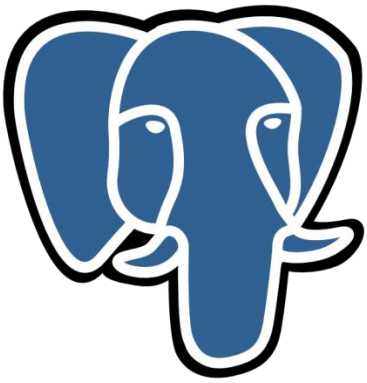
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Sistemas de Bases de Dados

Grupo 12



PostgreSQL

Docente: José Júlio Alferes

Trabalho realizado por:

Henrique Ataíde Nº 42392

João Serrado Nº 41849

Miguel Ferreira Nº 41983

1. Índice

2. Introdução

Neste trabalho, efectuado no âmbito da cadeira de Sistemas de Base de Dados (SBD), optámos por estudar o Sistema Gestão de Bases de Dados (SGBD) PostgreSQL. Esta escolha deve-se ao facto de este ser o SBGD onde nos é disponibilizado mais informação e por ser o open source mais avançado.Para além disso, existem outros factores preponderantes para a escolha deste SGBD:

- Trata-se de um sistema muito escalável e flexível;
- Encontra-se desenhado para ambiente que englobam grandes volumes de dados, dando ênfase à sua robustez;
- Polivalência em termos de execução de linguagens de programação onde se inclui o Java, C/C++, Python e ainda um PL/SQL muito similar ao do SGBD Oracle;
- O facto de ser um SBGD prestigiado e já ter dado provas do seu desempenho em termos comerciais e académicos.

O objectivo deste trabalho é aplicar os conhecimentos adquiridos ao longo do semestre estudando outro SGBD, vendo o seu funcionamento e comparando o mesmo com o estudado nas aulas.

Este relatório, mais do que um trabalho de investigação, pretende que os alunos fiquem a saber o funcionamento do PostgreSQL a vários níveis:

- Como funciona este SGBD;
- Noções de indexação e hashing aplicados a este sistema;
- Processamento e optimização de perguntas neste SGBD;
- Tratamento de transacções concorrentes no PostgreSQL e processamento das mesmas;
- Sistema de segurança neste SGBD;
- XML aplicado ao PostgreSQL;

2.1. Breve Introdução Histórica do Sistema

O PostgreSQL, originalmente chamado Postgres, foi criado na Universidade da Califórnia em Berkeley (UCB) por um professor de ciência da computação chamado Michael Stonebraker, que se tornou o CTO da Informix Corporation. Stonebraker começou a trabalhar no Postgres em 1986 como um projeto de acompanhamento ao seu antecessor, Ingres, agora propriedade da Computer Associates. O Ingres, desenvolvido 1977-1985, foi um exercício de criação de um SGBD de acordo com a teoria clássica RDBMS. O Postgres, desenvolvido entre 1986 e 1994, foi um projecto feito com o objectivo de inovar o conceito de SGBD e a exploração de tecnologias de objectos relacionais.

Stonebraker e os seus alunos de pós-graduação desenvolveram activamente o Postgres durante oito anos. Durante esse tempo, foram introduzidas no Postgres regras, procedimentos, tipos extensíveis com índices e o conceito de objectos relacionais.

Em 1995, dois alunos Ph.D. do laboratório de Stonebraker, Andrew Yu e Jolly Chen, substituíram a linguagem de consulta PostQUEL do Postgres por um subconjunto extendido do SQL. Eles renomearam este sistema para Postgres95.

Em 1996, o Postgres95 passou a ser um sistema open source, quando um grupo de programadores fora de Berkeley apercebeu-se do potencial do sistema e dedicou-se ao seu desenvolvimento continuado. Dedicando uma grande quantidade de tempo, habilidade, trabalho e perícia técnica, este grupo de desenvolvimento global transformou radicalmente o Postgres. Ao longo dos próximos oito anos, eles introduziram consistência e uniformidade no código-base, criando testes detalhados de regressão para a garantia de qualidade, criando listas de discussão de relatórios de bugs, corrigindo inúmeros bugs, acrescentandonovas funcionalidades e preenchendo várias lacunas tal como a documentação para programadores e utilizadores.

A qualidade do seu trabalho foi essencial para o aparecimento de uma nova base de dados que ganhou uma grande reputação. Com a sua chegada ao mundo open source, com muitos novos recursos e aprimoramentos, este SGBD passou a ter o seu nome actual: PostgreSQL.

No que diz respeito aos recursos do SQL, muitas melhorias foram efectuadas a nível das subconsultas, dos padrões, das restrições, das chaves primárias, das chaves estrangeiras, identificadores, do tipo string, do tipo de concatenação, e dos inteiros binários e hexadecimal, entre outros.

Hoje em dia, a base de utilizadores do PostgreSQL está maior do que nunca e inclui um grupo considerável de grandes empresas que a utilizam em ambientes exigentes. Algumas destas empresas como a Fujitsu e a Afilias fizeram contribuições significativas para o desenvolvimento do PostgreSQL. A versão 8.0 era a estreia aguardada do PostgreSQL e trouxe recursos como tablespaces, procedimentos armazenados em Java, entre outros. Com esta versão veio também um recurso muito esperado: uma porta nativa para o Windows.

3. Armazenamento e file structure

3.1. Buffer management

Os acessos ao disco rígido acarretam custos elevados, logo é necessário utilizar um mecanismo que permita armazenar temporariamente dados na memória do computador. Os sistemas de operação já possuem por si um gestor de memória, no entanto devido à natureza das operações realizadas por um SGBD e por forma a melhorar o desempenho, normalmente é implementado um gestor de memória próprio.

O PostgreSQL possui buffer management: buffer de cache partilhada, cuja função principal é controlar a quantidade e a forma como a informação é guardada em disco e consegue-o fazendo a transição dos dados entre o disco rígido e o backend do sistema.

Quando é necessário algum acesso a dados, primeiro é verificado se os mesmos já se encontram em cache. Se tal se verificar, estes poderão ser processados imediatamente, caso contrário terá que ser efectuado um pedido ao sistema de operação para que sejam carregados. A partir daqui podem ser realizadas operações de leitura/escrita, que posteriormente esta informação será escrita em disco.

Por defeito nas configurações do PostgreSQL são alocados 1000 shared buffers com 8 kilobytes cada. O número de buffers pode ser especificado modificando o valor de shared_buffers no ficheiro de configurações postgresql.conf. O valor ideal é aquele que permite responder a partir da cache à maioria dos pedidos evitando o swap de páginas.

Inicialmente todas as entradas da cache estão vazias mas são rapidamente preenchidas. De modo a libertar espaço no buffer, existe uma rotina chamada VACUUM, que verifica periodicamente se existem dados obsoletos ou desactualizados nas tabelas do buffer e, caso existam, actualiza os respectivos conteúdos na memória física. O PostgreSQL usa também uma política LRU (Least Recently Used) para decidir que páginas carregar em memória e que páginas libertar, em função do seu estado de actualização e da frequência que a informação é acedida. Juntamente com esta política, é ainda implementado um serviço que efectua uma contagem das vezes que

um determinado dado é utilizado. Assim sendo, sempre que uma nova página é carregada em cache esta vai substituir a página anterior com menor número de acessos, sendo esta obtido através do clock sweep. No clock sweep a cache do buffer é tratada como uma lista circular, ou seja, sempre que se chega ao final do buffer volta-se automaticamente ao início. Antes de se libertar a página menos acedida, é verificado se foram feitas alterações e, caso existam, os novos valores são gravados em disco.

3.2. File System

Esta secção descreve o formato de armazenamento a nível dos ficheiros e da directoria.

Tradicionalmente, a configuração e os ficheiros de dados usados por uma base de dados são guardados dentro da pasta do cluster, usualmente com nome de PGDATA. Uma localização comum para a PGDATA é /var/lib/pgsql/data. Clusters múltiplos, geridos por diferentes servidores, podem existir na mesma máquina. É possível consultar a lista de identificadores (base de dados) no ficheiro pg_database.

A directoria da PGDATA contém várias subdirectorias e ficheiros de controlo, tal como se pode verificar na tabela seguinte. Para além destes ficheiros essenciais, aqueles que dizem respeito à configuração do cluster (postgresql.conf,pg_hba.conf, and pg_ident.conf) estão normalmente guardados na PGDATA (a partir do PostgreSQL 8.0 é possível guardá-los em qualquer lado).

Item	Description
PG_VERSION	A file containing the major version number of PostgreSQL
base	Subdirectory containing per-database subdirectories
global	Subdirectory containing cluster-wide tables, such as pg_database
pg_clog	Subdirectory containing transaction commit status data
pg_multixact	Subdirectory containing multitransaction status data (used for shared row locks)
pg_notify	Subdirectory containing LISTEN/NOTIFY status data
pg_serial	Subdirectory containing information about committed serializable transactions
pg_snapshots	Subdirectory containing exported snapshots
pg_stat_tmp	Subdirectory containing temporary files for the statistics subsystem
pg_subtrans	Subdirectory containing subtransaction status data
pg_tblspc	Subdirectory containing symbolic links to tablespaces
pg_twophase	Subdirectory containing state files for prepared transactions
pg_xlog	Subdirectory containing WAL (Write Ahead Log) files
postmaster.opts	A file recording the command-line options the server was last started with
postmaster.pid	A lock file recording the current postmaster process ID (PID), cluster data directory path, postmaster start timestamp, port number, Unix-domain socket directory path (empty on Windows), first valid listen_address (IP address or *, or empty if not listening on TCP), and shared memory segment ID (this file is not present after server shutdown)

Os nomes das subdirectorias da directoria base são criados com o OID da base de dados. Estas directorias contêm os ficheiros da base de dados incluindo os catálogos do sistema. Estes últimos são particularmente importantes porque guardam informações sobre as tabelas e os atributos.

Cada tabela e índice estão guardados num ficheiro distinto em disco. A estes está ainda associado um free space map e um visibility map. No free space map que são guardadas as informações sobre o espaço livre na relação. Já no visibility map é mantida informação sobre as páginas que contêm tuplos visíveis a todas transacções activas.

Tanto as tabelas quanto os índices estão limitados ao tamanho máximo de 1GB, e quando ultrapassam esse tamanho é ultrapasso estes são divididos em segmentos de 1 GB (valor por defeito, podendo ser mudado nas configurações através de –with –segsize). O primeiro segment tem o mesmo nome que o fiheiro; os segmentos seguintes são numerados sequencialmente (por exemplo:nomeFicheiro.1, nomeFicheiro.2, etc.) Esta técnica é a usada para ultrapassar problemas em casos onde

haja problemas a nível de tamanho. O free space map e o visibility map também podem requerer múltiplos segmentos, mas isso é algo pouco provável na prática.

A utilização de tablespaces (localizações especificas no sistema de ficheiros) complica a gestão de ficheiros, o directório pg_tblspc guarda links simbólicos para as localizações reais dos ficheiros. A nomenclatura desses ficheiros baseia-se no OID dos tablespaces. Os tablespaces pg_global e pg_default não podem ser acedidos através de pg_tblspc, mas podem ser acedidos através do directório global e base, respectivamente. O comando pg_relation_filepath()devolve o caminho relativo a PGDATA para uma relação.

Os ficheiros temporários (usados para operações de ordenação de dados para além dos quem cabem em memória) são criados de maneira a identificar quem os criou. O nome destes ficheiros têm o formato pgsql_tmpPPP.NNN onde PPP é o PID do dono do ficheiro e NNN distingue vários ficheiros temporários do mesmo dono.

3.3. Mecanismo de Partições

O PostgreSQL suporta partição básica de tabelas. Esta secção descreve o como e o porquê de se implementar partição no design da base de dados.

A partição refere-se à capacidade de divisão de uma tabela excessivamente grande noutras de menor dimensão. A partição pode trazer vários benefícios:

- O desempenho das perguntas pode ser drasticamente improvisado em certas situações, particularmente quando os tuplos mais acedidos da tabela estão numa única partição ou num pequeno número de partições. A partição também permite reduzir o tamanho dos índices e tornar mais provável que as partes mais usadas dos índices caibam em memória.
- Quando as consultas ou actualizações acedem a uma grande percentagem de uma única partição, a performance pode ser melhorada tirando-se partido da pesquisa sequencial dessa partição ao invés de usar um índice e leituras aleatórias espalhadas por toda a tabela.
- As operações de carregamento e deleção podem ser obtidas através da adição e da remoção de partições, caso esse requerimento esteja planeado aquando

do design da partição Consegue-se evitar o overhead provocado pela rotina VACUUM durante um delete.

- Os dados raramente usados podem ser migrados para meios de armazenamento mais baratos e mais lentos.

Os benefícios do particionamento são apenas compensatórios quando uma tabela seria, não particionada, de dimensões muito elevadas. O ponto exacto em que a tabela beneficia da partição está dependente da aplicação em questão, apesar de uma regra usada ser que o tamanho da tabela deve exceder o tamanho da memória física no servidor da base de dados.

Actualmente, o PostgreSQL suporta particionamento através do mecanismo de herança de tabelas. Cada partição deve ser criada como uma tabela "filha" de uma única tabela "mãe". Essa tabela "mãe" está normalmente vazia e existe apenas para representar todos os dados.

As seguintes formas de particionamento podem ser implementadas no PostgreSQL:

Range Partioning: a tabela está entre um intervalo de valores definidos pela coluna-chave ou por um conjunto de colunas, sem que haja sobre posicionamento nesse intervalo de valores associados às diferentes partições. Por exemplo, é possível migrar migrações através de intervalos de datas, ou por intervalos de identificadores para certos objectos de negócios.

List Partioning: a tabela está particionada através da listagem explícita dos valores-chave que aparecem em cada partição.

3.4. Organização e clustering

No PostgreSQL os tuplos são guardados em heap, onde esses registos podem ser de tamanho variável, algo que é conseguido através de um mecanismo de slotted-page. Numa organização em heap as tabelas possuem apenas índices não clustered. Os tuplos não estão guardados por nenhuma ordem particular enão existe qualquer sequência na paginação.

Uma operação de clustering consiste em reordenar uma tabela com base num índice. Este conceito é suportado pelo PostgreSQL através do comando CLUSTER. Para executar este comando é necessário especificar a tabela que se pretende reordenar, assim como o respectivo índice. O índice a utilizar na operação de clustering terá que ter sido previamente definido na tabela. Clustering é uma operação que só pode ser executada uma vez: quando uma tabela é posteriormente actualizada, as modificações não são clustered, ou seja, os tuplos novos ou actualizados não guardados de acordo com a ordem do índice. Por exemplo:

```
CLUSTER cursos USING unique_name;
```

Depois de uma tabela ter sido clustered, o utilizador poderá efectuar nova operação sem se referenciar ao índice a utilizar, referindo-se apenas á tabela, que esta será reordenada segundo o índice previamente definido. Por exemplo:

```
CLUSTER cursos;
```

Usar o comando CLUSTER sem a existência de qualquer parâmetro reorganiza todas as tabelas anteriormente clustered na corrente base de dados cujo actual utilizador possui, ou todas as tabelas se usado por um super-utilizador. Esta forma de cluster não pode ser executada dentro de um bloco de transacção:

```
CLUSTER ;
```

O PostgreSQL não suporta multi-table clustering.

3.5. Paginação

Todas as tabelas e índices, normalmente demasiado grandes para caber em apenas uma página, são guardados num array como um array de páginas de tamanho fixo (normalmente 8KB, se bem que se pode definir um tamanho diferente aquando da compilação do servidor). Numa tabela, todas as páginas são equivalentes a nível lógico, portanto um tuplo particular pode ser guardado em qualquer página. Por outro lado, nos índices, a primeira página é geralmente reservada para os metadados que contêm a informação de controlo, podendo ainda existir diferentes páginas dentro do índice, dependendo do tipo de acesso deste.

Uma página possui as seguintes secções:

Item	Description
PageHeaderData	24 bytes long. Contains general information about the page, including free space pointers.
ItemIdData	Array of (offset,length) pairs pointing to the actual items. 4 bytes per item.
Free space	The unallocated space. New item pointers are allocated from the start of this area, new items from the end.
Items	The actual items themselves.
Special space	Index access method specific data. Different methods store different data. Empty in ordinary tables.

As linhas da tabela são todas estruturadas da mesma forma. Possuem um cabeçalho fixo de 23 bytes, um bitmap opcional, um identificador do objecto e os dados do utilizador. Os dados são armazenados a partir do deslocamento t_hoff, que está armazenado no header. Quando o bitmap está presente, este, possui um bit para cada coluna. Os valores do bitmap indicam se a coluna é null ou não.

Na representação de tabelas, o PostgreSQL não permite que um tuplo se divida por mais do que uma página, impedindo assim guardar directamente campos de grande tamanho. Para contornar este problema o PostgreSQL recorre a uma técnica denominada TOAST (TheOversized-Attribute Storage Technique), que ocorre de forma transparente ao utilizador. Apenas alguns tipos de dados suportam TOAST, uma vez que não é necessário impor o overhead a tipos de dados que não consegue produzir campos de grande tamanho.

4. Indexação e hashing

Esta secção visa abordar a temática da indexação, que é um mecanismo comum usado para melhorar a performance da base de dados.

O PostgreSQL permite a criação/ eliminação de índices, que permitem que o servidor da base de dados encontre e retorne tuplos específicos muito mais rapidamente do que na ausência do mesmo. Por outro lado, é preciso ter cuidado com a utilização excessiva dos mesmos, pois estes causam overhead ao sistema da base de dados, o que piora o seu desempenho.

Exemplo de uma tabela sem índice na coluna nome:

```
create table departamentos (cod_departamento numeric(3) not null,
nome varchar(40) not null,
primary key (cod_departamento));
```

Ao ser efectuada uma consulta do tipo:

```
SELECT nome FROM departamentos WHERE nome=”DI”;
```

O sistema teria de percorrer todos os tuplos desta tabela, sequencialmente, até encontrar a entrada com o id pretendido. De modo a resolver este problema procede-se à criação de um índice para a chave primária, pois esta consulta está dependente do campo id. O comando genérico para criar índices é o seguinte:

CREATE INDEX—define a new index

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

Este código representa a criação de um índice com nome name, na tabela table, do tipo method, no atributo column ou na expressão expression. Por defeito, no

PostgreSQL, os índices criados são árvores B+. Um exemplo de um índice muito simples para o exemplo anterior:

CREATE INDEX unique_name ON departamentos (nome);

De modo análogo, para se remover um índice, o comando a usar é o DROP INDEX. Usando ainda o exemplo anterior:

DROP INDEX unique_name;

Uma vez criados, os índices não necessitam de qualquer intervenção adicional uma vez que o sistema automaticamente os actualiza quando a tabela é modificada e os utiliza em consultas quando determina que o uso destes as tornam mais eficientes. Para que o sistema seja capaz de avaliar correctamente se é mais eficiente ou não usar o índice em questão, este usa estatísticas que lhe permitem tomar decisões “fundamentadas”. Logo, convém correr comando ANALYZE para actualizar as referidas estatísticas. Para além de aumentar a performance nas situações acima referidas, os índices podem ainda melhora-la nas instruções de UPDATE, DELETE que usem condições de procura ou no caso de procuras para junções.

No entanto, a criação de índices também tem desvantagens: a criação destes em grandes tabelas poderá ser um processo moroso. Por defeito, o PostgreSQL permite que sejam efectuadas leituras (operação SELECT) em paralelo a uma criação de índice. No entanto, todas as operações de escrita (INSERT, UPDATE, DELETE) são bloqueadas até o processo de indexação terminar. Em alguns ambientes isto não é aceitável, portanto, é possível permitir algo semelhante ao caso de leituras, sendo que há uma série de cuidados a ter.

A juntar a isto, após um índice ser criado, o sistema terá de o manter sincronizado com a tabela, o que implica algum overhead em operações de manipulação de dados. Consequentemente, índices que tenham pouca (ou nenhuma) utilização devem ser removidos.

4.1. *Tipos de Índices*

O PostgreSQL disponibiliza quatro tipos diferentes de estruturas para indexação: B-tree, Hash, GiST e GIN. Cada um dos tipos usa um algoritmo diferente, mais apropriado para diferentes tipos de consultas. Por defeito, o comando CREATE INDEX cria índices B-tree, que se adequam à maioria das situações.

B-tree

Tal como já foi referido anteriormente, o comando CREATE INDEX cria por defeito um índice B-tree. Uma B-tree trata-se de uma estrutura de dados que representa dadosordenados de modo a providenciar inserção, remoção e consulta de informação de forma eficiente. O B refere-se a "Balançado" e a ideia é que a quantidade de dados em ambos os lados da árvore é mais ou menos a mesma e cada caminho da raiz até às folhas tem sempre o mesmo comprimento. Os índices B-tree também são muito eficientes com caching, mesmo que esta seja apenas parcial. Este tipo de índices pode ser usado para tratar eficientemente de perguntas de igualdades e de intervalos de valor. Por isso, o optimizador de consultas do PostgreSQL considera o uso de índices B-tree quando as consultas são feitas com recurso a condições que envolvem os operadores aritméticos:<, <=,=, >= e >. Construções equivalentes a combinações destes operadores, como BETWEEN e IN, e até condições como IS NULL, IS NOT NULL, LIKE, entre outras.

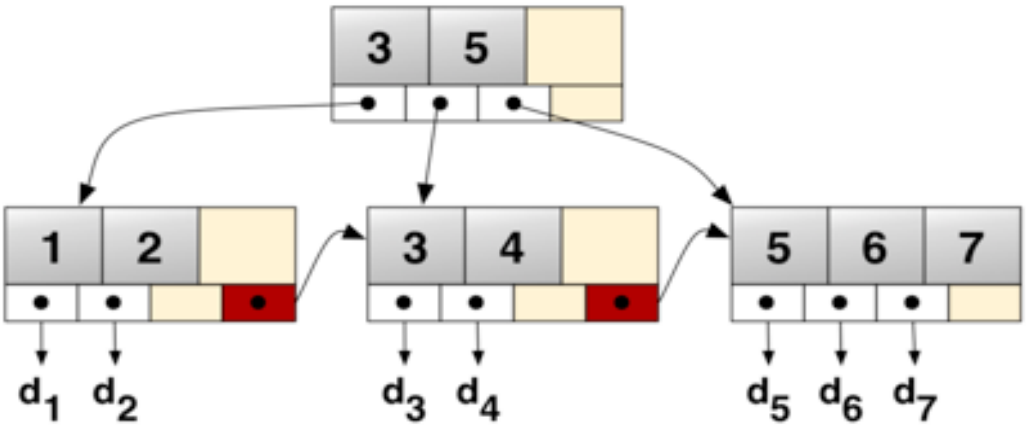


Figura 1 - Esquema de uma B-Tree

Os nós internos contêm chaves e apontadores para os filhos e podem ter entre (n-1)/2 e n-1 chaves e entre n/2 e n filhos (excepto a raiz), em que n é o número máximo de apontadores na árvore. Assim, a altura da árvore é muito pequena relativamente ao número de nós, facilitando a pesquisa de uma palavra no índice.

Exemplo de criação de um índice B-tree (explícito e não por defeito):

CREATE INDEX unique_name ON departamentos USING btree(nome);

Hash

Os índices hash conseguem lidar apenas com simples comparações de igualdade. O optimizador de perguntas considerará usar um índice hash quando está envolvida uma coluna indexada numa comparação com o operador "=".Actualmente os índices hash não são WAL-logged (Write-Ahead Logging), pelo que poderá ser necessário que estes necessitem de ser reconstruidos através do REINDEX caso a base de dados crache e haja mudanças por escrever. Por estas razões, o uso deste tipo de índices é presentemente desencorajado.

Para especificar o uso da indexação de Hash é usada a opção hash na cláusula USING, por exemplo:

CREATE INDEX unique_name ON departamentos USING hash(nome);

GiST (Generalized Search Tree)

Os indices GiST não representam um tipo específico de estruturas de dados, mas sim uma abstracção/interface que pode ser utilizada para implementar vários tipos de indexação em árvore. B-trees, R-trees e muitos outros tipos de indexação em árvore podem ser implementados em GiST. Uma vantagem do GiST é que permite o desenvolvimento de tipos de dados personalizados com os métodos de acesso apropriados, por um perito no domínio dos dados, em vez de um perito em bases de dados. Os operadores particulares de cada índice GiST podem variar consoante a estratégia de indexação (a operator class). Por exemplo, a distribuição standard do PostgreSQL inclui classes de operadores GiST para diversos tipos de dados geométricos e bidimensionais, que suportam consultas usando os seguintes operadores: <<,&<,&>,>>, <<|, &<|,|&>, |>>, @>, <@, ~= e &&. A grande vantagem deste tipo de índices é a sua versatilidade, ou seja, quanto ao que é possível fazer com estes. Por consequência, sendo as GIST estruturas abstractas, as mesmas exigem algum trabalho quanto à sua definição e utilização,

Para especificar o uso da indexação de GiSTé usada a opção gistna cláusula USING, por exemplo:

```
CREATE INDEX unique_name ON departamentos USING gist(nome);
```

GIN (Generalized Inverted Index)

A Generalized Inverted Index é uma estrutura de indexação invertida generalizada que armazena listas sem repetições de pares (key, posting list), onde posting list é um conjunto de tuplos onde a chave ocorre. Cada tuplo pode aparecer em vários posting lists, pois cada valor indexado pode conter mais que uma chave. Cada chave é guardada apenas uma vez, portanto um índice GIN é muito compacto para casos em que a mesma chave aparece muitas vezes.

O termo Generalized, aplica-se no mesmo sentido que as GiST, pois permite a utilização de várias técnicas de indexação, dependentemente dos operadores utilizados. Por exemplo, a distribuição standard do PostgreSQL inclui classes de

operadores GIN para arrays unidimensionais, que suportam perguntas indexadas usando os seguintes operadores: <@, @>, =, &&. Para além disso, o código de métodos de acesso do GIN não precisam de saber as operações específicas que aceleram. Em vez disso são usadas estratégias customizadas definidas por tipos de dados específicos. A estratégia deverá definir como é que as chaves são extraídas dos items índices e das condições das perguntas, e como se determina se um tuplo que contem algumas das chaves da pergunta satisfaz, de facto, a pergunta.

Uma vantagem dos índices GIN é que permite o desenvolvimento de tipos de dados personalizados com os métodos de acesso apropriados, por um perito no domínio dos dados, em vez de um perito em bases de dados, ou seja, é basicamente a mesmo vantagem dos índices GiST.

Por outro lado, os índices GIN não suportam pesquisas completas dos índices, sendo então impossível garantir que uma pesquisa de um dado índice devolva todos os tuplos de uma tabela. Outra limitação algo grave é o facto dos índices GIN ignorarem todas as keys com valor NULL.

Para especificar o uso da indexação de GIN é usada a opção gin na cláusula USING, por exemplo:

```
CREATE INDEX unique_name ON departamentos USING gin(nome);
```

Comparação entre os índices GIN e GiST

Existem diferenças substanciais de performance entre estas duas técnicas de indexação, logo no momento de decisão entre uma ou outra, deve-se ter em conta as seguintes características:

- Procuras usando índices GIN são à volta de três vezes mais rápidas que usando índices GiST.

- Os índices GIN demoram três vezes mais tempo a serem construídos do que os índices GiST.

•Os índices GIN são moderadamente mais lentos a actualizar que os índices GiST. No entanto, se o suporte a actualização rápida estiver desligado, a actualização torna-se dez vezes mais lenta.

•Os índices GIN são duas a três vezes maiores que os índices GiST.

Pode-se então concluir que os índices GIN são melhores para dados estáticos pois o processo de consulta é mais rápido. Para dados dinâmicos, os índices GiST são mais rápidos a actualizar. Especificamente, estes índices são muito bons para dados dinâmicos e rápidos se o número de palavras únicas for menor que 100,000 e por outro lado os índices GIN irão trabalhar melhor com um número mais elevado de palavras únicas mas são mais lentos a actualizar.

4.2. Índices multi-coluna

O PostgreSQL suporta índices multi-coluna, ou seja, índices que podem ser definidos em mais que uma coluna da tabela. Por exemplo, numa tabela da seguinte forma:

```
create table cadeiras(cod_cadeira numeric(5) not null,
        nome varchar(40) not null,
        creditos numeric(2) not null,
        cod_departamento numeric(3) not null,
        primary key (cod_cadeira),
        foreign key (cod_departamento) references departamentos);
```

Caso perguntas do seguinte tipo sejam frequentemente efectuadas:

```
SELECT * FROM cadeiras WHERE creditos = 6 AND cod_departamento  = 2;
```

poderá ser necessário definir um índice para as colunas principais e secundárias em conjunto, por exemplo:

```
CREATE INDEX unique_credit_dep ON cadeiras(creditos, cod_departamento);
```

Actualmente, apenas os índices B-tree, GiST e GIN suportam índices multi-coluna, podendo um índice deste género abranger até 32 colunas (limite que pode ser alterado aquando da construção do PostgreSQL) e pode ser usado com perguntas que envolvem qualquer subconjunto das colunas indexadas.

Um índice B-tree multi-coluna pode ser utilizado com condições de consulta que envolvam qualquer subconjunto de colunas do índice, mas este é mais eficiente quando há restrições sobre as duas colunas principais (mais à esquerda). Tendo um índice cadeiras (nome, creditos, cod_departamento) como exemplo, quando existe uma restrição de igualdade em nome e uma restrição de desigualdade em creditos a pesquisa utilizando o índice será limitada até ao ponto limitado por essas restrições. Adicionalmente, se houver uma restrição sobre cod_departamento, uma vez que esta coluna está à direita das duas colunas principais, essa restrição será verificada no próprio índice, de modo a evitar consultas à tabela, mas não será reduzida a parte do índice que terá de ser verificada. Este índice pode ainda ser usado para perguntas que não apliquem restrições em nome, mas como este índice é maioritariamente ordenado por nome o índice terá que ser todo ele consultado, portanto na maior parte dos casos o optimizador de consultas deverá escolher uma tabela sequencial em vez de um índice.

No caso dos índices GiST multi-coluna, a eficiência está grandemente relacionada com a quantidade de valores distintos na primeira coluna: caso existam poucos, a eficiência sera baixa mesmo que haja muitos valores distintos nas restantes colunas. Para além disso, a restrição na primeira coluna é mais importante que qualquer outra restrição nas restantes colunas para determinar a parte do índice a ser consultada.

Ao contrário dos índices B-tree e GiST multi-coluna, a eficiência da consulta usando índices GIN multi-coluna não depende das colunas indexadas.

Em qualquer caso, cada coluna deve ser usada com operadores adequados ao tipo do índice. Para além disso, os índices multi-coluna devem ser usados moderadamente pois, na maior das situações, um índice de coluna única costuma ser suficiente para responder às perguntas que possam ser feitas, poupando-se espaço e tempo.

4.3. Combinação de múltiplos índices

A combinação de múltiplos índices surgiu porque há duas grandes limitações no uso de índices simples:

- Apenas cláusulas de consultas que usam as colunas do índice com operadores das suas classes de operadores podem ser usadas.

- Apenas cláusulas conjuntas com AND podem ser usadas: uma consulta como WHERE a = 5 OR b = 6 não pode usar o índice directamente.

Para ultrapassar estas limitações, o PostgreSQL fornece o uso de múltiplos índices combinados, incluindo múltiplos usos do mesmo índice. No exemplo acima, a combinação de múltiplos índices seria realizada operando por cima do mesmo índice duas vezes (uma vez para cada coluna) e retornando a junção OR de ambos os resultados. Em detalhe, isto envolve consultas sobre os índices necessários e a criação, para cada índice, de um bitmap contendo a localização de cada tuplo coincidente com as condições do índice e então juntando os bitmaps adquiridos (um para cada índice) através de operações OR ou AND, dependendo que é necessário em função da consulta. Uma vez que os bitmaps são adequadamente juntos, os tuplos das tabelas representados são visitados em ordem física, o que significa que qualquer ordem dos índices já definida é perdida e consequentemente qualquer cláusula ORDER BY na consulta necessitará de uma ordenação à parte para fazer efeito. Por esta razão, tendo em conta a consultas extras aos índices que poderão eventualmente ser necessários, pode acontecer que o optimizador escolha o uso de um índice simples apesar da existência de índices adicionais. Esta solução pode ser superior à indexação multi-coluna, mas cada uma das situações merece uma análise cuidada antes de se decidir usar ou índices multi-coluna ou confiar na combinação de índices múltiplos do PostgreSQL.

4.4. Índices únicos e índices parciais

O PostgreSQL permite a criação de índices únicos e de índices parciais: um índice único é usado para forçar a unicidade do valor de uma coluna ou a unicidade de valores de índices multi-coluna. Apenas os índices B-tree podem ser declarados como sendo únicos. No caso dos índices multi-coluna, apenas os tuplos onde todas as colunas indexadas são iguais são rejeitados. Este tipo de índices são automaticamente criados sempre que se coloca a restrição UNIQUE sobre um atributo da tabela ou quando a tabela tem uma chave primária definida, portanto a criação explícita deste tipo de índices pode levar a uma situação de índices duplicados na tabela. Exemplo da criação de um índice único:

```
CREATE UNIQUE INDEX unique_local ON alunos(local);
```

Já os índices parciais são usados para aplicar um índice num subconjunto de uma coluna, indexando apenas a parte mais relevante da tabela, sendo que esse subconjunto é definido por uma expressão condicional (pelo predicado do índice parcial): os índices contem entradas apenas para os tuplos que satisfazem esse predicado.

Uma grande razão de se usar um índice parcial é para evitar indexar valores comuns na tabela: como uma consulta que pesquisa esses valores comuns não utilizará o índice, não há razão em manter esses tuplos indexados Isto permite que se reduza o tamanho do índice, o que provocará um aumento na rapidez dessas consultas que não usam o índice. Para além disso também aumentará a rapidez de muitas das operações de actualização da tabela, pois o índice não precisa de ser actualizado em todos os casos. Exemplo da criação de um índice parcial:

```
CREATE INDEX aluno_miguel ON alunos(nome) WHERE nome = 'Miguel';
```

4.5. *Organização de ficheiros usando índices*

Por defeito, no PostgreSQL, a organização das tabelas em ficheiro é feita em heap, sem clustering envolvido. Todavia, às vezes é preferível, do ponto de vista de maximizar a performance na obtenção das páginas do disco, organizar as tabelas segundo uma estrutura de indexação. Para tal, o PostgreSQL, permite a utilização do comando CLUSTER, usado da seguinte forma, por exemplo:

CLUSTER cursos USING unique_name; (em que unique_name é um índice já existente).

Depois de reorganizada a tabela, o sistema sabe qual é o índice a usar para organizar a tabela em ficheiro daqui por diante, sendo necessário usar apenas a cláusula:

```
CLUSTER cursos;
```

Podendo ainda ser atribuído um novo índice de organização através da cláusula ALTER TABLE.

Dado este comando é criada uma cópia temporária dos dados (quer da tabela quer do índice) para fazer a operação, sendo necessário haver espaço livre no disco equivalente ou superior ao espaço ocupado por ambos.

Por outro, importa realçar que este comando é apenas executado uma vez. Qualquer alteração futura na tabela será, mais uma vez, organizada em ficheiro, seguindo a estrutura por defeito, heap. Consequentemente, caso seja importante manter a estrutura do ficheiro segundo um certo tipo de índice, deve ser executado o comando de forma periódica. Para além disso, ao se alterar o parâmetro FILLFACTOR da respectiva tabela, para valores menores que 100%, pode ajudar a prevenir a ordenação do cluster durante actualizações, dado que os tuplos actualizados são mantidos na mesma página desde que exista espaço livre suficiente.

Por fim, quando uma tabela é reorganizada segundo um índice, um LOCK de acesso exclusivo é obtido. Isto previne que outras operações possam ser executadas sobre a tabela, enquanto a reorganização está em curso.

4.6. Estruturas temporariamente inconsistentes

No PostgreSQL é possível que os tipos de índices se tornem inconsistentes temporariamente, até ao final da transacção, através da cláusula DEFERRED (aplicada a restrições) nos comandos CREATE e ALTER TABLE e caso a restrição esteja INITIALLY DEFERRED, só é analisada no final da transacção. A análise da restrição pode ser alterada pelo comando SET CONSTRAINTS. Constituindo este facto uma necessidade para algumas operações no âmbito das transacções, este tópico será abordado mais à frente, no capítulo 5.

5. Processamento e optimização de perguntas

5.1. Introdução

A **optimização de interrogações ou perguntas** é o processo de escolha de entre as várias estratégias existentes, pela qual há uma selecção do plano mais eficiente, para execução de uma dada interrogação. O PostgreSQL assume por omissão que os utilizadores do sistema de gestão de base de dados não escrevem as interrogações na sua forma mais eficiente, logo havendo necessidade de ter um componente específico para optimização de perguntas.

O PostgreSQL quando recebe uma nova pergunta procede a sua análise através de três fases distintas: Análise léxica, sintáctica e semântica.

Na análise léxica é feita a leitura dos caracteres que constituem a pergunta, em palavras, e o reconhecimento das mesmas (Ex.: cláusula SELECT, nome da coluna, nome de uma função, operador de comparação, etc.)

Na fase de análise sintáctica é verificado se as palavras formam uma frase sintacticamente correcta, ou seja, o SBD verifica se a frase está correctamente compostas segundo as regras sintácticas da linguagem. Durante a análise sintáctica é construída uma representação interna da pergunta através de um modelo em árvore.

Após a garantia que a representação interna está sintáctica e semanticamente correcta durante a construção da árvore de interrogação, várias transformações são aplicadas antes de a pergunta ser processada pelo sistema de execução.

Designa-se por reescrita da interrogação a primeira fase de transformação, na qual o PostgreSQL usa um sistema de regras. As regras podem ser definidas por os utilizadores cuja activação é despoletada pelas operações: UPDATE, DELETE, INSERT e SELECT. As vistas encontram-se implementadas através de regras SELECT.

As regras UPDATE, DELETE, INSERT são em primeiro lugar geridas pelo SBD e só depois as regras SELECT. As operações de escrita podem conter operações SELECT embebidas.

Após a reescrita de uma pergunta, novas regras podem ser despoletadas e o sistema processa interactivamente o conjunto de regras até mais nenhuma ser aplicável.

O PostgreSQL por omissão não possui regras que permitam optimizar a pergunta nesta fase. As regras são criadas, explicitamente, pelo utilizador e, implicitamente, através da definição de uma nova vista, não sendo usadas para fins de optimização.

No fim da fase de reescrita, a pergunta é sujeita a fase de optimização e planeamento. É gerado o plano de execução parcial respectivo para cada bloco isoladamente, sendo a ordem de geração de planos iniciados nos blocos interiores da interrogação e termina nos blocos exteriores. O optimizador do PostgreSQL baseia-se na estimativa do custo de execução para determinar o melhor plano.

O modelo de custo pondera o custo das operações de I/O e o custo de processamento em amontoados e em índices, sendo os melhores planos caracterizados por um custo menor. Existe dois algoritmos para geração do plano óptimo:

Algoritmo de programação dinâmica, usado por omissão.

Algoritmo genético, utilizado sempre que o número de tabelas referenciadas numa operação SQL seja elevado.

O algoritmo de programação dinâmica na primeira fase determina o plano de acesso as tabelas. Na segunda fase determina as melhores estratégias para juntar grupos de duas tabelas do bloco da pergunta. Em seguida são encontradas as melhores estratégias para juntar grupos de três tabelas, tendo em conta os melhores planos determinados anteriormente na junção das duas tabelas, e assim o processo continua até que seja encontrado um plano óptimo para executar o actual bloco de interrogação.

Quando o número de tabelas referenciadas pelo bloco é muito elevado, o algoritmo de programação dinâmica torna-se dispendioso. Neste caso utiliza-se o algoritmo genético visto aparentar ser superior ao algoritmo de programação dinâmica quando o numero de tabelas da cláusula FROM e muito elevado.

Como resultado da fase de planeamento e optimização obtêm-se o plano óptimo de execução da interrogação.

5.2. *Representação interna das perguntas*

Internamente o PostgreSQL representa as interrogações através de um modelo em árvore. A árvore contém os seguintes nós principais:

- Tipo de comando (operação SQL utilizada pelo utilizador, que contêm os seguintes valores: UPDATE, INSERT, DELETE e SELECT).
- Lista das tabelas (lista de tabelas utilizada na pergunta, ou seja, contêm todas as tabelas da clausula FROM).
- Tabela dos resultados (Usado apenas nas operações de escrita e representa a tabela onde vai ser colocado o resultado).
- Lista das colunas (Representa o resultado da interrogação, através de uma lista de expressões. Na operação SELECT o nó contêm as colunas nas quais deve ser feita a projecção. Na operação de INSERT representa as expressões presentes na clausula VALUES ou na cláusula SELECT quando é utilizada uma interrogação para obter os registos a adicionar a tabela. Na operação UPDATE contêm as expressões usadas para obter os novos valores dos atributos, envolvendo constantes ou atributos da tabela do resultado. No caso da operação DELETE este nó não é utilizado, uma vez que não são produzidos novos registos).

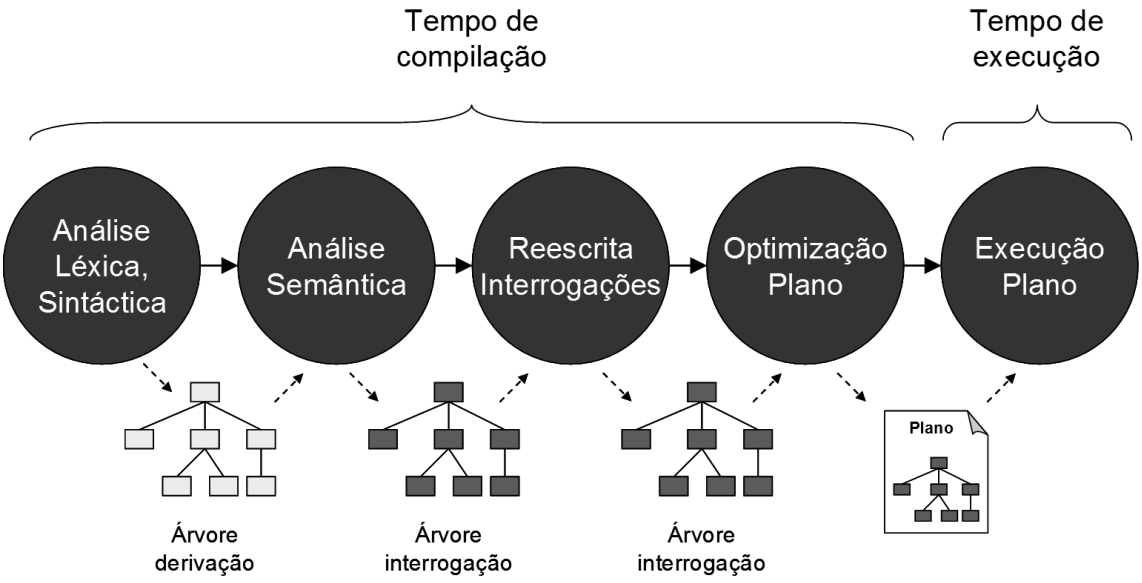


Figura 2 – Sequência de actividades de Processamento de Interrogações no PostgreSQL

- Expressão de selecção (Representa o conteúdo da clausula WHERE).

- Subárvore de junção (Contêm a lista das tabelas da cláusula FROM. No entanto, quando são usadas expressões JOIN, particularmente OUTER JOIN, é necessário representar a ordem da junção explicitamente, de acordo com as restrições impostas).

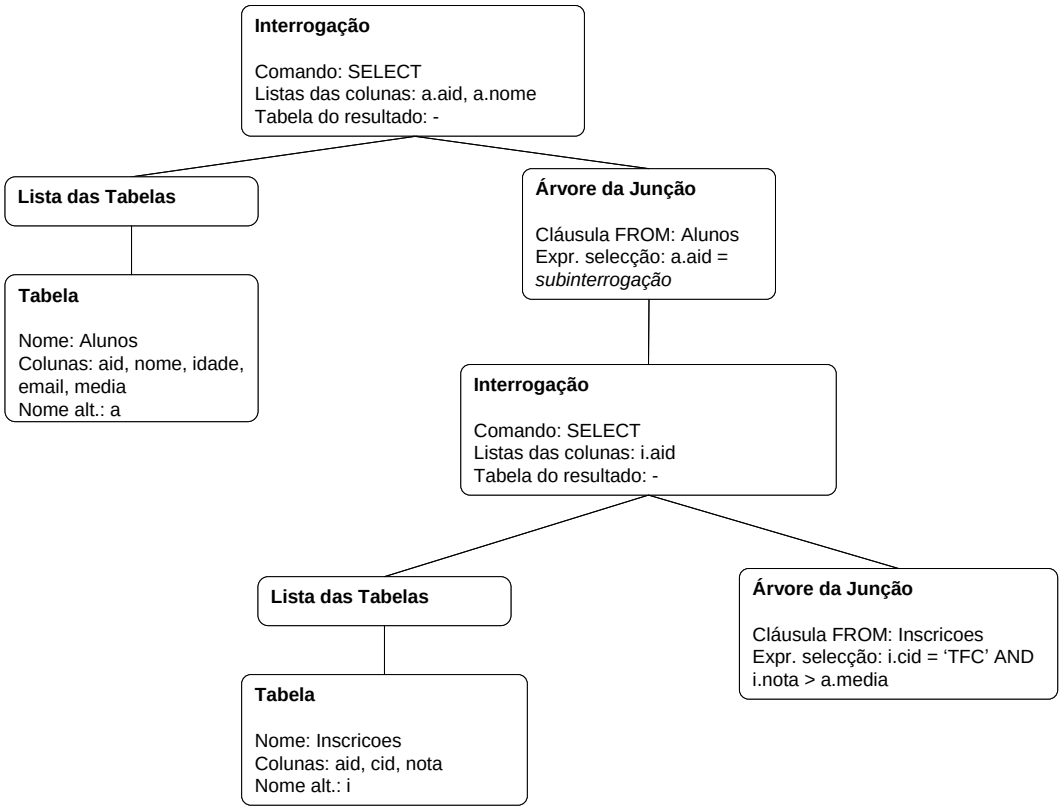
O processo de optimização de interrogações é apresentado através de um exemplo utilizando o seguinte esquema relacional:

alunos(aid:string, nome:string, idade:integer, email:string,media:real)
inscricoes(aid:string, cid:string, nota:integer)
aidFK alunos

Considera-se que o utilizador executa uma pergunta para determinar quais os alunos cuja nota obtida no trabalho final do curso é superior a sua média:

```
SELECT a.aid, a.nome
FROM alunos a
WHERE a.aid IN
    ( SELECT i.aid
    FROM inscricoes i
WHERE i.nota > a.media AND i.cid = 'TFC' )
```

Na figura seguinte pode-se visualizar a representação interna da pergunta.



5.3. Reescrita de perguntas

É permitido ao utilizador definir regras de reescrita de perguntas no servidor de base de dados. Ao contrário dos triggers e dos procedimentos armazenados, o sistema de regras actua entre a fase de análise semântica e o planeamento, modificando a pergunta original de acordo com um conjunto de regras. Após a transformação, a pergunta modificada é submetida ao planeador.

As vistas estão implementadas sobre um sistema de regras. Definição de vista:

CREATE VIEW vista AS SELECT * FROM tabela;

O SGBD converte automaticamente na sequência de comandos SQL:

CREATE TABLE vista (lista de colunas da tabela)

CREATE RULE "_RETURN" AS ON SELECT TO vista DO INSTEAD

SELECT * FROM tabela;

O significado de equivalência entre vistas e tabelas pode-se traduzir em: a informação presente nos catálogos da SGBD é idêntica para as duas entidades de dados e as vistas e as tabelas podem ser tratadas da mesma forma pelo analisador sintáctico.

Mesmo que os comandos usados sejam: DELETE, UPDATE e INSERT, As regras SELECT são sempre aplicadas como o ultimo passo na transformação das interrogações. A operação de SELECT ao contrário das regras das operações de escrita, modificam a árvore de interrogação localmente em vez de criarem uma nova.

Considera-se a seguinte vista:

```
CREATE VIEW v_inscricoes (aid, nome, cid) AS
SELECT a.aid, a.nome, i.cid
FROM alunos a, inscricoes i
WHERE a.aid = i.aid;
```

O PostgreSQL converte o comando anterior correspondente a definição de vista, em dois comandos equivalentes:

```
CREATE TABLE v_inscricoes (aid, nome, cid);
CREATE RULE _RETURN AS ON SELECT TO v_inscricoes DO INSTEAD
SELECT a.aid, a.nome, i.cid
FROM alunos a, inscricoes i
WHERE a.aid = i.aid;
```

A regra de reescrita é despoletada ao efectuar a operação SELECT:

```
SELECT * FROM v_inscricoes;
```

O modelo em árvore, após a leitura do analisador sintáctico, é equivalente á seguinte representação:

```
SELECT v_inscricoes.aid, v_inscricoes.nome, v_inscricoes.cid
FROM v_inscricoes v_inscricoes;
```

Na fase seguinte, a interrogação é transformada pelo sistema de regras. Para cada tabela da cláusula FROM, o SGBD verifica se existe alguma regra aplicável. Neste caso

existe a regra '_RETURN', criada automaticamente como consequência da definição da vista 'v_inscricoes':

```
SELECT a.aid, a.nome, i.cid
FROM alunos a, inscricoes i
WHERE a.aid = i.aid;
```

O sistema de reescrita cria uma sub-interrogação na cláusula FROM, de modo a fazer uma expansão, contendo a operação SELECT definida na regra. A árvore correspondente á pergunta transformada é equivalente ao seguinte código:

```
SELECT v_inscricoes.aid, v_inscricoes.nome, v_inscricoes.cid
FROM (SELECT a.aid, a.nome, i.cid
FROM alunos a, inscricoes i
WHERE a.aid = i.aid) v_inscricoes;
```

O PostgreSQL verifica se há mais alguma regra aplicável as tabelas da cláusula FROM da pergunta de topo. Não havendo mais tabelas, a verificação continua para a interrogação interior, terminando quando todas as tabelas tiverem sido analisadas. Tanto a tabela 'alunos' como a 'inscricoes' não possuem quaisquer regras de reescrita, significando que a fase de reescrita está concluída.

5.4. Optimização e planeamento

Existem 3 passos a dividir a tarefa de planeamento:

- Transformação da interrogação
- Análise
- Geração de planos

5.4.1. Transformação de interrogações

O objectivo é converter perguntas aninhadas em interrogações simples. A pergunta pode ser inicialmente escrita pelo utilizador (exemplo da secção 2) ou ser gerada pelo SGBD durante o processamento de vistas (exemplo da secção 3). A pergunta da secção 3:

```
SELECT v_inscricoes.aid, v_inscricoes.nome, v_inscricoes.cid
FROM (SELECT a.aid, a.nome, i.cid
FROM alunos a, inscricoes i
WHERE a.aid = i.aid) v_inscricoes;
```

É convertida na seguinte interrogação simples:

```
SELECT alunos.aid, alunos.nome, inscricoes.cid
FROM alunos, inscricoes
WHERE alunos.aid = inscricoes.aid;
```

O número de ordens de junções possíveis é limitado devido a existência de interrogações aninhadas. Considere-se a tabela "bolsas (aid:string, montante:real)", que guarda informação sobre os alunos que têm bolsa. Podemos realizar a seguinte pergunta se pretendermos saber sobre os alunos que têm bolsa.

```
SELECT v_inscricoes.aid, v_inscricoes.nome, v_inscricoes.cid FROM v_inscricoes, bolsas
WHERE v_inscricoes.aid = bolsas.aid;
```

Após a expansão da vista obtém-se a seguinte pergunta:

```
SELECT v_inscricoes.aid, v_inscricoes.nome, v_inscricoes.cid
FROM (SELECT a.aid, a.nome, i.cid
FROM alunos a, inscricoes i
WHERE a.aid = i.aid) v_inscricoes, bolsas;
WHERE v_inscricoes.aid = bolsas.aid;
```

Após a conversão, passa a ser possível considerar ordens de junção que envolvam as três tabelas:

```
SELECT alunos.aid, alunos.nome, inscricoes.cid
FROM alunos, inscricoes, bolsas
```

```
WHERE inscricoes.aid = alunos.aid
AND inscricoes.aid = bolsas.aid;
```

5.4.2. Análise de perguntas

São realizadas, essencialmente, duas operações de optimização:

- Dedução da igualdade implícita
- Aplicação antecipada dos predicados

Considere-se o exemplo anterior:

```
SELECT alunos.aid, alunos.nome, inscricoes.cid
FROM alunos, inscricoes, bolsas
WHERE inscricoes.aid = alunos.aid
AND inscricoes.aid = bolsas.aid;
```

A dedução de igualdade implícita pode ser determinada tendo como base as duas condições de igualdade: inscricoes.aid = alunos.aideinscricoes.aid = bolsas.aid. É possível deduzir uma terceira igualdade implicita: alunos.aid = bolsas.aid. A terceira igualdade permite juntar primeiro as tabelas 'Alunos' e 'Bolsas'. Esta opção é particularmente vantajosa caso as tabelas sejas pequenas em relação a tabela 'inscrições'.

A aplicação antecipada dos predicados permite que as expressões de selecção sejam avaliadas tão cedo quanto possível. Diz-se que esta regra de optimização representa uma heurística uma vez que tipicamente, mas nem sempre, reduz o custo de execução do plano.

5.4.3. Geração de Planos

A geração de planos de execução é feita após o trabalho preparatório das fases de transformação. Existem dois algoritmos para a geração do plano óptimo:

- Algoritmo de programação dinâmica
- Algoritmo genético

Nesta secção apenas é explicado o funcionamento do algoritmo de programação dinâmica.

O algoritmo de programação dinâmica representa um dos métodos tradicionais de optimização de perguntas. No PostgreSQL, apresentado na figura 3, gera apenas planos de junção recursivos à esquerda (left deep). Designa-se por **aridade** de um plano, o número de tabelas envolvidas na junção. Para um plano recursivo à esquerda de aridade 'n', o subplano esquerdo representa, sempre, um plano de junção de aridade '(n-1)' e o direito um plano para aceder aos dados de uma tabela.

Na primeira fase, linhas 1 a 4, são gerados planos de acesso às tabelas referenciadas pela interrogação. A função 'accessPlans' permite definir as estratégias de acesso, dentro das alternativas possíveis: acesso sequencial ou através de um índice sobre um subconjunto dos atributos de uma tabela. Os planos óptimos são armazenados em 'optPlan'.

Entrada: Interrogação q sobre as tabelas $\{R_1, \ldots, R_n\}$
Saída: Um plano de execução para q

1: **for** $i = 1$ **to** n **do** {
2: $optPlan(\{R_i\}) = accessPlans(R_i)$
3: $prunePlans(optPlan(\{R_i\}))$
4: }
5: **for** $i = 2$ **to** n **do** {
6: **for all** $S \subseteq \{R_1, \ldots, R_n\}$ **such that** $|S| = i$ **do** {
7: $optPlan(S) = \emptyset$
8: **for all** R_j, S_j **such that** $S = \{R_j\} \cup S_j$ **do** {
9: $optPlan(S) = optPlan(S) \cup joinPlans(optPlan(S_j), optPlan(\{R_j\}))$
10: $prunePlans(optPlan(S))$
11: }
12: }
13: }
14: $finalizePlans(optPlan(\{R_1, \ldots, R_n\}))$
15: $prunePlans(optPlan(\{R_1, \ldots, R_n\}))$
16: **return** $optPlan(\{R_1, \ldots, R_n\})$

A segunda fase de processamento, linhas 5 a 13, serve para encontrar a melhor ordem de junção das tabelas. Na primeira iteração são determinados os planos de junção entre duas tabelas, com base nos planos de acesso aos dados encontrados na primeira

fase. Seguidamente são determinados os planos de junção entre grupos de três tabelas, com base nos planos de junção 2-ários e de acesso aos dados. A seguir são gerados os planos 4-ários com base nos planos 3-ários, 2-ários e de acesso aos dados. O passo iterativo termina quando forem gerados todos planos óptimos de aridade 2 até 'n', permitindo juntar a totalidade das tabelas da interrogação. Note-se que em cada iteração é usada a mesma função 'joinPlans' para gerar planos de junção cada vez mais complexos, através de planos mais simples. Tal como é possível existirem vários planos de acesso, também podem existir métodos de junção alternativos, por exemplo: nested-loops-join, sort-merge-join e hash-join. A função 'joinPlans' devolve um plano alternativo por cada método de junção aplicável.

Na terceira e última fase, os planos 'n'-ários são processados pela função 'finalizePlans', linha 14, que adiciona os operadores relacionais necessários para completar o plano, por exemplo: projecção, ordenação ou agrupamento (group by). Um dos aspectos mais apelativos deste algoritmo é a sua capacidade de rejeitar planos sub-óptimos no final de cada passo. A tarefa de rejeição de planos é desempenhada pela função 'prunePlans', executada nas linhas 3, 10 e 15 do algoritmo. Por exemplo, durante a geração dos planos 2-ários, o algoritmo, ao considerar os planos 'A⋈B' e 'B⋈A', retém apenas a alternativa de menor custo, armazenando-a em 'optPlan({A, B})'. Isto só é possível devido à propriedade da comutatividade do operador '⋈', significando que ambas as alternativas produzem o mesmo resultado. Durante a construção dos planos 3-ários, 4-ários, …, 'n'-ários, apenas é considerada a alternativa de menor custo para juntar 'A' e 'B'.

Se o plano 'A⋈B' for menos dispendioso que o plano 'B⋈A', então qualquer plano completo que englobe 'A⋈B' é menos dispendioso que qualquer outro que englobe 'B⋈A'. Por exemplo, 'C⋈(A⋈B)' é mais eficiente que 'C⋈(B⋈A)'. Tanto o algoritmo de programação dinâmica como o da procura exaustiva analisam a totalidade dos planos possíveis, isto é 'n!' planos recursivos à esquerda. No entanto, devido ao mecanismo de rejeição dos planos sub-óptimos, o primeiro método consegue ser muito mais eficiente.

O algoritmo apresentado anteriormente representa uma versão simplificada da realidade. Uma vez que o algoritmo merge-join necessita que as duas tabelas de entrada estejam ordenadas segundo os atributos da junção, não basta guardar o plano de menor custo no final de cada iteração. Por este motivo, torna-se necessário guardar o melhor plano para cada **ordenação interessante**. Designa-se por ordenação interessante uma seriação dos registos que facilite a aplicação ulterior do algoritmo merge-join, facilite a ordenação dos registos segundo a cláusula ORDER BY ou o agrupamento segundo a cláusula GROUP BY.

5.5. Algoritmos de junção

No PostgreSQL é implementado três algoritmos para a junção de tabelas:

- Nested-loops-join
- Sort-merge-join
- Hash-join

No algoritmo nested-loops-join considere-se a junção téta entre duas tabelas: '$A \bowtie_{\theta} B$'. A tabela interior B é percorrida, integralmente para cada registo da tabela exterior 'A'. Para cada par de registos par de registos das duas relações, '$r_A.r_B$', é verificada a satisfação da condição 'θ'. Se a condição for satisfeita, o registo composto é adicionado ao resultado. Tal como o algoritmo de pesquisa linear, o nested-loops-join não depende da existência de índices e pode ser usado independentemente da condição de selecção. Uma vez que a relação interior é lida sempre que um novo registo é acedido, torna-se conveniente que seja a mais pequena das duas tabelas. Caso a relação interior caiba totalmente em memória, necessita ser lida apenas uma vez. Se existir um índice sobre a relação interior, então as pesquisas lineares podem ser substituídas por pesquisas em índices. Para cada registo exterior, o índice é usado para encontrar os registos interiores que satisfaçam a condição de selecção.

O algoritmo sort-merge-join pode ser usado para executar junções naturais e de igualdade. Considere-se a junção natural entre duas tabelas: '$A \bowtie B$', onde o conjunto '$A \cap B$' denota os atributos em comum. Se as tabelas já estiverem ordenadas segundo os atributos comuns, então o processo de junção assemelha-se à fase merge do algoritmo merge-sort. Por exemplo, se existir um índice primário sobre os atributos da

junção natural pertencentes a 'A', não é necessário proceder à ordenação destes registos. Caso não estejam ordenadas, torna-se necessário materializá-las e proceder a sua ordenação, usando o algoritmo merge-sort.

Tal como o sort-merge-join, o algoritmo hash-join é usado para implementar junções naturais e de igualdade, sendo uma boa estratégia de execução caso não existam índices sobre as duas tabelas. Neste algoritmo, é construído um índice hash em memória para a tabela interior, a menor das duas tabelas. Seguidamente, a tabela exterior é percorrida. Para cada registo exterior, os valores dos atributos de junção são usados para pesquisar o índice hash previamente criado.

5.6. Modelo de Custo

O PostgreSQL estima o custo de execução de cada plano gerado por forma a determinar o plano óptimo. O modelo de custo usado contempla:

- Custo das operações de I/O.
- Custo de processamento dos registos em amontoados e índices.

A fórmula de custo é definida da seguinte forma:

$$C = P + W \cdot R$$

Em que:

C - Custo total

P - Numero de páginas transferidas que corresponde ao custo I/O

R - Numero de registos examinados, que corresponde ao custo de processamento peloCPU

W – Peso relativo entre o custo de I/O e o custo de processamento.

No acesso sequencial, todas as páginas e registos de uma tabela são processados. Sempre que um índice é usado para aceder a uma tabela, é estimada a **selectividade** da interrogação. Designa-se por selectividade, a fracção dos registos que obedecem à condição de selecção da interrogação. Por definição, os registos dos índices primários

seguem a mesma ordem dos registos no ficheiro de dados, podendo existir apenas um índice deste tipo por tabela. O PostgreSQL implementa estes índices no próprio ficheiro de dados. Neste caso, estima-se que o custo E/S seja igual ao produto do número de páginas da tabela e da selectividade da interrogação. Similarmente, estima-se que o custo de processamento seja igual ao produto do número de registos da tabela e da selectividade. Os registos nos índices secundários estão ordenados de forma diferente dos registos da tabela, podendo haver mais do que um índice deste tipo por tabela. Nos acessos através de índices secundários, é feita uma pesquisa prévia no índice para determinar que registos da tabela base são necessários analisar. Neste caso, estima-se que o custo E/S seja igual à soma do número de páginas da tabela e o número de registos do índice, multiplicada pelo factor de selectividade. Similarmente, estima-se que o custo de processamento seja dado pela soma do número de registos na tabela e número de registos no índice, multiplicada pelo factor de selectividade.

Custo dos Acessos a Tabelas		
Custo / Acesso	P	R
Sequencial	NumPaginas	NumRegistos
Índice primário	NumPaginas*F	NumRegistos*F
Índice secundário	F*(NumPaginas + IRegistos)	F*(NumRegistos + IRegistos)

NumPaginas: número de páginas de uma tabela

NumRegistos: número de registos de uma tabela

IRegistos: número de registos no índice

F: factor de selectividade combinado para todas as condições de selecção

Tabela 1 – Estimativas de Custo para os Acessos a Tabelas

As fórmulas de estimativa de custo para as estratégias de junção são função do tamanho, em termos de páginas e registos, das tabelas interior e exterior. Para estimar

o tamanho de uma tabela, o SGBD multiplica o tamanho original da tabela pelo factor de selectividade associado aos predicados. Se alguma cláusula puder ser avaliada através de um índice, o factor de selectividade é calculado conforme explicado anteriormente. Se a "tabela exterior" for produto de uma junção, então o factor de selectividade diz respeito a essa operação de junção. O factor de selectividade de uma junção representa a fracção dos registos, pertencentes ao produto cartesiano das tabelas envolvidas, que se espera satisfaçam a condição de junção. A informação estatística usada na estimativa de custo é actualizada, periodicamente, pelo SGBD e, manualmente, sempre que se execute a operação ANALYZE.

Custo das Junções	
Nested Loops Join	$C_{ext} + N_{ext} * C_{int}$
Sort Merge Join	$C_{ext} + C_{ord\_ext} + C_{int} + C_{ord\_int}$
Hash Join	$C_{ext} + C_{const\_hash} + N_{ext} * C_{hash}$

onde:

C_{ext}: custo para aceder à tabela exterior

C_{int}: custo para aceder à tabela interior

C_{ord_ext}: custo para ordenar a tabela exterior (igual a zero, caso já esteja ordenada)

C_{ord_int}: custo para ordenar a tabela interior (igual a zero, caso já esteja ordenada)

C_{const_hash}: custo para construir o índice *hash* sobre a tabela interior

C_{hash}: custo de uma pesquisa através do índice *hash*

N_{ext}: tamanho da tabela exterior

Tabela 3 – Estimativa do Custo das Junções

6. Gestão de transacções e controlo de concorrência

NO SGBD estudado, internamente a consistência dos dados é garantida pelo modelo Multiversion Concurrency Control (MVCC). Este modelo permite, que cada transacção veja um snapshot (uma versão consistente da base de dados) como era antes de a transacção dar inicio ou da pergunta, de acordo com o nível de isolamento que esta seleccionado no momento. É garantido desta forma que a transacção não veja dados inconsistentes causados por transacções concorrentes, garantindo igualmente o isolamento. O MVCC, minimiza ao máximo o uso de locks, reduzindo assim o número de deadlocks, porque quando há leituras de dados estes nunca entram em conflito com escritas ou leituras.

Locks de tabelas ou de linhas também estão disponíveis pelo uso de explicit locking para quando se pretende gerir as próprias transacções, porém o uso do protocolo MVCC tem melhor desempenho comparado com o uso de explicit locking.

6.1. Isolamento

O SQL standart define 4 níveis de isolamento o **read uncommitted**, **read committed**, **repeatable read** e **serializable** para prevenir 3 fenómenos nas transacções concorrentes, sendo eles o dirty read que lê dados de uma transacção concorrente aos quais não foram commit, nonrepeatable read acontece quando é feita duas leitura seguidas e os dados foram alterados por uma transacção concorrente que fez commit de uma leitura para a outra e por último phantom read quando executa de novo uma pergunta, retornado um conjunto de linhas que satisfazem uma condição e descobre que esse conjunto foi alterado por uma outra transacção que fez commit.

Na tabela seguinte são demonstrados os quatro níveis de isolamento e os seus comportamentos.

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

No Postgresql podemos utilizar os quatro níveis de isolamento, mas internamente é implementado três níveis mais propriamente o read commited, repeatable read e serializable, quando é seleccionado o read uncommitted o que obtemos é o read committed, isto apenas permite saber qual o fenómeno que não acontece, não permitindo assim saber qual fenómeno que acontece. A razão pela qual só são

implementados três níveis é por ser a única maneira de aplicar o isolamento original para o protocolo MVCC.

O PostgreSQL usa por omissão o read committed, mas pode ser alterado com o seguinte comando o nível de isolamento:

```
SET TRANSACTION transaction_mode[, ...]

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


where transaction_mode is one of:


    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }

READ WRITE | READ ONLY
```

6.1.1. Read Committed

Por omissão como já foi referido anteriormente este é o nível utilizado pelo SGBD estudado, quando há um select sem a cláusula FOR UPDATE/SHARE apenas se vê dados que fizeram commit antes do inicio da pergunta, vendo um snapshot da base de dados no momento que foi feita a pergunta, dois selects seguidos podem retornar resultados diferentes desde que uma outra transacção concorrente tenha feito commit dos dados da pergunta feita, nunca vê dados que foram uncommitted por transacções concorrentes apenas os da sua transacção.

Os comandos como UPDATE, DELETE, SELECT FOR UPDATE e SELECT FOR SHARE é semelhante ao SELECT, no aspecto de verem só os dados que fizeram commit antes do início da pergunta, podendo um tuplo ser alterado, apagado ou obter um lock por uma outra transacção concorrente, devendo nesse caso se tiver sido apagado não fazer nada e continuar normalmente ou se tiver sido alterado, altera-se novamente verificando em ambas as condições a cláusula where novamente.

Em resumo este nível é óptimo para comandos que não envolvam muita complexidade nas suas condições, sendo óptimo para transacções pequenas como a que é mostrada em baixo mesmo que os dados sejam alterados sempre é alterado a ultima versão.

```
BEGIN;
UPDATE cadeiras SET creditos = creditos + 1 WHERE cod_cadeira = 1;
COMMIT;
```

Nota: No exemplo estamos a assumir que existe um tuplo com cod_cadeira = 1 na nossa base de dados.

6.1.2. Repeatable Read

Neste nível de isolamento só se vê dados inseridos antes do início da transacção, nunca os dados que foram commit depois do início da transacção, com excepção para os dados alterados pela sua própria transacção, sendo que duas leituras seguidas vão ser retornar os mesmos tuplos.

Os comandos UPDATE, DELETE, SELECT FOR UPDATE, e SELECT FOR SHARE, tem um comportamento semelhante ao SELECT, no entanto se um tuplo esta a ser alterado ou apagado por uma transacção concorrente, fica-se a aguardar que este faça rollback o que permite à nossa transacção continuar ou no caso commit a situação exige o rollback, mostrando uma mensagem de erro, ERROR: could not serialize access due to concurrent update, porque os tuplos que foram modificados ou apagados não podem ser alterados, devendo neste caso abortar e começar de novo, com um snapshot actualizado da base de dados.

Este nível tem uma garantia que a transacção vê uma versão consistente da base de dados, contudo pode haver um número elevado de rollbacks, porque outras transacções concorrentes alteram os dados, situação que pode ser resolvida com o uso de explicit locking.

6.1.3. Serializable

O nível de isolamento é idêntico ao anterior com a excepção de as transacções são executadas "uma após a outra", em série. Neste nível podem ocorrer falhas por causadas por outras transacções concorrentes.

De seguida vamos ilustrar um exemplo de duas transacções no nível de serializable:

```
class | value
-------+-------
    1 |    10
    1 |    20
    2 |   100
    2 |   200
```

Uma transacção A vai executar a seguinte pergunta, introduzindo o valor 30 retornado da soma com class 2:

```
SELECT SUM(value) FROM tabela WHERE class = 1;
```

E outra transação B que introduz na class 1 o valor 300 da soma:

```
SELECT SUM(value) FROM tabela WHERE class = 2;
```

Se A fizer commit antes de B o resultado de B já deveria ser 330 como se tivessem sido executadas a A e depois a B, a transacção B é abortada e aparece a seguinte mensagem de erro. No nível repeatable read terminariam ambas sem erros.

```
ERROR: could not serialize access due to read/write dependencies among
transactions
```

Para garantir a serialização o SBGD usa-se predicate locking, são usados para identificar dependências em transacções serializáveis, sem causar deadlocks.

6.2. Explicit Locking

Nesta secção serão demonstradas os vários modos de locks para o controlo concorrente aos dados nas tabelas, que serão usados para quando o comportamento do MVCC não seja o mais adequado, assegurando que os dados das tabelas não sejam apagados/alterados no decorrer das transacções concorrentes.

De seguida serão apresentados os vários níveis de locks das tabelas que serão usados automaticamente pelo PostgreSQL, ou podendo também ser usados com o comando:

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

where***lockmode*** is one of:

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

Por exemplo, qualquer outra transacção concorrente que tente fazer uma leitura dos dados da tabela departamentos, fica bloqueado ate que a transacção que fez o lock faça o commit.

Begin;

LOCK TABLE departamentos IN ACCESS EXCLUSIVE MODE;

select * from departamentos;

Commit;

Os locks apenas podem entrar em conflito quando se referem a mesma tabela e em transacções diferentes, nunca na mesma transacção mesmo que se obtenha vários tipos de locks na mesma tabela.

6.2.1. Table-level lock modes:

ACCESS SHARE – Usado para o comando SELECT para consultas a tabelas, sem que seja preciso fazer modificações.

ROW SHARE – Os comandos SELECT FOR UPDATE/SELCT FOR SHARE obtêm este lock para os tuplos da tabela.

ROW EXCLUSIVE – É usado nos comandos UPDATE, DELETE e INSERT, obtêm este lock novamente para os tuplos da tabela, mais propriamente este modo é adquirido por comandos que façam alterações na tabela.

SHARE UPDATE EXCLUSIVE – Adquirido pelo VACCUM (sem FULL), ANALYZE, e CREAT INDEX CONCURRENTLY.

SHARE – Adquirido pelo comando CREATE INDEX (sem CONCURRENTLY)

SHARE ROW EXCLUSIVE – Este modo não é adquirido por nenhum comando automaticamente, como vai ser estudado a seguir.

EXCLUSIVE – Este modo não é adquirido automaticamente nas tabelas, por nenhum comando do PostgreSQL.

ACCESS EXCLUSIVE- Adquirido pelos comandos ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER e VACUUM FULL, este é modo por omissão para o comando LOCK TABLE que não especifica o modo de lock.

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

Na tabela anterior encontra-se os vários tipos de locks que entram em conflito com os de outras transacções.

6.2.2. Row-Level Locks

Como foi verificado para as tabelas também existem locks de tuplos que podem ser exclusive ou shared. O primeiro é atribuído sempre que se faz uma alteração ou apaga-se um tuplo sendo mantido até um rollback ou commit como nos locks de tabelas. Os locks nos tuplos não afectam as leituras apenas boqueiam alterações para o mesmo tuplo.

Para adquirir exclusive lock num tuplo, faz-se um SELECT FOR UPDATE, pudendo ser alterados de seguida sem conflitos.

Para o shared lock utiliza-se o comando select for share, não bloqueando outras transacções concorrentes de obterem um shared lock, contudo nenhuma transacção pode alterar tuplos nem adquirir exclusive locks, ficando sempre bloqueados até se libertar os shared locks.

6.2.3. Deadlocks

O uso de explicit locking aumenta a ocorrência de deadlocks, quando duas ou mais transacções pedem locks no que uma outra tem, como no exemplo a seguir. O PostgreSQL detecta automaticamente os deadlocks, abortando de imediato uma transacção e continuando a outra.

Considere o caso em que existe na tabela departamentos os tuplos (1, ‘DI’) e (2, ‘DM’) e se tenta alterar a tabela, uma transacção A obtém um lock sobre o tuplo com cod_departamento = 1.

```
Begin;

UPDATE departamentos SET nome = ‘Departamento Informatica’ WHERE cod_departamento = 1;
```

De seguida uma outra transacção concorrente B obtém um lock sobre o cod_cadeira = 2 e fica bloqueado quando tenta alterar para o cod_cadeira = 1.

```
Begin;

UPDATE departamentos SET nome = 'Departamento Mat' WHERE cod_departamento = 2;

UPDATE departamentos SET nome = 'Departamento Inf' WHERE cod_departamento = 1;

Commit;
```

Enquanto a transacção A tenta alterar o tuplo com cod_cadeira = 2 ficando também bloqueada.

```
UPDATE departamentos SET nome = 'Departamento Matematica' WHERE cod_departamento = 2;

Commit;
```

Para evitar a ocorrência deadlocks não é aconselhável haver transacções concorrentes que obtém locks na ordem inversa uma da outra, ou em operações que durem muito tempo.

6.2.4. Advisory Locks

O PostgreSQL tem meios para criar locks nas aplicações, são chamados advisory locks porque o sistema não força o seu uso, é a aplicação que o deve usar bem. São uteis para o locking quando não se adequa o MVCC, sendo a sua maior diferença que são usados durante uma seção e não como foi visto anteriormente durante uma transacção.

Exemplo de um uso de advisory lock:

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- ok
```

6.3. Consistência

Quando não é possível fazer no nível de serializable as transacções, para garantir a consistência deve-se usar os comandos SELECT FOR UPDATE, SELECT FOR SHARE, ou um explicit locking da tabela.

Por exemplo um banco que deseja verificar se a soma dos créditos é igual aos depósitos de outra tabela, quando ambas as tabelas como seria de esperar de um banco têm muitas transacções concorrentes. O resultado de dois select sum nunca iria ser o mesmo no nível de read commit, para este exemplo o melhor seria um explicit locking das duas tabelas para os dados serem consistentes.

6.4. Locking and Indexes

Os vários métodos para se acederem aos índices implementados pelo SBGD estudado serão explicados de seguida:

B-tree e GiST – É usado o lock para acessos de leitura/escrita, sendo libertados imediatamente apos ser feita a procura ou inserção. Este tipo de índice proporciona uma maior concorrência sem elevados deadlocks.

Hash – O lock é libertado apos o bucket ser processado, proporcionando uma melhor concorrência comparativamente aos index-level do anterior, mas ocorrem deadlocks dado que as operações são mais longas que as de índices.

GIN – O lock é libertado imediatamente apos o índice ser encontrado ou inserido, mas realçando que a inserção produz mas index key por linha, sendo substancialmente trabalhoso para uma única inserção, apesar de semelhante ao primeiro.

Em resumo os índices B-tree oferecem um melhor desempenho para transacções concorrentes, uma vez que têm mais recursos do que os índices hash, sendo recomendados para aplicações que precisam de índex scalar data, quando não se trata de non-scalar data os índices Gin devem ser usados.

6.5. SAVEPOINT

Um SAVEPOINT é uma instrução que ocorre numa transacção que indica um "ponto" e a partir desse ponto até se encontrar a instrução ROLLBACK TO SAVEPOINT, a instrução anula tudo o que tinha feito desde o início do savepoint, se encontrar um RELEASE SAVEPOINT continua a sua execução normalmente.

Por exemplo na seguinte instrução, só serão introduzidos os departamentos DI e DF.

```
BEGIN;
   INSERT INTO departamentos VALUES (1, 'DI');
   SAVEPOINT savepoint;
   INSERT INTO departamentos VALUES (2, 'DM');
   ROLLBACK TO SAVEPOINT savepoint;
   INSERT INTO departamentos VALUES (3, 'DF');
COMMIT;
```

Mas neste exemplo já serão introduzidos todos os tuplos.

```
BEGIN;
   INSERT INTO departamentos VALUES (1, 'DI');
   SAVEPOINT savepoint;
   INSERT INTO departamentos VALUES (1, 'DM');
RELEASE SAVEPOINT savepoint;
COMMIT;
```

7. Suporte para bases de dados distribuídas

No SGBD estudado não existe bases de dados distribuídas, no entanto a partir da versão de 2008, já se pode fazer a replicação de dados, apesar de ser só uma copia e exigir que os dados alterados sejam depois inseridos no servidor, levando a conflitos pelos tuplos que podem ter sido alterados por mais do que uma pessoa,

Existem aplicações gratuitas para as bases de dados distribuídas como o slony, PgCluster.

Como as redes tiveram uma enorme evolução nos últimos anos, havendo cada vez menos falhas o uso de bases de dados distribuídas, na nossa opinião tendera a ser cada vez menos, como pelos protocolos de acesso concorrente e alteração de dados não serem os mais eficazes ou até mesmo a ideia da base de dados ser centralizada.

8. Outras Características do PostgreSQL

8.1. *Suporte XML*

O PostgreSQL suporta o tipo dados XML e tem agregado um conjunto de funções responsáveis pela sua manipulação.

Para produzir valores do tipo xml a partir de caracteres, utiliza-se a função xmlparse:

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

Exemplos:

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

Embora esta seja a única maneira de converter cadeias de caracteres em valores de XML de acordo com o padrão SQL, a sintaxe específica do PostgreSQL é a seguinte:

```
xml '<foo>bar</foo>'
'<foo>bar</foo>'::xml
```

O tipo xml não suporta validações dos dados com (DTD), mesmo quando os valores especificam um tipo de declaração (DTD). Não existe actualmente um suporte built-in para validar esquemas de linguaguens XML.

A operação inversa, de produzir strings de valores XML, utiliza a função xmlserialize:

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

type pode ser character, character varying, ou text. Novamente, de acordo com o standard SQL, esta é a única forma de converter o tipo xml em tipos de caracteres. PostgreSQL também permite que seja realizado o cast dos valores.

Quando é realizado um cast numa string de caracteres sem utilizar as funções XMLPARSE or XMLSERIALIZE, respectivamente, a escolha de DOCUMENT versus CONTENT é determinado pelo parâmetro de configuração de sessão XML OPTION, na qual pode ser afectado utilizando o seguinte comando por omissão:

```
SET XML OPTION { DOCUMENT | CONTENT };
```

Ou utilizando a sintaxe do PostgreSQL.

```
SET xmloption TO { DOCUMENT | CONTENT };
```

Se o valor por defeito é CONTENT, então todas as formas de dados XML são permitidos.

8.2. Linguagens Procedimentais

As várias linguagens procedimentais incluídas com a distribuição do PostgreSQL são as seguintes:

- PL / pgSQL
- PL / Tcl
- PL / Perl
- PL / Python.

Além disso, há um certo número de línguas procedimentais que são desenvolvidas e mantidas fora do núcleo da distribuição PostgreSQL. Exemplo de algumas:

- PL/Java – Java
- PL/PHP – PHP
- PL/Py – Python
- PL/Ruby – Ruby

8.3. Segurança

O PostgreSQL é um SGBD que tem como ponto forte a segurança, sendo em muitos casos o factor determinante na escolha entre os SGBDs Open Source.

Um dos objectivos primordiais da equipa de desenvolvimento do PostgreSQL, foi oferecer um SGBD robusto e fiável, sendo recomendado a sua utilização em aplicações complexas, que exijam o controlo de grandes volumes de dados, ou em casos de tratamento de informações críticas.

O controlo de acesso aos dados deve ser feito de acordo com a necessidade de cada sistema. Existem informações que não devem ser vistas por todos os utilizadores, como exemplo, uma tabela que contenha os salários dos funcionários, ou os dados de um projecto novo que esteja em fase de desenvolvimento. O PostgreSQL oferece um controle baseado em direitos de acesso ou privilégios e é através deles que um utilizador pode ou não aceder aos dados. A identidade do utilizador é que vai determinar o conjunto de privilégios disponíveis.

A autenticação dos clientes é feita através de um arquivo localizado na pasta de dados do cluster, chamado pg_hba.conf. Este ficheiro é composto de vários registros, onde cada registo determina o tipo de conexão e contêm valores como:

- O nome da base de dados
- A banda de endereços de IP de cliente
- Método de autenticação a ser utilizado nas conexões que correspondem a estes parâmetros

Com a utilização deste arquivo que armazena os registos comos parâmetros necessários às autenticações, é possível, por exemplo, determinar se uma base de dados possa ser acedida por um determinado utilizador ou conjunto de utilizadores, registado no ficheiro. É possível também especificar os endereços IP ou a banda de endereços IP que o registro corresponde, autorizando o acesso apenas a partir desses endereços.

O PostgreSQL dispõe métodos de autenticação, dentre os quais é possível destacar:

- Por palavra-passe onde o principal método é o md5 (Message-Digest Algorithm5) que suporta palavras passe encriptadas. As senhas, encriptadas ou não,ficam armazenadas numa tabela do catálogo do sistema chamada pg_user.
- Kerberos implementa um protocolo de transporte de dados em rede, assegurando a comunicação dos dados mesmo numa rede insegura.

Além destes, a autenticação pode ser feita utilizando o nome do utilizador e senha do sistema operativo, porém este método não é recomendado pois a segurança torna-se dependente da oferecida pelo sistema operativo.

A tabela do catálogo, pg_user, pode ser visualizada apesar de conter as passe dos utilizadores. Desta forma o campo palavra-passe é sempre mostrado com asteriscos ao invés dos caracteres reais da senha.

Para aumentar a segurança dos dados que navegam na rede, é possível encriptar as comunicações entre o cliente e o servidor através do suporte nativo conexões SSL (Secure Sockets Layer) através da instalação do OpenSSL no cliente e no servidor. Ainda é possível criar as conexões de clientes através de túneis SSH (SecureShell).

Ao tratar da segurança oferecida pelo PostgreSQL um dos principais aspectos é o mecanismo para criação de cópias de segurança e seu restabelecimento em caso de falhas.

O PostgreSQL possui duas formas de backup que contemplam as principais necessidades deste tipo de processo. São elas:

- O dump é um recurso que armazena diversos comandos em um ficheiro de texto, que podem reconstruir os dados da base de dados através de comandos SQL de criação de tabelas e todos os outros comandos necessários. Pode ser feito o dump de apenas alguns registros da base de dados ou de todos os registros contidos no cluster, o chamado dumpall.
- O point-in-time-recovery é capaz de reconfigurar os dados para a data e hora informada pelo DBA (Data Base Administrator). Este recurso pode ser feito pela utilização do WAL como um sistema de armazenamento de ficheiros.

Além dos recursos descritos anteriormente, pode ser feito backup através do sistema operativo, copiando o cluster dos dados. Todavia, esse recurso trás algumas restrições: o software deve estar fechado no momento da cópia do cluster; não é possível escolher as tabelas ou registros a serem copiados, isso quer dizer todos os dados serão copiados, inclusive arquivos de controle do próprio banco que não se relacionam com as tabelas e registros; o processo é muito mais demorado e garante muito menos a integridade dos arquivos do que os anteriores.

A cópia de segurança poit-in-time recovery somente é possível devido à existência do WAL. O WAL foi criado com o intuito de evitar perda de dados numa eventual queda

do sistema, pois trabalha com pontos de verificação que são gravados sempre que uma modificação é feita.

9. Conclusões

A principal vantagem do PosgreSQL em relação ao Oracle deve-se ao facto de ser uma SGBD open source. O PostgreSQL é livre de pagamentos e tem a possibilidade de ser personalizado a medida do cliente, o que não acontece nos pagos com o código fechado.

O PostgreSQL adopta dois tipos de índices (GIST e GIN), que não se encontram presentes no sistema Oracle. Qualquer utilizador do PostgreSQL através do índice GIST tem a liberdade de implementar a estrutura de árvore que desejar, coisa que não acontece no Oracle.

Ambos os sistemas oferecem em termos de código SQL, um conjunto extremamente completo de recursos, permitindo consultas com uma grande complexidade. O PostgreSQL ao contrário do Oracle, no caso de complexidade elevada, pode tornar-se mais confuso devido a sintaxe dos seus comandos.

Comparativamente ao capítulo 4, o isolamento o Oracle implementa mais níveis do que o PostgreSQL, sendo que na parte propriamente dita da concorrência têm comportamentos muito semelhantes embora os implementem de maneiras diferentes como é o caso do MVCC utilizado no SGBD estudado.

Um dos pontos negativos do PostgreSQL é não haver suporte para as bases de dados distribuídas, havendo necessidade de instalar programas a parte como o Slony entre outros oferecidos.

10. Bibliografia

- http://www.postgresql.org/docs/9.3/static/index.html