Os índices GiST não são um único tipo de índice, mas na verdade uma infraestrutura baseadas em árvores equilibradas na qual muitas estratégias de indexação diferentes podem ser implementadas. De acordo com esta característica, os operadores com os quais um índice GiST pode ser utilizado variam de acordo com a estratégia de indexação. A distribuição habitual do PostgreSQL inclui as classes de operadores GiST equivalentes às classes de operadores da R-Tree, no entanto muitas outras classes estão disponíveis em outros projectos da equipa do PostgreSQL. Actualmente a integração GiST suportar concorrência, recuperação e *update*, previamente apenas disponíveis para índices B-Tree.

São usadas essencialmente para indexar para operações de igualdade ou tipos de dados de representação geométrica ou textual.

Quando implementados, os GiST são usados pelos operadores << , &< , &> , >> , <<| , &<| , |&> , |>> , @> , <@ , ~= e &&.

3.1.5 Indexação SP-GiST

Os indices SP-GiST (*Space Partitioned GiST*, GiST particionados espacialmente), tal como os indices GiST, são uma infraestrutura que suporta várias classes de procura. Suporta árvores de pesquisa particionadas, o que facilita um larga colecção de diferentes estruturas não balançadas, como as *quad-trees*, *k-d trees* ou árvores de sufixo.

O PostgreSQL usa SP-GiST indexes para pontos bidimensionais, que apoiam consultas indexadas e que usam os seguintes operadores de igualdade: << , >> , ~= , <@ , <^ e >^

3.1.6 Indexação generalizada invertida (GIN)

O GIN é uma estrutura que guarda um par de de chave e lista de publicação, em que uma lista de publicação é um conjunto de linhas em que a chave ocorre. (Cada valor indexado pode ter muitas chaves e por consequência o mesmo ID de um linha pode aparecer em várias listas de publicação).

O GIN permite o desenvolvimento de tipos de dados particulars, tal como o GiST. Em particular, as estratégias de indexação do GIN, tal como os operadores associados são da responsabilidade de quem o implementa.

3.2 Combinações de Índices Compostos

É possível compor índices (mesmo compondo o mesmo índice múltiplas vezes) para se tornar possível operações que requeiram referência multipla a determinado atributo, tal como no seguinte caso:

```
SELECT a FROM tabela
WHERE b = 1 OR b= 2 OR b = 3
```

Para combinar índices compostos, o PostgreSQL percorre cada índice necessário e cria um bitmap em memória que fornece a localização dos tuplos que verificam as condições dos índices. Os bitmaps são unidos de acordo com a *query* (usando operações lógicas sobre os mesmos). e os tuplos reais são devolvidos. Como os *bitmaps* são visitados pela ordem física em que estão guardados, a ordenação do índice original é perdida e é necessário um clausula ORDER BY para garantir alguma ordem (assim o optimizador necessita de percorrer o índice várias vezes, o que em ocasiões pode optar por percorrer índices simples mesmo havendo compostos disponíveis).

3.3 Hashing no PostgreSQL

Como referido anteriormente, para indices baseados em hash, o PostgreSQL usa o algoritmo de *hashing* linear de W. Litwin. Os registos são indexados com uma chave primária e a função de hash atribiu a cada chave conjunto de linhas (*bucket*) único. As divisões de buckets são feitas ordenadamente, pelo que caso se chegue ao limite de um bucket, é necessário recorrer a técnicas de overflow.Ao usar o hashing linear, o número de buckets aumenta linearmente e é exactamente do tamanho necessário. Para qualquer número de inserções, a maior parte dos registos são movidos para buckets iniciais pelas divisões, e assim o número de registos de overflow são reduzidos. Os dados são normalmente encontrados com apenas um acesso.

O hashing usado no PostgreSQL é estático, contudo, existem bibliotecas dinamicas disponíves, como é o caso de dynahash.c.

Funções de hashing são disponibilizadas ao utilizador através do módulo *pgcrypto*, que contém as seguintes funções:

3.3.1 digest();

Computa uma hash binária e pode utilizar algoritmso standart como md5, sha1, sha224, sha256, sha384 e sha512.

Utiliza-se como no exemplo:

CREATE OR REPLACE FUNCTION sha1(bytea) returns text AS $$
SELECT encode(digest($1, 'sha1'), 'hex')
$$ LANGUAGE SQL STRICT IMMUTABLE

data é o dado a converter (pode ser um byte ou uma string). O digest tem a seguinte nomenclatura:

digest(data text/bytea, type text) returns bytea

em que *type* é o algoritmo a usar.

3.3.2 hmac();

Calcula a hash MAC e usa-se da seguinte forma:

hmac(data bytea/text, key text, type text) returns bytea

em que a *key* é a chave a utilizar e o restantes argumentes semelhantes ao *digest()*.

3.3.3 crypt();

Usado para encriptar palavras-chave. Suporta bf, md5, xdes e des.

É especificada da seguinte forma:

crypt(password text, salt text) returns text

onde *password* é a palavra-chave original, *salt* o valor obtido pela função *get_salt();*. Esta última função gera um valor aleatório e tem a seguinte respecificação:

gen_salt(type text [, iter_count integer]) returns text

em que *type* é o algoritmo a usar, *inter_count*, o iterador do contador para os algoritmos que o usam (xdes e bf) e *ret* o resultado retornado.

Para além das funções base de hashing, o PostgreSQL ainda implementa várias funções que constituiem a parte criptográfica do openPGP.

4 - Query Processing

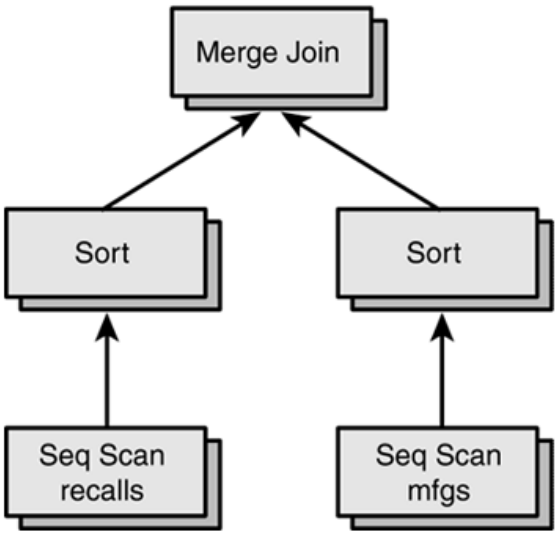
O processamento de um query em PostgreSQL tem os seguintes passos:
Recebe-se a query a executar, logo em seguida faz-se o parsing, do próprio para depois passar a construção do plano de execução, e por fim a execução do plano.

Parser - Logo após o o PostgreSQL receber o query a analisar do cliente, passa o contéudo da query ao parser e este vai verificar se não contém erros de sintaxe. Se a query estiver sintaticamente correcta o parser vai tranformar a o texto da query numa parse tree, em que esta será uma representação da tua query, numa estrutura de dados, tendo em conta as regras apresentadas na query.

Query Rewrite – é responsável pelo sistema de regras do PostgreSQL , essas regras que estão guardadas num catálogo que é utilizado durante o processo de rewrite da query, onde e descobre se ali quais as regras apropriadas para determinado query. O processo de rewrite começa-se por tratar de todos os statements UPDATE, DELETE, INSERT, activando logo todas as regras apropriadas. Depois para as outras tal como select e etc, até que não houver nenhuma regra a ser processada.

Planning e Optimization – depois de ser feito o parsing da query ,é atribuído ao Planner a query tree, e este tem a responsabilidade de ver a parse tree e encontrar todos os planos de execução possíveis para a query em questão. Não se trata de encontrar obrigatoriamente um plano perfeito, mas sim o melhor possível, o menos caro. O processo de optimização é baseado no planeamento standard ou então com o uso do Genetic query optimizer, caso contrário.

Figure 4.6. A simple execution plan.



Executor - depois de escolher o plano de execução pretendido o query executor começa no início do plano e pede aos operadores mais altos para produzirem o resultado.

4.1 - Operações básicas

4.1.1 - Selecção

Sequential Scan - a pesquisa linear é implementada numa tabela de forma normal.Além de ser a melhor forma de acessar a uma tabelas muito pequenas, funciona sempre o que não é preciso criar índices em princípio.
Index Scan - Pesquisa por índices , dado uma ordem no índice retorna um conjunto de tuplos que satisfaçam o predicado e ordenados pela ordem que o índice indicar.Os tipos mais populares de índex incluem, tree, hash tables e bitmaps

4.1.2 - Junção

O PostgreSQL suporta métodos de junção tipo o merge join, nested loop join, hash join com os algortimos estudados nas teóricas..

Nested Loop – Faz o scan na relação da direita porcada tuplo encontrado na relação da esquerda

Merge Join – cada relação é ordenada pelos atributos da junção antes de ser feita a junção. Em cada relação só será scaneada uma vez, e ambas as relações serão pesquisadas em paralelo e todos os tuplos semelhantes serão combinados para formar os tuplos do join.

Hash Join – primeiramente é pesquisada a árvore da direita e guardada dentro de uma hash table usando os atributos de junção como hash keys. A seguir a relação da esquerda é pesquisada e os valores de cada tuplo encontrado é utilizado como hashkeys pra localizar os tuplo correspondentes na tabela.

4.1.3 - Ordenação

Os algoritmos de ordenação existentes são dois, o **external merge sort**, e o **quicksort**. O quicksort é utilizado sempre que a relação cabe em memória e quando isso não é possível é utilizado o external merge sort. E ambas as são parecidas as estudadas nas teóricas.

4.2 - Suporte a Expressões Complexas

No PostgreSQL não existe maneira de o utilizador deifinir qual o mecanismo que vai utilizar. O executor apanha no plano de execução oferecido pelo planner/optimizer e recursivamente processa o para extrair o conjunto de tuplos requeridos, isto essencialmente é um mecanismo de demand driven pipeline, ou seja a medida que são requerido os tuplos vão sendo criados pela operação no topo, o que faz uma cascata para as operações inferiores.

4.3 - Estimativas

Para cada plano feito pelo planner sempre tem como objectivo encontrar um que tenha um bom desempenho.
O comando EXPLAIN do faz com que possamos ver o plano definido para determinada query. E essa instrução tem a seguinte estrutura:

EXPLAIN [(option [, ...])] statement
EXPLAIN [ANALYZE] [VERBOSE] statement
    Onde option pode ser :
    ANALYZE [boolean]
    VERBOSE [boolean]
    COSTS [boolean]
    BUFFERS [boolean]
    TIMING [boolean]
    FORMAT { TEXT | XML | JSON | YAML }

O comando apresenta o plano de execução que o planner do PostgreSQL gera e como as tabelas referenciadas serão pesquisadas, (por pesquisa sequential scan e index scan) e se forem muitas as tabelas referenciadas, quais os algoritmos de junção a serem utilizados para unir os tuplos.Também são apresentados:

-A estimativa do custo inicial
-A estimativa do custo total da operação
-Outras estatísticas se for usado a opção ANALYZE
A opção ANALYZE se for usada faz com que o plano seja executado e não só planeado, e depois mostra as estatísticas reais do runtime, incluindo o tempo total gasto em cada nó do plano e o número total de tuplos retornados, o que serve para ver se a estimativa feita pelo planner é próximo do real.
Também pode se especificar qual o formato do output que se quer, (TEXT, XML, JSON, or YAML) mas o formato default é TEXT.
As estatísticas são actualizadas com o uso do comando ANALYZE.

4.4 - Parametrização

Caso o plano escolhido pelo planner/optimizer para uma query específica não for óptima usa-se parâmetros de configuração para fazer com que o optimizer escolha um plano com uma melhor solução temporária. Os parâmetros são:

enable_bitmapscan (boolean) - ativa ou desativa a criação planos do tipo bitmap scan;
enable_hashagg (boolean) - ativa ou desativa o uso de agregação por hashing;
enable_hashjoin (boolean) - ativa ou desativa o uso de planos do tipo hashjoin;
enable_indexscan (boolean) - ativa ou desativa o uso de planos do tpo index scan;
enable_indexonlyscan (boolean) - ativa ou desativa o uso de planos do tipoindexonlyscan;
enable_material (boolean) - ativa ou desativa o uso de materialização;
enable_mergejoin (boolean) - ativa ou desativa o uso de planos do tipo mergejoin;
enable_nestloop (boolean) - ativa ou desativa o uso de planos do tipo nestedloopjoin;
enable_seqscan (boolean) - ativa ou desativa o uso de planos do tipo seqscan;
enable_sort (boolean) - ativa ou desativa o uso de passos de ordenação explícitos;
enable_tidscan (boolean) - ativa ou desativa o uso de planos do tipo TID scan;

Entre outras formas de melhorar a qualidade do plano escolhido pelo optimizer está o acto de correr o ANALYZE com mais frequência.

5. Gestão de transacções e controlo de concorrência

5.1 Transacções

O PostgreSQL usa intrinsecamente transacções a cada comando SQL interpretado. No entanto, estas podem ser definidas pelo utilizador e para tal iniciar a transacção com BEGIN colocar os comandos pretendidos após esta clausula e depois terminar com COMMIT. Um exemplo seria:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
   WHERE name = 'Alice';
COMMIT;
```

Assim, todas as operações entre BEGIN e COMMIT irão correr como se fossem atómicas no seu conjunto. Caso seja necessário voltar atrás na transacção por alguma operação falhada

usa-se a *keyword* ROLLBACK que irá reverter todas as alterações feitas pela currente transacção.

5.1.1 Savepoints

Em PostgreSQL tal como no Oracle é possível determinar *checkpoints* para impedir que caso haja algum problema com a transacção se volte a repetir todos os passos, que, por vezes, se tratam de longas operações. Estes *checkpoints* são denominados *savepoints* no PostgreSQL. Um exemplo de utilização destes pontos intermédios é o seguinte:

```
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;
```

Os savepoints são então usados para no caso da transacção ser abortada (ROLLBACK) pode reverter apenas as suas alterações após o savepoint. Como se vê no exemplo apresentado, fez-se um ROLLBACK não até ao BEGIN mas até à linha de my_savepoint, um ponto intermédio criado previamente.

Para definir um savepoint utiliza-se a seguinte sintaxe:

```
SAVEPOINT nome_savepoint ;
```

Para voltar atrás numa transacção até a um *savepoint* previamente criado basta usar a seguinte sintaxe:

```
ROLLBACK TO SAVEPOINT nome_savepoint;
```

É importante referir que este mecanismo permite que o utilizador possa escolher e definir as suas transacções como pretende, bastando para tal utilizar *savepoints* como referido neste relatório.

5.1.2 Tipos particulares de transacções

5.1.2.1 Nested Transaction

O sistema de Base de Dados retratado não lida com Nested Transactions.

5.1.2.2 Transacções longas

Nada na documentação do PostgreSQL impede a implementação de transacções longas. No entanto, isto não é aconselhado por senso comum, uma vez que quanto maior for a transacção maior o risco de bloqueio e de rollback das mesmas.

5.2 Isolamento

No sistema PostgreSQL apenas são possíveis intrinsecamente 3 níveis de isolamento ao contra os 4 níveis de isolamento standard SQL. Assim existe a seguir equivalência entre o standard e os níveis aplicados no sistema apresentado:

Nível Standard	Realizado no PosgreSQL
Serializable	Serializable
Repeatable Read	Repeatable Read
Read committed	Read committed
Read uncommitted	Read committed

Para definir o nível de isolamento de uma transacção corrente basta usar o seguinte comando:
SET TRANSACTION nível_isolamento ;
onde nível isolamento é um dos níveis apresentados e escrito em maiúsculas.

5.3 Protocolo de Isolamento

Internamente o PostgreSQL usa o protocolo MVCC - Multiversion Concurrency Control - para manter a consistência de dados. Desta forma, cada pergunta à BD tem uma visão *Snapshot.* Assim, garante -se que todas as operações de leitura numa transacção vêem um estado consistente da BD, e permite ainda aumentar a concorrência. Este aumento vem do facto dos Locks de leitura não interferirem com os Locks de escrita e vice-versa. É importante frisar que um Lock é mantido até ao fim da mesma, ou seja até ao commit, a não ser claro que haja ROLLBACK.

5.3.2 Locks e Granularidade

É possível utilizar dois níveis de granularidade em PostgreSQL : a nível da tabela ou a nível do tuplo. Em seguida apresentam-se os Locks disponíveis para cada tipo e as suas características.

- Nível Tabela
    - **AccessShareLock (ASL)** - Um lock de leitura adquirido automaticamente nas tabelas que são alvo de queries.
    - **RowShareLock (RSL)** - Adquirido pelos comandos SELECT FOR UPDATE e por LOCK TABLE nas queries em **SHARE MODE**.
    - **RowExclusiveLock (REL)** - Adquirido pelos comandos : UPDATE, DELETE, INSERT e LOCK TABLE em modo IN **ROW EXCLUSIVE MODE**.
    - **ShareLock (SL)** - adquirido pelos comandos : CREATE INDEX e LOCK TABLE no modo **SHARE MODE**.
    - **ShareRowExclusiveLock (SREL)** - adquirido pelos comandos : LOCK TABLE no modo **SHARE ROW EXCLUSIVE MODE**.
    - **ExclusiveLock (EL)** -adquirido pelos comandos : LOCK TABLE no modo **EXCLUSIVE MODE.**

- **AccessExclusiveLock (AE)** - adquirido pelos comandos : ALTER TABLE, DROP TABLE, VACUUM e LOCK TABLE.

Na tabela seguinte, representamos as compatibilidades entre os Locks apresentados. A sombreado encontram-se as incompatibilidades.

Lock	ASL	RSL	REL	SL	SREL	EL	AE
ASL							
RSL							
REL							
SL							
SREL							
EL							
AE							

- Nível Tuplo

Este tipo de Locks são adquiridos aquando de updates em tuplos, ou ainda, na sua remoção. Estes Locks actuam a nível das escritas num tuplo específico sendo que as leituras ocorrem sem problema. É importante referir que não existe qualquer limite no número de Locks em tuplos em determinado momento. Depois das alterações nos tuplos serem efectuadas, os Locks são libertados.

5.3.3 Deadlocks

Este sistema detecta automaticamente situações de *deadlock* e simplesmente aborta uma das transacções envolvidas nesse *deadlock*. Assim, as que se mantêm em execução podem terminar sem problema. O aborto da transacção significa que existe um *rollback* na mesma.

5.4 Consistência, Atomicidade e Durabilidade

A atomicidade neste sistema é garantida através do uso de transacções. Não só daquelas definidas pelo utilizador, mas porque cada instrução é convertida numa transacção. Assim, é garantido que ou todas as alterações são guardadas ou nenhuma tem sucesso, e, mesmo assim será possível voltar atrás e voltar a executar. Os mecanismos de recuperação usados foram referidos ao longo desta secção.

Quanto à durabilidade, o PostgreSQL usa um Log para manter a BD consistente no tempo. Assim, em caso de falha, a técnica Write-ahead logging - WAL - permite que ao reiniciar a máquina esta se mantenha consistente. O WAL mantém a integridade dos dados ao apenas fazer alterações no disco depois de escrever no log os detalhes dessas alterações. Esta técnica também permite que não sejam despejados para o disco todos os dados no fim de cada transacção, uma vez que esta estão guardadas no Log todas as alterações a ser feitas e mesmo que haja um crash as mudanças a fazer não foram perdidas, e vão ser refeitas e então *flushed* para o disco.

Caso se use o nível de isolamento: Serializable, não é necessário criar qualquer controlo de consistência. Caso contrário terão de se usar Locks de tabelas.

Com estas características é possível manter a Base de Dados **consistente**, a respeitar a **atomicidade** e a **durabilidade** das transacções.

6 - Suporte para Bases de Dados Distribuídas

O PostgreSQL nativamente não apresenta um suporte para base de dados distribuídos, mas é utilizado algumas soluções como aplicações externas que utilizam os métodos de replicação específicas deles.
Entre os exemplos de aplicações externas usa-se diferentes métodos de replicação e também podem ser acedidos a algumas funcionalidades com extensões do PostgreSQL tal como PostgreSQL-XC.
Os métodos de replicação podem variar de programa para programa, e os métodos podem ser assíncronos ou não. Desde a versão 9.0 que o PostgreSQL faz o uso de técnicas de replicação, depois foram implementadas em versões posteriores replicações síncronas.

Hot Standby/Streaming Replication - oferecem uma replicação binária assincrona a um ou mais standbys. Este é o tipo de replicação mais rápida disponível com os dados WAL (Write-Ahead Logs) a serem enviados de imediato em vez de esperar que um segmento complete seja criado e enviado.

Warm Standby/Log Shipping - é uma solução que faz a replicação de um cluster de base de dados para um arquivo, ou um servidor em standby. É muito fácil, simples e uma solução apropriada se o que se considera mais importante é ter um backup contínuo e um baixo failover-time.

Exemplos de programas e tipos de replicação.

Programa	Maturidade	Método de Replicação	Sincronismo
PgCluster	Estagnado	Master-Master	Sincrono
pgpool-I	Estável	Statement-Based Middleware	Sincrono
Pgpool-II	Novo lançamento	Statement-Based Middleware	Sincrono
slony	Estável	Master-Slave	Assincrono
Bucardo	Estável	Master-Master, Master-Slave	Assincrono
Londiste	Estável	Master-Slave	Assincrono
Mammoth	Estagnado	Master-Slave	Assincrono
rubyrep	Estagnado	Master-Master, Master-Slave	Assincrono

7- Outras Caraterísticas do PostgreSQL

7.1 - Serviços suportados

O PostgreSQL tem um suporte para XML mas só para algumas caraterísticas. Estas features são o armazenamento, importação e exportação, validação, eficiência de modificação , indexação e a transformação de XML em SQL. O uso deste tipo de dados requer a instalação com o comando : **configure --with-libxml.** A vantagem do XML sobre o texto normal é que com este é mais fácil gravar informações e de forma mais organizada.Para criar um documento XML através de texto usa se o 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>')
```

E caso o utilizador quiser fazer o contrário, ou seja passar do código XML para o texto com o comando xmlserialize

XMLSERIALIZE ({ DOCUMENT | CONTENT } type AS
{ character | character varying | text }
Implementa ainda uma linguagem de interrogação XML de nome XPATH,que é como um subconjunto do XQuery e assim faz se consultas aos dados utilizando expressões simples ao estilo XML.

O PostgreSQL executa stored procedures de um número elevado de linguagens de programação tais como Perl, Java, Python, Ruby, C/C++ Tcl e a sua própria linguagem PL/pgSQL e tal como essas linguagens suportadas pelo PostgresSQL também tem existem inúmeras interfaces de bibliotecas que possibilitam que oferecem o PostgreSQL uma interface e interaja com um conjunto de linguagens compiladas e interpretadas.
Entre elas estão interfaces para linguagens para tal como Java(JDBC), ODBC, Python, Perl, C, C++, Ruby, PHP , Scheme , Qt e Lisp, entre outros.
O sistema não apresenta um suporte para Web Semântica tal como SPARQL ou RDF/Schema, mas dá a possibilidade de fazer a ligação entre as duas com uso de ferramentas.
O PostgreSQL também suporta JSON, que permita trabalhar com dados JSON,e tem suporte á utilização de triggers.

7.2 - Tipos de Dados suportados

Integer: BIGINT, INTEGER, SMALLINT.
Decimal: DECIMAL, NUMERIC**.**
Floating Point: DOUBLE PRECISION, REAL.
String: CHAR, CHARACTER, CHARACTER VARYING, TEXT, VARCHAR**.**
Binary: BYTEA.
Boolean: BOOLEAN
Date/Time: DATE, INTERVAL, TIME, TIMESTAMP**.**

7.3 - Ferramentas Disponíveis

Psql O principal frontend para PostgreSQL e é uma aplicação que executa sobre a consola, e é utilizado para consultas SQL de forma directa, ou execução de operações apartir de ficheiros.

pgAdmin ferramenta de administração de interfaces gráficas para PostgreSQL, , também muito utilizado compatível com inúmeros sistemas operativos.

phpPgAdmin aplicação web de administração de bases de dados PostgreSQL, escrita em PHP e baseada na interface do phpMyAdmin

PostGIS - componente usado para oferecer compatibilidade com objectos geográficos.

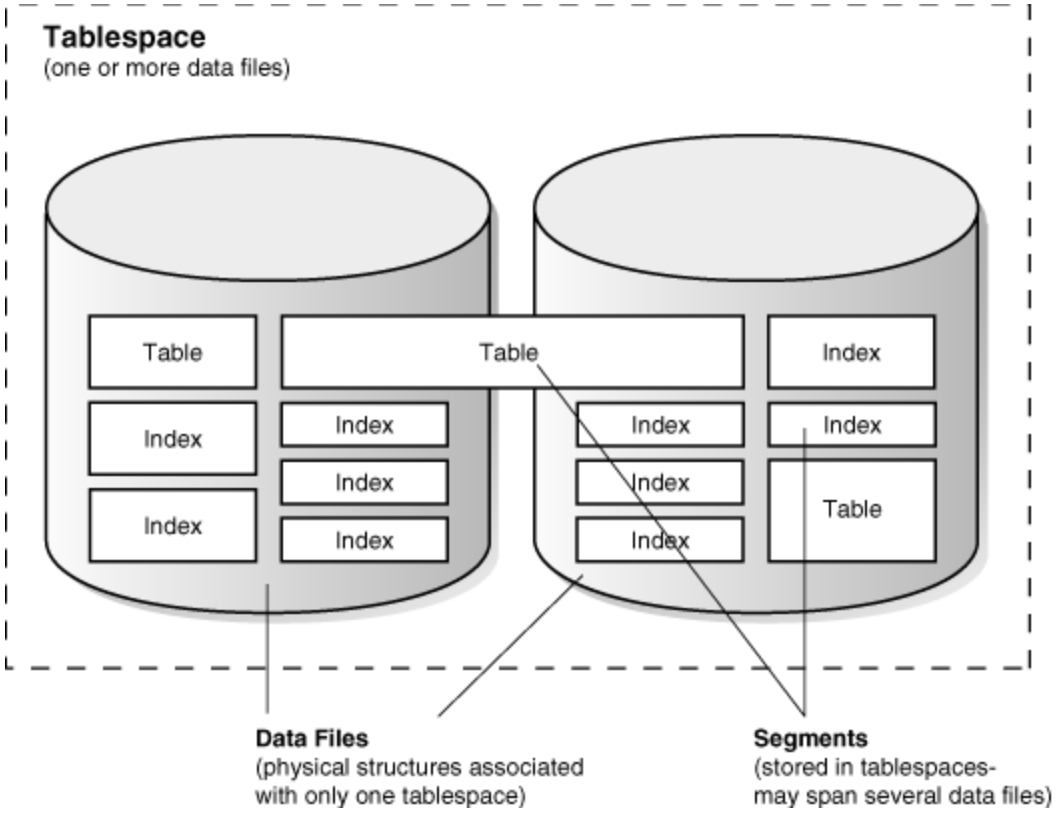
8. Comparação com o Oracle

8.1 Armazenamento e estrutura de ficheiros

8.1.1 Armazenamento

O Oracle instancia uma base de dados directamente em memória persistente, dividindo os seus ficheiros em 3 categoris:
- ficheiros de dados (guardam os dados: tabelas e indexes da base de dados entre si)
- ficheiros de controlo (guardam os meta-dados da base de dados)
- o Online Redo Log (guarda as informações de alterações feitas à base de dados)

Um ficheiros de dados, ao contrário do PostgreSQL pode guardar várias tabelas e vários indexes, ou mesmo parte de uma tabela, como representado na seguinte figura:



Uma base de dados em Oracle pode conter apenas um ou mais ficheiros de dados, segmentados em tabelas e indexes.

As configurações em oracle são guardadas num únicoficheiro de controlo, ao contrário do PostgreSQL. Um base de dados tem um ficheiro de controlo e um ficheiro de controlo corresponde a uma e uma só base de dados.

O registo de alterções no Oracle é feito no Redo Log à semelhança do Write-Ahead Log do PostgreSQL. A maior diferenºa entre os dois difere na configuração. Enquanto que o PostgreSQL usa sempre um ficheiro WAL de 16MB, o Oracle decide o número de logs que quer usar para uma base de dados.

No Oracle, as tabelas têm um limite de 4GB, enquanto que no PostgreSQL tem um timite de 32TB. No primeiro um coluna pode ter 30 caracteres de designação e pode conter até 1000 colunas por linha, enquanto que no PostgreSQL, 63 caracteres e 250 a 1600 colunas, consoante o tipo. Cada linha pode chegar aos 8KB e 1.6TB no Oracle e no PostgreSQL, respectivamente.

O Oracle, ao contrário do PostgreSQL suporta particionamento composto e por *hashing*.

8.1.2 Estrutura de ficheiros

O Orcale cria um ficheiro para tabelas, alocando o espaço necessário para as suas entradas mais um cabeçalho. que contem metadados sobre o próprio ficheiro como o seu tamanho, o ultimo ponto de alteração da base de dados (o ponto de controlo SCN, para recuperação) e o número absoluto e relativo do ficheiro (identificadores do ficheiro para a base de dados e para a *tablespace*).

Um ficheiro de dados é alocado com determinado tamanho que é enchido com a necessidade, por extensões de segmentos. Extensões podem ser alocadas e libertadas e novamente reutilizadas, conforme podemos ver na seguinte imagem:

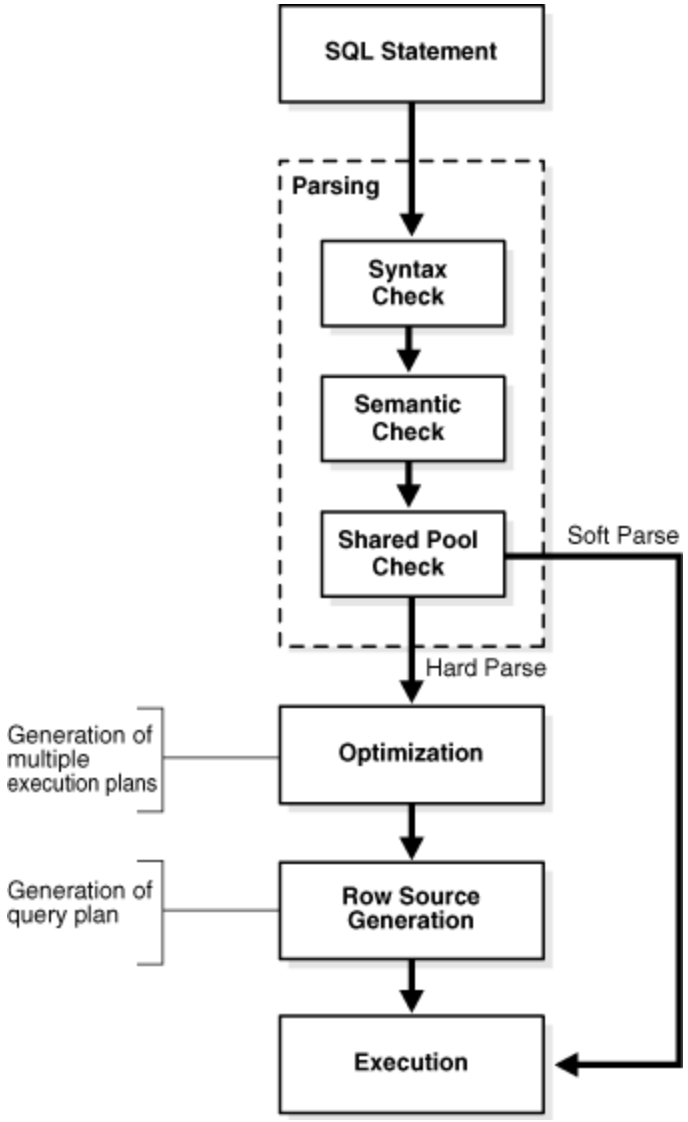
8.2 Indexação e hashing

O Oracle 11g suporta todos os tipos de indexação que o PostgreSQL suporta à excepção de GiST e GIN.

O Oracle suporta os seguintes algoritmos de hashing: MD5, SHA-1, MD4, HMAC_MD5 e HMAC_SH1, ou seja não suporta SHA224, SHA256, SHA384 e SHA512 nem MD5 para hashing criptográfico simples, mas suporta MD4, em comparação com o PostgreSQL.

8.3 Processamento e optimização de perguntas

O Oracle processa *queries* de forma bastante semelhante ao PostgreSQL, actuando da seguinte forma:



Aqui *soft parse* é o *parsing* feito quando existe código gerado para a *query* feita, e *hard parse*, quando nenhum código possa ser usado para responder à *query*.

O Oracle, tal como o PostgreSQL tem como linguagem intemédia o C, mas também o C++.

8.4 Gestão de transações e controlo de concorrência

O Oracle, tal como o PostgreSQL permite o uso de um mecanismo de *locks* que permite controlar a partilha de um recurso, mas, ao contrário do PostgreSQL, ao nível do tuplo apenas.

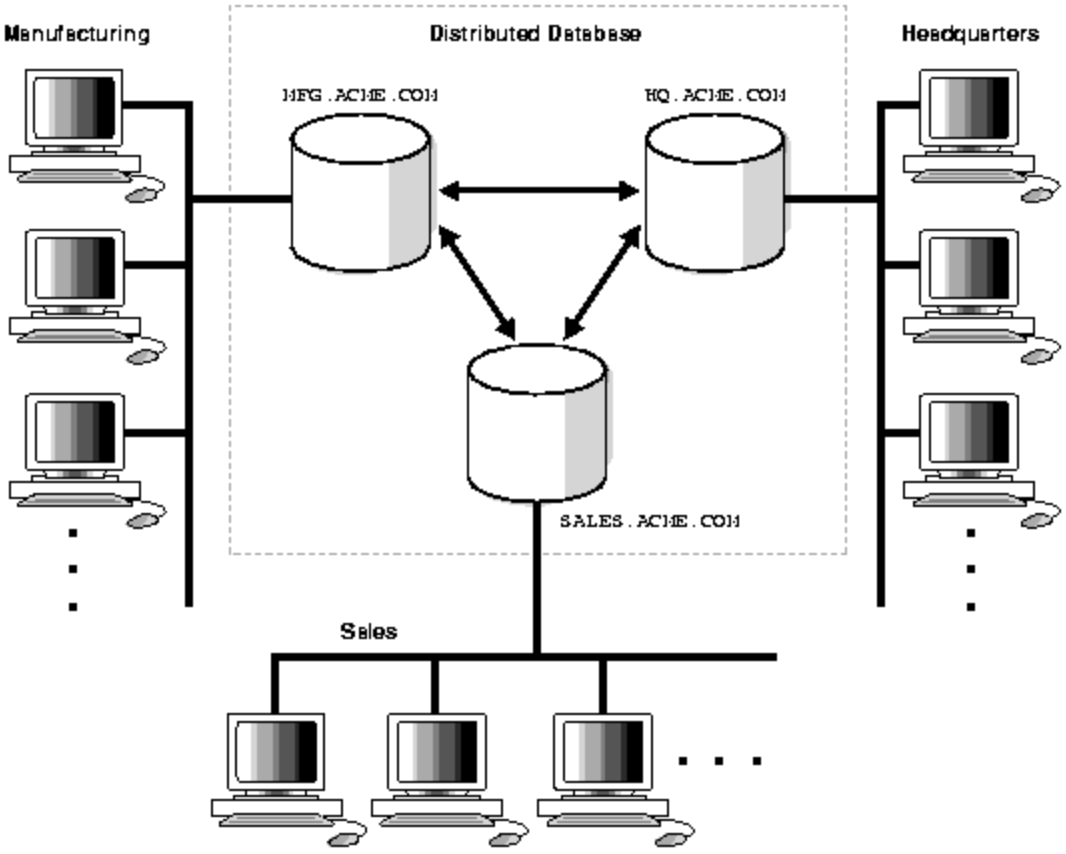
O Oracle tem dois modos de bloqueamento: *exclusive lock* e *share lock*. O primeiro protege um recurso especifico de ser partilhado enquanto activo, enquanto que o segundo permite acesso concorrente ao ficheiro (com escrita simultânea bloqueada).

Existe também aqui a possibilidade de *deadlock*, com resultado em terminação dos processos concorrentes.

São consideradas também, à semelhança do PostgreSQL, transacções com vários níveis de isolamento: *read commited* (dados de antes da *query* viziveis), *serializable* (dados de antes e após a *query* visíveis) e *read-only* (dados após o *commit* apenas visíveis).

8.5 Suporte para bases de dados distribuidas

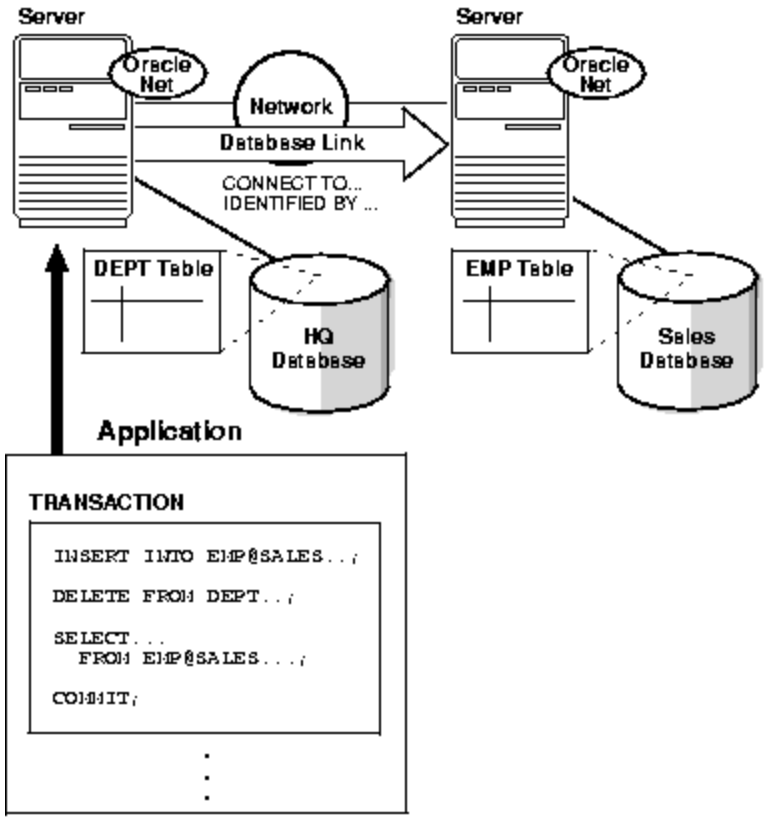
O Oracle, ao contrário do PostgreSQ, suporta bases de dados nativamente, como no seguinte exemplo:



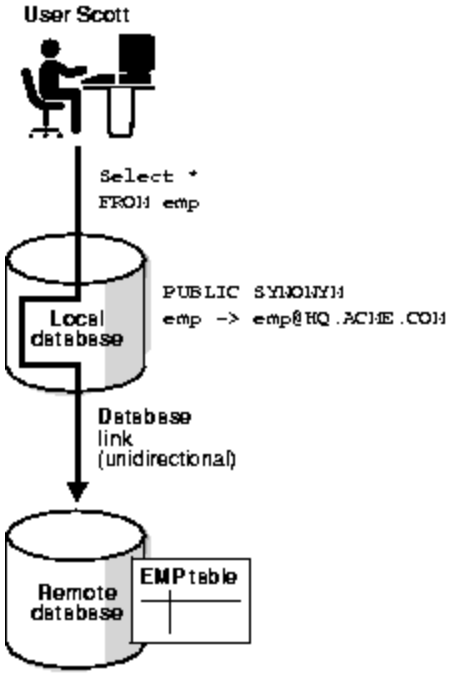
Umas base de dados distribuída Oracle, pode aceitar inclusivo vários subsistemas utilizando versões diferentes do Oracle. Isto é possível através do Oracle Transparent Gateway.

O Oracle permite ainda incorporação ou comunicaão com sistemas diferentes do Oracle, utilizando um dos seguintes agentes: *Heterogeneous Services ODBC* ou *Heterogeneous Services OLE DB*.

O Oracle permite atribuir nomes, designados de *sinónimos*, a recursos externos Um acesso a um recurso dentro do servidor local é chamado de directo e remoto, de indirecto, como é o caso do acesso à tabela DEPT e EMP, respectivamente no seguinte exemplo:



Também podemos perceber a declaração e utilização de sinónimos no seguinte caso:



A designação *emp* foi associada como sinónimo à ligação indirecta *emp@HQ.ACME.COM* e, a partir daí, o sistema local reconhece-a como simplesmente *emp*, permitindo assim maior transparência ao sistema.

8.6 Outras comparações

O Oracle suporta os seguintes tipos de dados a mais que o PostgreSQL: NUMBER, BINARY_DOUBLE, BINARY_FLOAT, NCHAR, NVARCHAR, Unknown, LONGRAW, RAW, AUDIO, DICOM, IMAGE, SPATIAL, VIDEO. XMLTYPE.

O PostgreSQL suporta os seguintes tipos de dados a mais que o Oracle: BIGINT, INTEGER, SMALLINT, DOUBLE PRECISION, REAL, DECIMAL, CHARACTER, CHARACTER VARYING, TEXT, BIT, BOOLEAN, BYTEA, INTERVAL, TIME, ARRAYS, CIDR, CIRCLE, ENUM, GIS data types, INET, MACCADDR, MONETARY, PATH, POLYGON, SEQUENCE, TIMESTAMP, UUID e XML. Permite aindaao utilizador defenir os seus tipos de dados.

O Oracle está desponível para os sistemas operativos AIX, HP-UX, Linux, OS X, Unix, Solaris, Windows e z/OS. O PostgreSQL está disponível para os sistemas HP-UX, Linux, OS X, Solaris, Unix, BSD e Windows.

O Oracle disponibiliza as seguintes API's: ODP.NET, Oracle Call Interface (OCI), JDBC e ODBC. Estas são compatíveis com as linguagens C, C#, C++, Clojure, Cobol, Eiffel, Erlang, Fortran, Groovy, Haskell, Java, JavaScript, Lisp, Objective C, OCaml, Perl, PHP, Python, R, Ruby, Scala, Tcl e Visual Basic.

O PostgreSQL estã presente nas bibliotecas naticas do C e disponibiliza as seguintes APIs: ADO.NET, JDBC e ODBC. Estas são compatíveis com as seguintes linguagens: .Net, C, C++, Java, Perl, Python e Tcl.

O Oracle 11g não suporta ficheiros XML nativamente, mas o Oracle 12c já suporta bases de dadnos nativas em XML.

Referências

[1] Efficient Locking for Concurrent Operations on B-Trees, Lehman e Yao, 1981
[2] http://www.postgresql.org/docs/8.1/static/functions-comparison.html
[3] http://www.postgresql.org/docs/8.1/static/functions-matching.html
[4] A. Guttman. R-Trees: a dynamic index structure for special searching. In Proc. ACM SIGMOD, p. 47-57, Boston, Mass, June 1984
[5] W. Litwin. Linear Hashing: A new tool for file and table addressing. 1980
[6] http://www.postgresql.org/docs/9.2/static/spgist.html
[7] http://www.postgresql.org/docs/8.2/static/gin.html
[8] http://www.confio.com/logicalread/oracle-11g-hash-indexes-mc02/
[9] http://www.confio.com/logicalread/oracle-11g-bitmap-join-indexes-mc02/
[10] http://docs.oracle.com/cd/E11882_01/server.112/e25789/indexiot.htm
[11] http://psoug.org/snippet/Oracle-PL-SQL-INDEXES-Reverse-Key-Indexes_811.htm
[12] http://www.postgresql.org/docs/8.3/static/pgcrypto.html
[13] http://www.postgresql.org/docs/9.1/static/transaction-iso.html
[14] http://db-engines.com/en/system/Oracle%3BPostgreSQL
[15] http://www.moeding.net/archives/37-PostgreSQL-WAL-vs.-Oracle-Redo-Log.html
[16] http://gerardnico.com/wiki/database/oracle/sql_processing
[17] http://shabrinath.wordpress.com/2009/06/22/transaction-and-concurrency-control-in-oracle/
[18] http://docs.oracle.com/cd/A57673_01/DOC/server/doc/SCN73/ch10.htm
[19] http://database-management-systems.findthebest.com/compare/36-43/Oracle-vs-PostgreSQL
[20] http://docs.oracle.com/cd/B10501_01/server.920/a96521/ds_concepts.htm#20409
[21] http://www.oracle.com/technetwork/database/database-technologies/xmldb/overview/index.html
[22] http://www.postgresql.org/docs/current/static/sql-createindex.html
[23] http://www.postgresql.org/docs/current/static/sql-dropindex.html
[24] http://www.postgresql.org/docs/8.1/static/sql-analyze.html
[25] http://www.postgresql.org/docs/8.1/static/sql-update.html
[26] http://www.postgresql.org/docs/8.1/static/sql-delete.html
[27] http://www.bluerwhite.org/btree/
[28] http://nlhgis.nlh.no/gis/applets/rtree2/jdk1.1/help.html
[29] http://en.wikipedia.org/wiki/Hash_function
[30] http://www.mcs.vuw.ac.nz/technical/software/PostgreSQL/gist.html
[31] http://www.databasejournal.com/features/oracle/article.php/3706506/Hashing-in-Oracle.htm
[32] http://www.postgresql.org/about/history/
[33] **http://www.postgresql.org/docs/9.0/static/storage-file-layout.html**

[34] http://etutorials.org/SQL/Postgresql/Part+I+General+PostgreSQL+Use/Chapter+4.+Performance/How+PostgreSQL+Organizes+Data/
[35] http://momjian.us/main/writings/pgsql/hw_performance/
[36]**http://www.postgresql.org/docs/9.2/static/sql-explain.html**
[37]**http://www.slideshare.net/Sammy17/chapter26-postgresql**
[38]**http://etutorials.org/SQL/Postgresql/Part+I+General+PostgreSQL+Use/Chapter+4.+Performance/Understanding+How+PostgreSQL+Executes+a+Query/**
[39]**http://www.neilconway.org/talks/executor.pdf**
[40]**https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxyb2JlcnRtaGFhc3xneDoxM2YzMTA0ZTMwZTYyNWZk**
[41]**http://www.postgresql.org/docs/8.1/static/runtime-config-query.html**
[42]**https://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling**
[43]**http://www.postgresql.org/docs/9.2/static/datatype-xml.html**
[44]**http://www.theregister.co.uk/2014/05/16/postgresql_nosql_conversion/**
[45]**http://www.postgresql.org/docs/8.2/static/datatype-xml.html**
[46]**https://wiki.postgresql.org/wiki/XML_Support**
[47]**http://www.postgresql.org/about/**