Estudo sobre o sistema PostgreSQL

Relatório

01-06-2014

Grupo 30
39665,Emídio Dias
41744, Mira Amaidas
41593, Soraia Freitas

Tabela de conteúdos

1 Introdução

O presente relatório trata-se do estudo do Sistema de Gestão de Base Dados orientado a objetos, PostgreSQL. O objetivo deste relatório é estudar as funcionalidades, vantagens e desvantagens deste sistema. Para esta análise será utilizado o sistema Oracle 11g como objeto de comparação.

O PostgreSQL foi desenvolvido pela Universidade da Califórnia em Berkeley, no departamento de ciências da computação. Foi o resultado de uma ampla evolução que se iniciou no projeto Ingres (INteractive Graphics REtrieval System), liderado pelo Michael Stonebraker, um dos pioneiros de bases de dados relacionais, o projeto fora patrocinado por Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), e ESL, Inc.

1.1 Cronologia

1985	Stonebraker começou o projeto pós-Ingres com o objetivo de resolver problemas com o modelo de base dados relacional. O projeto resultante, chamado **Postgres**, era orientado a introduzir a menor quantidade possível de funcionalidades para completar o suporte a tipos
1986	A equipa divulgou uma série de documentos descrevendo a base do sistema.
1988	O projeto possuía um protótipo funcional.
1989	A versão 1 foi liberada para um grupo pequeno de usuários.
1990	Versão 2 com um sistema de regras reescrito.
1991	Para a versão 3, foram adicionados suporte para múltiplos gerenciadores de armazenamento e um melhorado motor de consultas.
1993	Após a liberação da versão 4, a qual era uma simples versão de limpeza, o projeto foi oficialmente abandonado pela Universidade de Berkeley.

1994	Dois estudantes, Andrew Yu e Jolly Chen, adicionaram um interpretador SQL para substituir a linguagem QUEL (desenvolvida para o Ingres) e o projeto foi renomeado para **Postgres95**
1996	Marc Fournier, Bruce Momjian e Vadim B. Mikheev lançaram a primeira versão externa da Universidade de Berkeley , o projeto foi renomeado para **PostgreSQL** a fim de refletir a nova linguagem de consulta à base de dados: SQL.
1997	A primeira versão de PostgreSQL, a 6.0, foi libertada.
2000	Foi liberada a versão 7.0. As versões 7.x trouxeram as seguintes novas funcionalidades: *Write-Ahead Log* (WAL), esquemas SQL, *outer joins*, suporte a IPv6, indexação por texto, suporte melhorado a SSL e informações estatísticas do base de dados.
2005	A versão 8.0 foi lançada e entre outras novidades, foi a primeira a ter suporte nativo para Microsoft Windows , pode-se destacar o suporte a *tablespaces*, *savepoints*, *point-in-time recovery*, *roles* e *Two-Phase Commit* (2PC).
2010	Foi lançada a versão mais recente: 9.0

1.2 Aplicabilidade

PostgreSQL é um sistema de base de dados objeto-relacional *open source* poderoso. Ele funciona em todos os principais sistemas operacionais, incluindo Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), e Windows. Possui interfaces de programação nativas para C / C + +, Java. Net, Perl, Python, Ruby, Tcl, ODBC, entre outros, e documentação excecional.

Existem sistemas PostgreSQL ativos em ambientes de produção que gerem mais de 4 *Terabytes* de dados. Alguns limites do PostgreSQL estão incluídos na tabela abaixo.

Limit	Value

Maximum Database Size	Unlimited
Maximum Table Size	32 TB
Maximum Row Size	1.6 TB
Maximum Field Size	1 GB
Maximum Rows per Table	Unlimited
Maximum Columns per Table	250 - 1600 depending on column types
Maximum Indexes per Table	Unlimited

2 Armazenamento e Estrutura de ficheiros

2.1 Armazenamento físico da base de dados

2.1.1 Database File Layout

Esta secção descreve o formato de armazenamento a nível de ficheiros e diretorias. Todos os dados necessários para um *cluster* de base de dados são armazenados dentro da diretoria chamada normalmente por PGDATA, sendo o local comum /var/lib/pgsql/data, esta mesma contem várias subdiretorias e arquivos de controlo. O conteúdo desta diretoria é a seguinte, apresentada na tabela.

Item	Descrição
PG_VERSION	O ficheiro contém a maior número de versões do PostgreSQL
base	Subdiretoria contendo subdiretoria por base de dados
global	Subdiretoria contém tabelas de todo agrupamento (*cluster-wide*), como a pg_database
pg_clog	Subdiretoria contém transações de *commit status* data
Pg_subtrans	Subdiretoria contém dados de *status* de subtransacções
Pg_tbspc	Subdiretoria contém ligações simbólicas para os *tablespaces*
pg_xlog	Subdiretoria contém ficheiros WAL (Write Ahead Log)
Postmaster.opts	Um ficheiro de gravar as opções de linha de comando do postmaster com que foi iniciado pela última vez
postmaster.pid	Um ficheiro de bloqueio(*lock*) gravação do *postmaster* atual PID e ID segmento de memória compartilhada (não presente após o desligamento *postmaster*)

Para cada base de dados no *cluster* existe uma subdiretoria dentro de PGDATA / base, em homenagem a OID da base de dados em pg_database. Este subdiretoria é o local

padrão para os ficheiros de base de dados; em particular, os seus catálogos do sistema são armazenados lá.

Quando uma tabela ou índice ultrapassa 1Gb, estes são divididos em segmentos de tamanho gigabyte. O nome dos ficheiros passam a denominar-se por *filenode* (para o primeiro), e os seguintes passam a ter um valor numérico associado (ex. filenode.1, filenode.2, etc). No caso das tabelas com entradas muito grandes terão uma tabela TOAST (The Oversized-Attribute Storage Technique) associada.

Nota: OID é um identificador único para cada objeto (tabela llamese, coluna, tipo de dados, etc) e pode usar esse número para gerar as chaves primárias em uma tabela.

2.1.2 Toast (The Oversized-Attribute Storage Technique)

O PostgreSQL usa um tamanho fixo de página, normalmente de 8Kb, e não permite tuplos para abranger várias páginas, não é possível armazenar valores de campo muito grandes diretamente.

O TOAST ajuda a resolver este problema comprimindo ou dividindo um tuplo em várias linhas. Isto é totalmente transparente para o utilizador e não causa grandes transtornos a nível de código. Obviamente o TOAST só pode ser utilizado em tipos que possam atingir valores grandes, em termos de espaço.

2.1.3 Database page layout

Aqui iremos descrever a forma de armazenamento das tabelas e dos índices. De registar que tanto as tabelas *TOAST* como as de sequência seguem o mesmo formato.
No caso das tabelas será guardada informação referente às linhas, enquanto nos índices iremos guardar as suas entradas. Abstendo-nos das diferenças entre ambos vamos apelida-los de itens.

A tabela a baixo contém a informação correspondente às várias parcelas das páginas dos ficheiros de armazenamento.

Item	Descrição
PageHeaderData	Tamanho de 20 bytes. Contem a informação geral sobre a página, incluído espaço livre de apontadores.
ItemIdData	Array de (offset, length) pares a apontar para os itens. 4 bytes por item.
Free space	O espaço não alocado. Apontadores para novos itens são alocados do início desta área, e os novos itens no final.
Items	Os próprios itens
Spacial space	Método de acesso de índices a dados específicos. Diferentes métodos guardam dados diferentes. Vazio numa tabela comum.

2.2 Buffer management

PostgreSQL mantém a sua própria cache de *buffer* interno. A versão 9.3 usada possui uma estratégia de substituição de *buffer* mais inteligente, que vai fazer melhor uso dos buffers compartilhados disponíveis e melhorar o desempenho. O impacto no desempenho do *vacuum* e *checkpoints* também é menor.

As versões anteriores usavam o least-recently-used (LRU) cache para manter páginas referenciadas em memória. Mas este método não considerava o número de vezes que uma cache específica era acedida, por isso, as pesquisas em tabelas enormes pudessem forçar páginas de cache uteis. O novo algoritmos de cache usa quatro listas separadas para rastrear as páginas de cache mais usadas e frequentemente usadas e otimizar dinamicamente a sua substituição com base na carga trabalho. Isso deve levar a um uso mais eficiente do *buffer* cache compartilhado.

Sendo assim, quando processo necessita de uma bloco de disco, este irá estar guardado num *buffer*. Se o bloco estiver em cache, será marcado como *pinned*,

significando que mais nenhum processo o poderá utilizar, senão encontrar o *buffer* disponível terá de fazer escolha usando o *clock sweep*, sabendo que a buffer cache é simplesmente um *array* de entradas do buffer, escolhe-se um ponto de partida, e move-se ao longo do *array* com um simples *loop* de incremento sobre o número da entradas, depois de chegar ao fim o volta-se novamente ao inicio.

2.3 Partição

A partição refere-se a divisão lógica de uma tabela de grande dimensão para vários pedaços físicos. Este método pode trazer benefícios, tais como, o melhoramento da performance da query em certas situações, particularmente quando as linhas mais acedidas da tabela estão numa só partição ou num pequeno número de partições. Reduz-se também o tamanho dos índices, e faz com que caibam em memória. É possível tirar vantagem da procura sequencial quando se usa uma partição quase na sua totalidade, em vez de se usar índices. Aumenta-se a eficiência dos carregamentos e eliminações inserindo ou removendo partições. Em PostgreSQL podem ser implementados duas formas de partições, sendo estas, as range *partitioning* e *List Partitioning*.

- *Range Partitioning*: A tabela é dividida por intervalos de valores, sendo definida por coluna de chaves ou varias colunas.
- *List Partitioning*: A tabela é dividida listando explicitamente que os valores-chave aparecem em cada partição.

A partição implementa-se ao criar uma tabela principal que não conterá dados, e por onde haverá herança por parte das partições. Em cada partição deverá de ser adicionada restrições para definir os valores de chaves permitidas. Para além disso poderão ser usados índices que melhoram a eficiência na procura, ou opcionalmente criar um *trigger* para a tabela principal.

2.4 Estrutura de dados

Tal como a maioria dos sistemas, este também usa a estrutura de dados em pilha (*Heap*) para a organização de tuplos.

Vantagens:

- Inserções eficientes, com novos records adicionados no fim do ficheiros, devolvendo uma ordenação cronológica dos tuplos.
- Eficiente para relações pequenas onde as indexações causariam *overhead*.
- Eficiente quando a recuperação envolve grande proporções de dados guardados.
- Desvantagens:
- Ineficiência, uma vez que a procura é feita linearmente.
- Requere organização periódica, que poderá levar ao consumo de tempo.

2.5 Clustering

Em PostgreSQL é possível agrupar uma tabela de acordo com o índice. Quando é feito o *cluster* de uma tabela, esta é fisicamente reordenada baseando-se na informação do índice. *Clustering* é uma operação de *one-time*: Quando uma tabela é atualizada posteriormente ao *cluster*, as mudanças não são agrupadas de acordo com o índice, mas o reagrupamento é possível através do comando.

No caso de acedermos uma única linha da tabela, a ordem dos dados na tabela é irrelevante. Contudo, se acedermos algum dado mais do que os outros, e ter índice que os agrupa, haverá benefício em usar o cluster. Caso estejamos a pedir um valor em range *index*, o *cluster* ajudará, pois este identifica a página onde se encontra a primeira linha e a partir daí poderá encontrar os restantes valores nessa página, significando que se poupou o acesso e leitura ao disco e se acelera a *query*.

Quando se cria um *cluster* sobre uma tabela, é adquirido o *lock* de ACCESS EXCLUSIVE. Isto previne que quaisquer outras operações da base de dados, tanto leituras e escritas, de operar sobre a tabela até que o *clustering* tenha terminado.

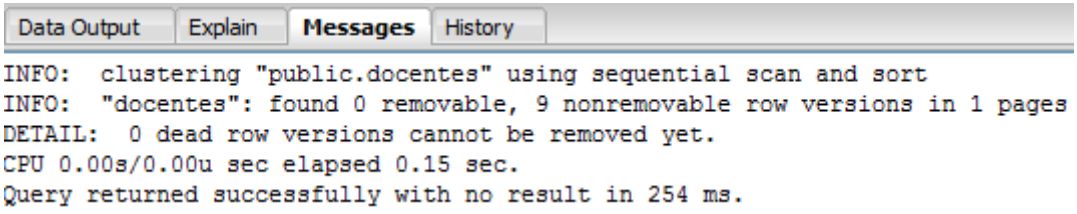
Sintaxe:

CLUSTER [VERBOSE] table_name [USING index_name]

CLUSTER VERBOSE

Exemplo:

```
cluster verbose docentes using docentes_cod_docente_idx
```



3 Indexação e hashing

Em base de dados a utilização de índices facilita o acesso em tabelas, aumentando assim a sua performance, mas uso inapropriado dos índices pode levar a baixo desempenho. O PostgreSQL sendo um sistema bastante completo, este permite diversas operações de tipos de índice, tal como índices multi-coluna. Fornece também métodos de indexação como B-Tree, Hash, GiST e GIN. Os utilizadores também poderão definir os seus próprios métodos, mas isso acabaria por complicar. Uma vez que o índice é criado, não é necessário qualquer tipo de intervenção, porque o sistema faz a atualização do índice quando a tabela é modificada, e poderá usar o índice em consultas quando lhe parecer mais eficiente que a procura sequencial da tabela, é possível analisar essa eficiência através do comando **ANALYSE**.

ANALYZE [VERBOSE] [table [(column [, ...])]]

Quando temos uma cláusula WHERE, um índice parcial é criado. Um índice parcial é um índice que contem entradas para uma determinada porção da tabela, essa porção normalmente tornasse mais útil na indexação do que propriamente a tabela toda em questão. Uma outra possibilidade é usar WHERE com UNIQUE, o que força a singularidade sobre o subconjunto da tabela.

As expressões usadas na cláusula WHERE podem referir-se somente a colunas de uma "sub-tabela", mas poderá usar todas as colunas, não só as que estão a ser indexadas. Presentemente, as *subqueries* e expressões de agregação são proibidas nesta cláusula.

Os índices também podem tirar benefícios dos comandos UPDATES e DELETES com condições de procura. Os índices podem além disso ser usados em junções de pesquisas. Assim, um índice definido em uma coluna que faz parte de uma condição de junção também pode acelerar significativamente as consultas com junções.

Sintaxe de criação de um índice:

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]

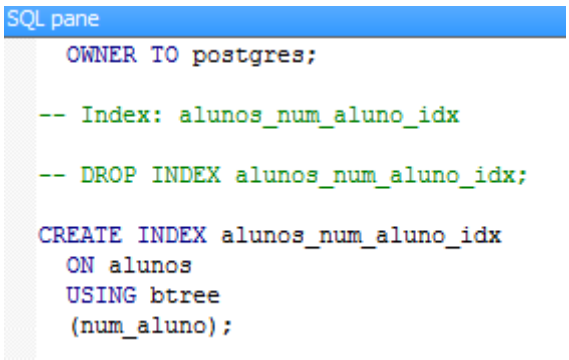
Este comando permite criar um índice com [*name*] sobre uma tabela, usando um determinado método de indexação, por *default* é em B-Tree, a chave em questão é tratada em *column*.

O UNIQUE é opcional, em PostgreSQL só os índices B-Tree podem declarar a singularidade isto significa que quando um índice é declarado como único, várias linhas com valores indexados iguais não são permitidos.

Concurrently é opcional, é usado para criar um índice sem tirar qualquer *locks* que impeçam inserções, atualizações ou remoções concorrentes em tabelas. Um índice padrão constrói um bloqueio sobre a escrita, mas não sobre a leitura, sobre a tabela até que este acabe.

Exemplo usando o comando:
create index on alunos(num_aluno);

Verifica-se que foi criado um *index* B-tree sobre a tabela pessoas por *default*:



Remoção:

DROP INDEX [IF EXISTS] name [, ...] [CASCADE | RESTRICT]

O [IF EXISTS] evita que o sistema envie uma exceção caso não exista o índice com o nome especificado. O [CASCADE] permite a eliminação em cascata onde todos os elementos dependentes do índice, em alternativa ao anterior, o [RESTRICT] permite a eliminação somente do índice em causa sem afetar as dependências. Caso afete, o índice não é eliminado.

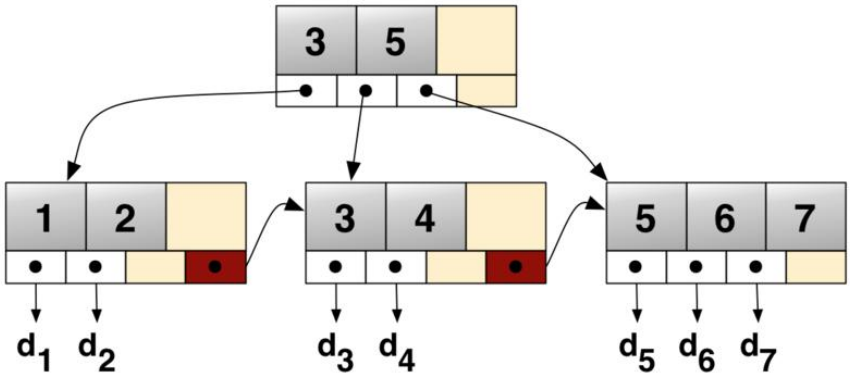
3.1 Tipos de índice

3.1.1 B-Tree

Comando:

CREATE INDEX name ON table USING btree (column);

O PostgreSQL usa índices B+tree, por definição as B+tree guardam todos os nós filhos no último nível, isto é, nas folhas. Todas estas folhas estão ligadas entre si, formando uma lista ligada, sabe-se que todos os caminhos desde a raiz e a folha devem de ter o mesmo caminho. Cada nó que não seja raiz ou folha deverá de ter n/2 e n filhos. No caso das folhas tem de ter (n-1)/2 a n-1 valores. Devido a este tipo de organização a procura no disco é minimizada e mais rápida.



As B-tree's geram igualdades e facilita consultas utilizando intervalos de valores (>= > = < <=). Permite a reorganização automática com pequenas alterações consoante as

modificações, o que significa que não é necessário organização total do ficheiro em causa.

É possível definir o tipo de organização, podendo ser descendente ou ascendente ou se os valores nulos se guardam no fim ou no início, por omissão seria ascendente com os nulos no fim.

3.1.2 Hash

O *hashing* é o valor de identificação produzido através da execução de uma operação numérica, denominada função de *hashing*, em um item de dado. O valor identifica de forma exclusiva o item de dado, mas exige um espaço de armazenamento bem menor. Por isso, o computador pode localizar mais rapidamente os valores de *hashing* que os itens de dado, que são mais extensos. Uma tabela de *hashing* associa cada valor a um item de dado exclusivo.

Comando:

CREATE INDEX name ON table USING hash (column);

Os índices *hash* conseguem manipular simples comparações de igualdade. O planeador da consulta considera usá-lo quando uma coluna de um índice está envolvida numa comparação usando o operador "=". (Estes índices não suportam procuras com *is null*).

Os índices *hash* não são WAL-logged, significa que poderão precisar de ser reconstruídos com o comando REINDEX depois da base de dados falhar. Devido a isso seria desvantajoso usar o índice *hash*.

Nota: WAL (write ahead log) é usado pelo PostgreSQL para manter todas as alterações á base de dados, de modo a conseguir recuperar o estado das bases de dados anterior à falha.

A tabela *hash* dinâmica só existe em memória, a fim de facilitar a estrutura de dados de pesquisa em memória, e não o índice. A ideia do *hashing* dinâmico é efetuar a divisão dos *buckets* ordenadamente. Quando o *bucket* excede a sua capacidade são usados *buckets* de *overflow*.

```
--hash--
create index on departamentos using hash(cod_departamento)
explain analyse select * from departamentos where cod_departamento = 1
select * from departamentos
```

	QUERY PLAN text
1	Index Scan using departamentos_cod_departamento_idx on departamentos
2	Index Cond: (cod_departamento = 1::numeric)
3	Total runtime: 0.070 ms

(cost=0.00..8.02 rows=1 width=110) (actual time=0.032..0.035 rows=1 loops=1)

3.1.3 GiST(Generalised search tree)

Comando:

CREATE INDEX name ON table USING gist (column);

Os índices GiST não são um tipo singular de índices, mas sim uma infraestrutura dentro da qual muitas estratégias de indexação diferentes podem ser implementadas. Os operadores com que a GiST index pode ser usado consoante a estratégia de indexação. Permite então a definição de métodos de indexação personalizados. O GIST permite indexar tipos de dados geométricos de duas dimensões, para isso suporta perguntas com operadores do tipo: <<, &<,&>, >>, <<|, &<|, |&>, |>>, @>,<@, ~=, &&.

3.1.4 GIN

Comando:

CREATE INDEX name ON table USING gin (column);

Os índices GIN são índices invertidos que conseguem manipular valores que contenham mais do que uma chave, como por exemplo, *arrays*. Tal como GiST, GIN suporta vários tipos de estruturas de indexação definidos pelo utilizador e operadores particulares com que um índice GIN podem ser utilizados variam de acordo com a estratégia de indexação. Como exemplo, a distribuição padrão do PostgreSQL inclui classes de operadores GIN para matrizes unidimensionais, que suportam consultas indexados usando esses operadores: <@, @>,=, &&.

3.2 Índice multicoluna

Um índice pode ser definido sobre mais do que uma coluna da tabela.

Exemplo 1.1.

CREATE INDEX index_inscricoes ON inscricoes (num_aluno,cod_curso);

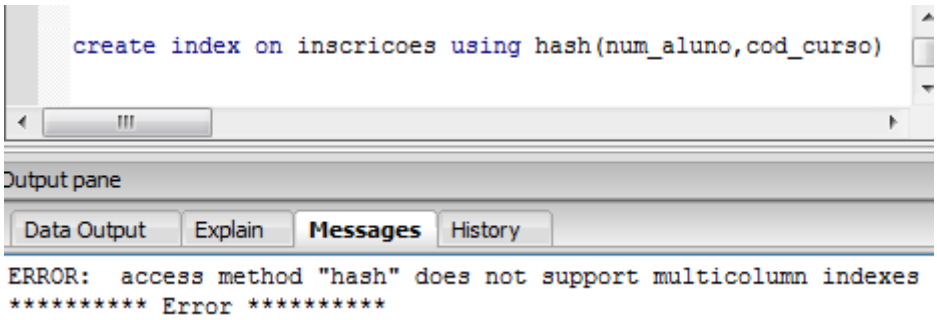
SQL pane

```
    OWNER TO postgres;

-- Index: index_inscricoes

-- DROP INDEX index_inscricoes;

CREATE INDEX index_inscricoes
  ON inscricoes
  USING btree
  (num_aluno, cod_curso);
```

Automaticamente criou um índice B-tree sobre as duas colunas.

Atualmente, somente a B-tree, GIST e tipos de índices GIN suportam índices com várias colunas. Até 32 colunas pode ser especificado.

Hash:



B-tree:

Pode ser usado com consultas que envolvem qualquer subconjunto de colunas do índice, mas é mais eficiente quando existem restrições sobre as colunas principais, permite otimização de procuras.

GiST:

Pode ser utilizado com condições de consulta que envolvem qualquer subconjunto das colunas do índice. Condições em colunas adicionais restringem as entradas devolvidas pelo índice, mas a condição na primeira coluna é o mais importante para determinar o quanto o índice precisa ser digitalizado. Um índice GiST será relativamente ineficaz se a sua primeira coluna tem apenas alguns valores distintos, mesmo se houver muitos valores distintos em colunas adicionais.

GIN:

Pode ser utilizado com condições de consulta que envolvem qualquer subconjunto das colunas do índice. Ao contrário da B-Tree ou do GIST, a eficiência da pesquisa é a mesma para qualquer que seja o atributo usado nas condições da pergunta.

3.2.1 Bitmap com combinação de índices múltiplos

O utilizador não consegue criar os índices bitmaps, mas o PostgreSQL usa-os internamente. Para combinar índices múltiplos, o sistema procura cada índice preciso e prepara o bitmap na memória dando a localização das linhas da tabela que são

relatadas como correspondências das condições do índice. Os bitmaps são ANDed e ORed juntos quando for necessário para a consulta. Finalmente, as linhas da tabela reais são visitados e devolvidos. As linhas da tabela são visitados em ordem física (em disco), isto significa que qualquer ordenação dos índices originais é perdida, sendo necessário ordenar os dados no fim, caso o utilizador queira, usando um ORDER *BY*.

```
--bitmap scan
set enable_bitmapscan to on
set enable_seqscan to off

create index b_tree_local on alunos using btree(local)
select * from alunos
explain verbose select * from alunos where local ='lisboa'  or cod curso >=2
```

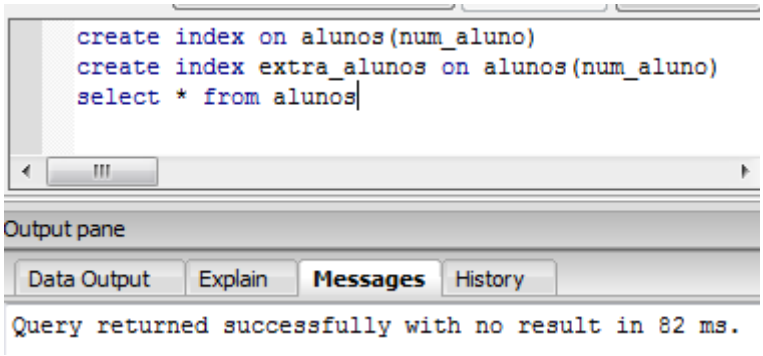
	QUERY PLAN text
1	Bitmap Heap Scan on public.alunos (cost=12.37..16.45 rows=5 width=184)
2	Output: num aluno, nome, local, data nsc, sexo, cod curso
3	Recheck Cond: (((alunos.local)::text = 'lisboa'::text) OR (alunos.cod curso >= 2::numeric))
4	-> BitmapOr (cost=12.37..12.37 rows=5 width=0)
5	-> Bitmap Index Scan on b tree local (cost=0.00..4.14 rows=1 width=0)
6	Index Cond: ((alunos.local)::text = 'lisboa'::text)
7	-> Bitmap Index Scan on alunos num aluno cod curso key (cost=0.00..8.22 rows=4 width=0)
8	Index Cond: (alunos.cod curso >= 2::numeric)

3.3 Índices para organização de ficheiros

Em relação á organização sabe-se que o PostgreSQL usa a *Heap* como estrutura de dados tal como referido no capítulo 3, significando que não é necessária a utilização de índices para a organização interna.

3.4 Múltiplos ficheiros de índices para os mesmos atributos

O PostgreSQL não tem qualquer tipo de restrição no número de índices por tabela ou conjunto de atributos. Consequentemente também não impõe qualquer restrição quanto ao número de índices por tabela. Segue-se em exemplo:


create index on alunos(num_aluno)
create index extra_alunos on alunos(num_aluno)
select * from alunos
Output pane
Data Output
Explain
Messages
History
Query returned successfully with no result in 82 ms.

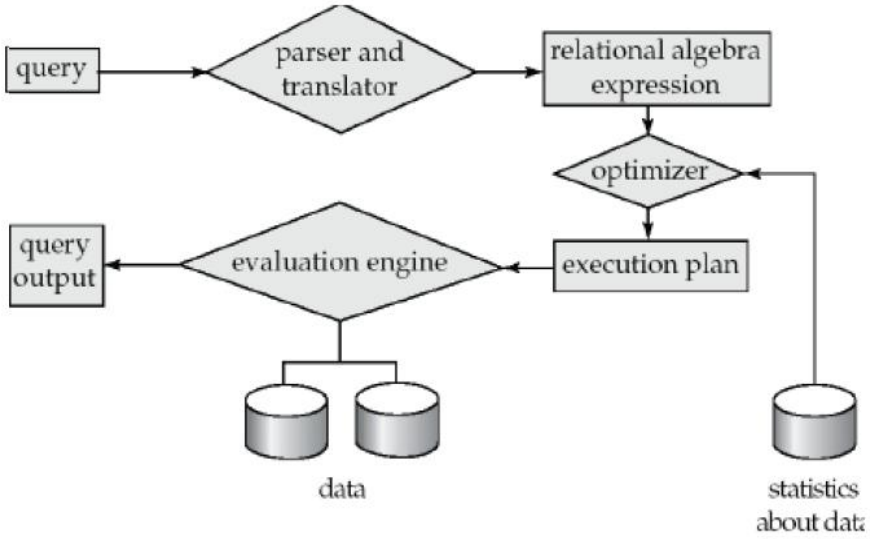
4 Processamento e otimização de perguntas

O processamento e otimização de perguntas é uma ferramenta muito forte para quando nos é apresentada a função de gestão de sistemas de bases de dados.

Consiste num conjunto de operações complexas que permitem manipular dados da base de dados. Podendo determinar o modo mais eficiente de executar uma consulta no sistema, tentando-se com que esta seja do modo mais otimizado, ou seja, do modo mais rápido e com menor consumo de recursos, ficando assim a sua execução mais rápida.

Este conjunto de operações contém a conversão de uma linguagem de alto nível (tal como o SQL) por forma a serem utilizadas pelo nível físico do sistema, tal como a transformação das consultas. No PostgresSQL, esta conversão da linguagem passa por uma linguagem intermédia que tem por base a transformação de consultas numa árvore (a consulta numa forma estruturada). Este sistema dispõe de um conjunto de comandos que permitem a configuração de alguns parâmetros nesta fase.

Para que seja mais percetível as transformações por que passa uma consulta ao sistema, na figura seguinte é esquematizado este processo.



4.1 Percurso de uma consulta no sistema

No PostgreSQL o percurso de uma consulta é idêntico à imagem anterior. Agora, vamos estudar esse percurso de uma forma mais aprofundada. Este percurso pode ser descrito pelos seguintes passos fundamentais:

1. Estabelece-se uma ligação da aplicação para um servidor do PostgreSQL. A aplicação transmite a pergunta e espera por uma resposta por parte deste.
2. O *parser* verifica que a pergunta está correta, a nível de sintaxe, e cria uma *query tree*.
3. O *rewrite system* pega nesta árvore criada e verifica se existe alguma regra a aplicar nesta árvore.
4. O planeador/optimizador do sistema pega na árvore reescrita e cria um plano de execução que servirá de *input* para o executor.
5. O executor pega nestes dados e executa-os retornando os tuplos na forma representada na árvore. O executor usa o *storage system* enquanto percorre relações, executando ordenações e junções, avaliando qualificações e retornando os resultados pretendidos.

4.1.1 Ligação ao servidor

Existe um processo responsável por inicializar e terminar o servidor e também por tratar de pedidos de conexões por parte de novos clientes.

No PostgreSQL estas ligações ao servidor são efetuadas utilizando um mecanismo de processo por utilizador, baseado no modelo cliente/servidor, onde um processo de um cliente está ligado apenas a um processo de servidor. Esta relação é realizada por um *postgres* (um *master process* – servidor mestre que cria um novo processo de servidor) que recebe pedidos numa porta específica de TCP/IP.

4.1.2 Parser

O *parser* recebe uma *string* (codificada em ASCII) que representa a consulta a ser processada. Dada esta *string* verifica a sua validade a nível de sintaxe. Caso não contenha erros, é gerada uma árvore (*query tree*), caso contrário é retornada uma mensagem de erro.

Para que tal seja possível, o *lexer,* que está definido no ficheiro *scan.l* e é responsável por reconhecer identificadores, por cada identificador descoberto, é gerado um *token* e passado para o *parser*. Este está definido no ficheiro *grammar.y* e consiste um conjunto de regras gramaticais e ações a ser executadas (o código das ações, escrito em c, é utilizado para construir a arvore).

A árvore gerada posteriormente é entregue ao *rewrite system,* que lhe aplica um conjunto de regras por forma a produzir um resultado com zero ou mais árvores (*query tree*) para serem posteriormente passadas ao planeador/optimizador.

4.1.3 Rewrite system

Neste ponto do processo é feita uma interpretação semântica á árvore, gerada anteriormente, onde é aplicado um conjunto de regras por forma a obter uma *query tree* mais simplifica, visto que as operações podem ser substituídas por operações mais básicas. Esta simplificação tem como objetivo facilitar o próximo passo (planeador/optimizador).

No PostgreSQL é possível ao utilizador definir regra a serem analisadas e possivelmente aplicadas nesta fase.

Uma *Query Tree* consiste numa representação interna de uma pergunta SQL onde todos os elementos são guardados separadamente.

Estas árvores podem ser consultadas no ficheiro de log do servidor ativando os parâmetros de configuração debug_print_parse, debug_print_rewrittenou

debug_print_plan. Para se obterem as regras a serem aplicadas pode-se consultar a pg_rewrite.

A estrutura das *query tree* é a seguinte:

- Tipo de comando – valor que contem a indicação de qual a instrução a ser executada. Podendo ser instruções de SELECT, INSERT, UPDATE ou DELETE;
- Tabela de abrangência – lista que descreve as relações usadas na consulta a ser processada.
- Relação resultante – índices para a tabela de abrangência que identificam a relação para onde irão os dados da consulta (por exemplo, num INSERT o resultado é a tabela ou vista onde as mudanças são feitas).
- Lista de alvos – lista que recebe os resultados da consulta efetuada. No caso de um SELECT, essas expressões são as únicas que constroem o resultado final da consulta. Elas correspondem às expressões entre as palavras-chave SELECT e FROM. A instrução DELETE não produz qualquer resultado. No entanto, uma instrução de INSERT tem como resultado os novos tuplos a serem inseridos. Tal resultado é obtido pela instrução VALUES no caso de serem especificados os tuplos a serem inseridos ou pelo resultado da consulta interna gerado pela cláusula SELECT (no caso da instrução INSERT.. SELECT). Finalmente, para instruções do tipo UPDATE a lista de alvos descreve os tuplos a serem atualizados e quais os seus novos valores. Esta expressão pode ser um valor constante, um apontador para uma coluna de uma das relações da tabela de abrangência, um parâmetro, ou uma árvore de expressões constituída por chamadas de funções, constantes, variáveis, operadores, etc.
- Qualificação – expressão cujo resultado é um valor booleano que informa se uma das operações (de INSERT DELETE UPDATE ou SELECT) deve ou não ser executada para o resultado final. Corresponde à cláusula WHERE duma expressão SQL.
- Árvore de junções – mostra a estrutura da cláusula FROM. Para uma consulta simples como SELECT * FROM a, b, c; a árvore de junções representa apenas uma lista de elementos existentes a seguir ao FROM (a, b, c). Se estivermos a utilizar OUTER JOIN já temos de efetuar a junção seguindo uma determinada

ordem e neste caso a árvore representa a estrutura das expressões de junção. As restrições associadas a determinadas junções (através das expressões ON ou USING) são guardadas como expressões de qualificação associadas a esses nós da árvore de junções. Por vezes também é útil associar a estes nós as expressões da cláusula WHERE como expressões de qualificação. Posto isto, na realidade a árvore de junções representa tanto a cláusula FROM como a WHERE dum comando SELECT.

- Outros - restantes instruções existentes na consulta (como a cláusula ORDER BY).

4.1.4 Planeador/Optimizador

O planeador/optimizador tem como função gerar um plano de execução ótimo, a partir das *query tree* obtidas através do ponto anterior. Visto que existem diversas formas de executar uma consulta, sendo que cada forma tem custos associados, o planeador/optimizador procura (se for possível) todas as hipóteses existentes para obter a que é mais vantajosa em termos de custos. A criação de todos estes planos é uma operação computacionalmente muito pesada. O PostgreSQL quando se depara com uma situação que pode consumir muitos recursos ao nível do CPU ou memória (ou quando se ultrapassa um determinado limite de JOIS numa consulta) utiliza uma ferramenta denominada *Genetic Query Optimizer*, por forma a conseguir gerar um plano ótimo em tempo útil.

A árvore do plano gerado (*plan tree*) consiste em nós de junções, ordenações e agregações e seleções. O planeador anexa ainda aos nós necessários projeções.

4.1.5 Executor

O executor pega no plano dado pelo planeador/optimizador e recursivamente processa-o para extrair os tuplos pretendidos. Sempre que um nó do plano é chamado, este tem de devolver um ou mais tuplos, ou reportar que já terminou.

Caso a consulta seja do tipo SELECT, o executor limita-se a devolver todos os tuplos retornados pela avaliação do plano. Para operações de INSERT cada tuplo devolvido é inserido na respetiva tabela. Para operações de UPDATE a execução do plano devolve todos os tuplos com os valores atualizados seguidos do identificador desse tuplo. De seguida o executor insere na tabela os tuplos com os valores atualizados e marca os tuplos referenciados pelos identificadores como apagados. Caso se trate de um DELETE então a execução do plano devolve os identificadores dos tuplos a serem removidos e remove-os efetivamente.

4.2 Operações básicas

Para obter os algoritmos que vamos usar, deveremos utilizar antes da consulta a instrução EXPLAIN e após a criação a criação ta tabela de exemplos utilizar a instrução ANALYSE. Caso estas instruções não forem efetuadas os dados estatísticos serão recalculados e o algoritmo *Seq San* será sempre utilizado.

4.2.1 Algoritmos para a Selecção

Seq Scan - é o algoritmo de pesquisa mais básico. É sempre gerado uma *query plan* para este algoritmo. O modo de funcionamento do *Sequential* Scan passa por percorrer todas as linhas de uma tabela, e, por cada linha, verifica se a condição é respeitada. De notar que, se a pesquisa for efetuada numa chave primária, este algoritmo pode termina ao encontrar o primeiro *tuplo* que satisfaça a condição.
Para ativar ou não o uso deste algoritmo usa-se enable_seqscan(boolean);

```
explain select *  from inscricoes
```

	QUERY PLAN text
1	Seq Scan on inscricoes (cost=0.00..1.23 rows=23 width=58)

Bitmap Scan - este algoritmo de pesquisa é diferente do anterior, mas algo semelhante ao de pesquisa por índice. O bitmap scan utiliza os diferentes índices criados numa dada tabela para otimizar e aumentar a velocidade de pesquisas mais complexas. De notar que, poderá realizar bastantes *seeks*, pelo que devera ser utilizado para um conjunto pequeno de valores.

Para ativar ou desativar usa-se o enable_bitmapscan(boolean);

```
--bitmap scan
set enable_bitmapscan to on
set enable_seqscan to off

create index b_tree_local on alunos using btree(local)
select * from alunos
explain verbose select * from alunos where local ='lisboa'  or cod curso >=2
```

	QUERY PLAN text
1	Bitmap Heap Scan on public.alunos (cost=12.37..16.45 rows=5 width=184)
2	Output: num aluno, nome, local, data nsc, sexo, cod curso
3	Recheck Cond: (((alunos.local)::text = 'lisboa'::text) OR (alunos.cod curso >= 2::numeric))
4	-> BitmapOr (cost=12.37..12.37 rows=5 width=0)
5	-> Bitmap Index Scan on b tree local (cost=0.00..4.14 rows=1 width=0)
6	Index Cond: ((alunos.local)::text = 'lisboa'::text)
7	-> Bitmap Index Scan on alunos num aluno cod curso key (cost=0.00..8.22 rows=4 width=0)
8	Index Cond: (alunos.cod curso >= 2::numeric)

Index Scan - este algoritmo baseia-se numa pesquisa sobre um índice. Se for dado um valor inicial de pesquisa, o *Index* Scan apenas irá começar nesse valor, e se for dado um valor final, o *Index Scan* parará nesse valor. Quando comparado com o *seq* scan, a vantagem do *index* scan é que percorre apenas alguns valores ao contrário do *seq* scan que percorre todas as entradas da tabela. Contudo, índex scan retorna os valores de acordo com a ordem no índice, ao contrário do *seq* scan que retorna os valores por ordem de entrada na tabela. Este retorno de valores por ordem dos índices pode causar um aumento de *seeks*.

Para ativar ou desativar usa-se o enable_indexscan(boolean);

```
explain analyse select *  from inscricoes where num aluno=1 and cod curso=2 ;
```

	QUERY PLAN text
1	Index Scan using inscricoes pkey on inscricoes (cost=0.14..8.16 rows=1 width=58) (actual time=0.043..0.046 rows=3 loops=1)
2	Index Cond: ((num aluno = 1::numeric) AND (cod curso = 2::numeric))
3	Total runtime: 0.103 ms

TID Scan – este algoritmo raramente é utilizado. Cada tuplo tem um identificador que é único na tabela, pelo qual é possível efetuar uma pergunta. Este identificador pode ser acedido através de um ctid, atributo especial para cada tabela.

4.2.2 Algoritmos para a Junção

Nested Loop Join – este algoritmo é bastante simples, que pode ser utilizado em diversas situações, porem, com custos elevados. O *Nested Loop Join* consiste em comparar cada tuplo da primeira relação com todos os tuplos da segunda e, caso satisfaçam as condições de junção esses pares de tuplos são adicionados ao espaço de resultados. O custo pode tornasse menor nos casos em que ambas as relações cabem em memória ou quando existem índices nos atributos de junção em pelo menos uma das tabelas.

Para ativar este algoritmo usa-se o enable_nestloop(boolean);

```
--nested loop
select * from inscricoes
explain select *
from inscricoes i1, inscricoes i2
where i1.nota<15 and i1.nota <i2.nota
```

	QUERY PLAN text
1	Nested Loop (cost=20000000000.00..20000000005.33 rows=61 width=116)
2	Join Filter: (i1.nota < i2.nota)
3	-> Seq Scan on inscricoes i1 (cost=10000000000.00..10000000001.29 rows=8 width=58)
4	Filter: (nota < 15::numeric)
5	-> Materialize (cost=10000000000.00..10000000001.34 rows=23 width=58)
6	-> Seq Scan on inscricoes i2 (cost=10000000000.00..10000000001.23 rows=23 width=58)

Merge Sort Join – este algoritmo apenas pode ser utilizado em junções naturais e junções usando condições de igualdade. O Merge Sort Join começa por ordenar as relações segundo o atributo de junção (caso não se encontrem ordenadas) e de seguida verifica o topo de cada relação em cada passo e devolve o menor. Um ponto interessante deste algoritmo é o facto de cada tabela apenas ser percorrida uma vez.

Para ativar o uso deste algoritmos usa-se o enable_mergejoin(boolean);

```
SET enable_sort to on;

explain select *
from inscricoes i1, alunos a
where i1.nota > 10 and i1.num_aluno = a.num_aluno
```

	QUERY PLAN text
1	Merge Join (cost=0.27..24.95 rows=4 width=242)
2	Merge Cond: (i1.num_aluno = a.num_aluno)
3	-> Index Scan using inscricoes_pkey on inscricoes i1 (cost=0.14..12.54 rows=8 width=58)
4	Filter: (nota > 10::numeric)
5	-> Index Scan using alunos_num_aluno_cod_curso_key on alunos a (cost=0.14..12.32 rows=12 width=184)

Hash Join – este algoritmo fornece capacidades de paralelização, no entanto, este mecanismo não foi implementado no PostgreSQL. Tal como o *Merge Sort Join*, este algoritmo também só pode ser utilizado em junções naturais e em junções usando condições de igualdade. Porém é um algoritmo substancialmente diferente dos restantes. O *Hash Join* consiste em particionar as relações r e s segundo uma função de *hash* e de seguida efetuar a junção em cada partição r e s (utilizando correspondência por *hash keys*).

Para ativar o uso deste algoritmo usa-se o enable_hashsort(boolean);

4.2.3 Algoritmos para a Ordenação

Sort – este algoritmo é chamado quando é usada a cláusula ORDER BY. O algoritmo *Sort* é aplicado apos a obtenção de um conjunto de respostas, ordenando estas. O PostgreSQL utiliza dois tipos de ordenação, o *quicksort* (por memória principal) e o *merge sort* (por disco rígido).

Para ativar ou desativar o uso deste algoritmo usa-se o enable_sort(boolean).

```
select * from alunos
explain select * from alunos order by data_nsc
```

	QUERY PLAN text
1	Sort (cost=10000000001.34..10000000001.37 rows=12 width=184)
2	Sort Key: data_nsc
3	-> Seq Scan on alunos (cost=10000000000.00..10000000001.12 rows=12 width=184)

4.2.4 Agregação

A agregação é implementada através de algoritmos baseados em *hash* e em ordenação. As funções de agregação devolvem um único valor de um conjunto de valores de *input*. Utilizado sempre que se usam instruções como: AVG(), COUNT(), MAX(), MIN(), SUM(), etc. O operador hashaggregate também é usado quando se inclui a clausula group by.

```
explain select avg(nota)
from inscricoes
```

	QUERY PLAN text
1	Aggregate (cost=10000000001.29..10000000001.30 rows=1 width=12)
2	-> Seq Scan on inscricoes (cost=10000000000.00..10000000001.23 rows=23 width=12)

4.2.5 Eliminação de Duplicados

A eliminação de duplicados pode ser feita aquando de consultas que usam a cláusula DISTINCT, que recorre ao operador UNIQUE. Este operador necessita que o conjunto de dados esteja ordenado.

```
explain select distinct nome
from alunos
order by nome
```

	QUERY PLAN text
1	Unique (cost=10000000001.34..10000000001.40 rows=12 width=78)
2	-> Sort (cost=10000000001.34..10000000001.37 rows=12 width=78)
3	Sort Key: nome
4	-> Seq Scan on alunos (cost=10000000000.00..10000000001.12 rows=12 width=78)

4.2.5 Append

Append trata das operações de união. Para este operador é necessário duas tabelas ordenadas. A medida que se vai juntando vai se eliminando os duplicados.

4.2.6 SubQuery e SubPlan

Subquery e *SubPlan* são operadores relacionados com operações básicas sobre conjuntos, tais com a união e subselecções, respetivamente.

4.2.7 SetOp

Há quatro operadores de SetOp: INTERSECT, INTERSECT AL, EXCEPT, EXCEPT ALL. Estes operadores requerem sempre dois conjuntos de entrada, combinando os inputs, em listas ordenadas e depois em grupos de linhas idênticas.

```
explain select * from inscricoes
except select * from inscricoes
```

	QUERY PLAN text
1	HashSetOp Except (cost=10000000000.00..20000000003.61 rows=23 width=58)
2	-> Append (cost=10000000000.00..20000000002.92 rows=46 width=58)
3	-> Subquery Scan on "*SELECT* 1" (cost=10000000000.00..10000000001.46 rows=23 width=58)
4	-> Seq Scan on inscricoes (cost=10000000000.00..10000000001.23 rows=23 width=58)
5	-> Subquery Scan on "*SELECT* 2" (cost=10000000000.00..10000000001.46 rows=23 width=58)
6	-> Seq Scan on inscricoes inscricoes_1 (cost=10000000000.00..10000000001.23 rows=23 width=58)

4.3 Mecanismos para Expressões Complexas

4.3.1 Materialização

Materialização consiste em analisar as expressões (de baixo para cima), guardando o seu resultado em relações temporárias guardadas na base de dados. Os resultados de cada operação intermédia são armazenados e utilizados em avaliações de operações nos níveis superiores.

O PostgreSQL considera materialização quando o planeador/optimizador decide que tem menos custos, tal como em situações em que pode ser necessário obter mais do que uma vez o mesmo tuplo da mesma relação. No entanto, o utilizador pode forçar a não utilização de materialização com o comando: set enable_material = off.

4.3.2 Pipelining

Pipelining consiste na passassem de tuplos para as operações "pai", durante a execução da operação corrente.

Para além de disponibilizar diversos mecanismos, não permite ao utilizador a escolha de qual mecanismo usar, sendo responsável por esta escolha o planeador/optimizador, dando este preferência ao *pipelining* (que em principio obtém resultados mais rápido).

4.3.3 Paralelização

Paralelização é um mecanismo que consiste na execução de diversas tarefas paralelamente.

Embora seja um ótimo mecanismo, o PostgreSQL não o implementa.

4.4 Planos

Como já foi referido anteriormente, a escolha do plano ótimo de consulta é fundamental para uma boa performance por parte do sistema, porém pode ser bastante dispendiosa e, em certos casos, pode não ser possível gerar todos os planos, sendo necessário recorrer a estimativas. A estrutura de um *query plan* consiste numa

árvore de nós, onde cada nó diz respeito à pesquisa que têm de ser realizada para que tal consulta produza resultados.

Para que tal plano seja possível de visualizar é necessário utilizar o comando EXPLAIN seguido da consulta pretendida.

EXPLAIN [ANALYZE] [VERBOSE] statement

Este comando mostra as informações sobre o plano de execução para uma determinada consulta em SQL. Tais como, a forma como a tabela vai ser pesquisada (se por pesquisa sequencial ou por índex) e, se múltiplas tabelas forem referenciadas, que algoritmos de junção vão ser utilizados. Contudo, a informação mais importante é o custo das operações medidas em unidades de acesso a disco.

Parâmetros:

Analyze – esta opção provoca a execução da consulta, não somente a sua planificação.

Verbose – esta opção inclui no output do EXPLAIN a representação interna da árvore de planificação.

Statement – qualquer consulta SQL (*SELECT, INSERT, UPDATE, DELETE, EXECUTE, DECLARE*) que o utilizador pretenda visualizar o planeamento.

```
explain analyse select * from alunos
```

	QUERY PLAN text
1	Seq Scan on alunos (cost=10000000000.00..10000000001.12 rows=12 width=184) (actual time=0.012..0.014 rows=12 loops=1)
2	Total runtime: 0.052 ms

Onde os valores obtidos têm o seguinte significado:

- Cost – o custo estimado ate ser possível produzir output (primeiro valor) e o custo total se todos os tuplos forem lidos (segundo valor).
- Rows – estimativa do número de tuplos a devolver.
- Width – estimativa da média do tamanho dos tuplos devolvidos (em bytes).

Os parâmetros, que podem ser editados, usados para os cálculos do custo são os seguintes:

- ***seq_page_cost*** **-** custo de obter uma pagina do disco, que faça parte de uma pesquisa sequencial. Por defeito, tem como valor 1.0.
- ***random_page_cost*** - custo de obter uma pagina não sequencial do disco. Por defeito, tem como valor 4.0.
- ***cpu_tuple_cost*** **-** custo de processamento de um tuplo. Por defeito, tem como valor 0.01.

4.5 Estimativas

No PostgreSQL o planeador/optimizador necessita de estimar o custo de um plano de execução (estimando o numero de linhas que uma determinada consulta ira devolver) para obter o melhor plano de execução. É bastante importante que tais estimativas tenham a precisão necessária para que o planeador/optimizador escolha a opção que contem o menor custo. Para que estes valores possam ser calculados e analisados, é necessário guardar valores estatísticos sobre os dados da base de dados.

É através do comando ANALYSE que são obtidas as estatísticas sobre as tabelas pertencentes a base de dados. Estas estatísticas, no PostgreSQL, são constituídas entre outros dados, pelo número de entradas em cada tabela e em cada índice, tal como o número de blocos em disco ocupados por cada tabela e índice.

Usando o comando VACUUM é possível recuperar espaço ocupado por tuplos eliminados, tal com é possível quando usado em conjunto com o comando ANALYSE, otimizar e atualizar as estatísticas da base de dados.

Esta informação está contida na vista pg_stats, que contem as estatísticas calculadas tendo como base as estatísticas reais que estão contidas nos catálogos do sistema.

Como a recolha de dados estatísticos pode adicionar alguma sobrecarga na execução de consultas, o sistema PostgreSQL, pode ser configurado para permitir ou não que

este colete a recolha destes dados. Estas configurações são efetuadas através de parâmetros que podem ser editados no ficheiro postgresql.conf.

- **track_activities** – parâmetro que permite a monitorização do comando atual que está a ser executado por qualquer processo do servidor.
- **track_counts** – parâmetro que controla se as estatísticas são recolhidas sobre acessos a tabelas ou a índices.
- **track_functions** – parâmetro que permite seguir a utilização das funções definidas pelo utilizador.

Normalmente, estes parâmetros são definidos no arquivo postgresql.conf aplicando-se a todos os processos do servidor, mas é possível ativa-los ou desativa-los para sessões individuais através do comando SET (por uma questão de segurança, apenas os *super* utilizadores podem alterar estes parâmetros com este comando).

O *colector* de estatísticas guarda a informação recolhida para ficheiros temporários guardados por defeito na diretoria stats_temp_directory, pg_stat_tmp. Quando o servidor termina é armazenada uma copia permanente dos dados estatísticos na diretoria global. Se for pretendido aumentar a performance, diminuindo as operações I/O em disco, poder-se-á apontar o parâmetro stats_temp_directory para um sistema de ficheiros em RAM.

5 Gestão de transações e controlo de concorrência

5.1 Conceito de Transação

Coleções de operações que formam uma unidade lógica de trabalho são chamadas transações. Um sistema de base de dados deve garantir as transações de sistema operacional de execução adequados apesar das falhas, ou a transação inteira se execute ou nada disso faz. Para além disso, deve gerenciar a execução simultânea de transações de uma forma que evita a introdução de inconsistência. As atualizações feitas sobre uma transação aberta são invisíveis para outras transações até que a transação seja concluída, quando todas as atualizações se tornam visíveis ao mesmo tempo. Para que tal garantia seja possível, as transações de um SGBD deve respeitar as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Deve de ser garantida que todas as atualizações feitas por uma transação são registrados no armazenamento permanente (ou seja, em disco) antes da transação ser considerada completa.

O PostgreSQL não suporta transações distribuídas, por isso todos os comandos de uma transação são executados por um *backend*. E de momento também não lida com transações aninhadas.

5.1.1 Propriedades ACID

Atomicidade: Do ponto de vista das outras transações, ou a transação acontece por inteiro, ou nada acontece.

Consistência: A execução isolada da transação (i.e. sendo a única a correr) preserva a consistência da BD.

Isolamento: Mesmo que várias transações estejam em execução concorrente, qualquer que seja o par Ti e Tj, Ti considera que, ou Tj acabou antes de Ti começar, ou começará depois de Ti acabar (todas as transações consideram estar sozinhas em execução).

Durabilidade: Depois de uma transação completar com sucesso, todas as alterações feitas à BD persistem (mesmo depois de falhas de sistema).

5.1.2 Estados de transação

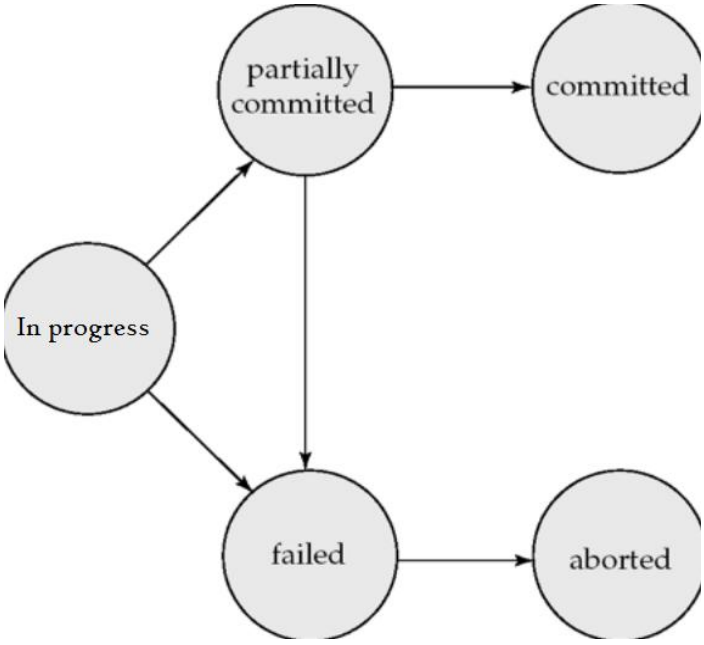
In progress - o estado inicial, a transação fica neste estado enquanto esta se executa.

Partially Committed - depois da última afirmação ser executada

Failed – Depois da descoberta de que a execução normal não poder ser executada

Aborted – depois da execução ser rolled back e a base de dados ter restaurado para o seu estado antes do início da operação.

Committed –Após a conclusão bem sucedida.



5.1.3 MVCC (Multiversion Concurrency Control)

Internamente a consistência dos dados é mantida usando o modelo MVCC. Significa que enquanto se consulta a base de dados cada transação vê uma *snapshot* dos dados (sendo uma versão da base de dados) como era há algum tempo atras, independente do estado corrente dos dados subjacentes. Isto protege a transação de ver dados inconsistentes que poderiam causados por atualizações de uma outra concorrente sobre os mesmos dados, fornecendo isolação da transação por cada sessão de base de dados.

MVCC minimiza a disputa do bloqueio, a fim de permitir que o desempenho razoável em ambientes multiutilizadores.

5.2 Níveis de Isolamento

Existem quatro níveis de isolamento definidos como *standard* no SQL.

O objetivo dos níveis de isolamento é evitar que ocorram três tipos situações indesejadas. Situações em que uma transação lê dados escritos por outra transação que ainda não tenham sido *committed* (*Dirty read*), casos em que uma transação lê dados que tinha lido anteriormente e que entretanto foram alterados por outra transação (*Nonrepeatable read*) e situações em que uma transação reexecuta uma consulta cujo resultado satisfaz uma determinada condição e que no momento da reexecução o resultado da consulta já não satisfaz a condição devido a uma transação que foi *committed* entretanto (*Phantom read*).

No nível de isolamento *Read committed* as *queries* realizadas à base de dados vêm apenas os dados que foram *committed*. No entanto se entre duas queries de uma transação existir um *commit* de outra transação, as duas *queries* podem ver dados

diferentes e/ou fazer modificações aos dados diferentes. Portanto para este nível de isolamento é possível a ocorrência de situações de *Nonrepeatable read*.

Para o nível de isolamento *Repeatable read*, trata-se da mesma situação que com *Read commited*, com diferença que neste caso a transação quando inicia obtém um *snapshot* da base de dados, e para toda a transação as *queries* são feitas com base no *snapshot*. Este nível de isolamento evita situações de *Repeatable read* pois, *queries* veem e modificam os mesmos dados independentemente do momento em que a *query* é executada.

O nível de isolamento *Serializable* trata de simular que as operações são executadas sequencialmente. O comportamento é semelhante ao nível *Repeatable read* com a diferença que é possível uma transação verificar que uma linha, que é obtida ou modificada pela transação, foi excluída ou alterada, e nesse caso a transação aguarda que a transação que começou primeiro faça *commit* ou *abort*. Caso a outra transação faça *commit* a transação aborta, caso contrário a transação faz *commit*.

No PostgreSQL são suportados os quatro níveis de isolamento, no entanto apenas implementa dois tipos, o *Read committed* e *Serializable*. Quando é selecionado o nível *Read uncommitted* o nível oferecido é o *Read committed*. Quando é selecionado o nível *Repeatable read* é oferecido o nível *Serializable*.

Num sistema puramente *Serializable* duas transações concorrentes deixam os dados exatamente no mesmo estado que se tivessem sido executadas em modo sequencial. No PostgreSQL o *Serializable* não é puro, pois para garantir este nível de isolamento seria necessária a utilização de *predicable locks* que impede que uma transação altere uma determinada linha que esteja a ser lida por outra transação em simultâneo. Este tipo de *locks* não é utilizado pelo PostgresSQL porque diminui bastante a *performance* dum sistema, e os problemas com origem da sua não utilização é pouco expectável que ocorram.

5.3 Explicit locking

O PostgreSQL fornece vários modos de bloqueio para poder controlar acesso concorrente a dados de tabelas. Alem disso, vários comandos adquirem automaticamente *locks* apropriados para garantir que as tabelas referenciadas não são eliminadas ou modificadas de forma incompatível enquanto os comandos estão a ser executados.

Por exemplo, o TRUNCATE não pode ser executado concorrentemente com segurança com outras operações sobre a mesma tabela, então obtém um lock exclusivo sobre a tabela.

5.3.1 Table-level locks

LOCK TABLE obtém um *table-level lock*, se estiver a espera de um *lock* conflituante que esteja para ser libertado. Se NOWAIT for especificado, o LOCK TABLE não espera para adquirir o *lock* esperado, se não for possível adquirir o *lock* de imediato, o comando é a abortado ou dá erro. Uma vez obtido, o *lock* é mantido até a restante da transação corrente.

Nota: todos os *locks* são libertados no fim da transação.

Os *locks* ao nível das tabelas podem ser criados com vários modos. Os nomes dos vários modos são históricos, e o que interessa de facto são quais os modos que causam conflitos entre si.

Table-level Lock Modes / niveis de granularidade

ACCESS SHARE – O comando SELECT adquire este *lock* em tabelas referenciadas. No geral, para qualquer *query* que só leia a tabela e não a modifique.

ROW SHARE – Para comandos como SELECT FOR UPDATE e SELECT FOR SHARE.

ROW EXCLUSIVE – Para comandos como UPDATE,DELETE, e INSERT. No geral, este modo será requerido para qualquer comando que altere os dados da tabela.

SHARE UPDATE EXCLUSIVE – Requerido pelo VACUUM , ANALYZE, CREATE INDEX CONCURRENTLY, e alguns ALTER TABLE.

SHARE – Requerido pelo CREATE INDEX (sem CONCURRENTLY).

SHARE ROW EXCLUSIVE – Este modo não é adquirido automaticamente por nenhum comando em PostgreSQL.

EXCLUSIVE – Este modo não é adquirido automaticamente sobre tabelas com nenhum comando em PostgreSQL

ACCESS EXCLUSIVE – Adquirido por ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, e VACUUM FULL – É também o modo default.

Conflicting Lock Modes

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

5.3.2 Row-level locks

Em adição aos *locks* sobre tabelas, existem *locks* sobre linhas de cada tabela. Estes podem ser partilhados ou exclusivos. Um *lock* exclusivo é automaticamente adquirido sempre que são alterados os conteúdos de uma linha. Um *lock* é mantido até uma transição terminar. Um *lock* sobre uma linha não afeta as consultas à mesma, apenas evita que haja mais que um escritor sobre a linha.

Para adquirir um exclusive *row-level lock* sem modificar a linha, seleciona-se a linha com SELECT FOR UPDATE.

Para adquirir a *shared row-level lock* sobre uma linha, seleciona-se a linha com SELECT FOR SHARE.

O PostgreSQL não se lembra de qualquer informação sobre a modificação de linhas em memória, por isso não a limite sobre o número de linhas bloqueadas num tempo só.

Em adição a *table* e *row locks*, *page-level share*/*exclusive locks* são usados para controlar os acessos de *read*/*writes* para páginas de tabela em *shared buffer pool*. Os *locks* são libertados de imediato depois da linha ser procurada ou atualizada.

5.3.3 Advisory locks

São chamados de *advisory locks*, porque o sistema não obriga o seu uso, o uso destas depende da aplicação. Estes *locks* podem ser uteis para estratégia de bloqueio que se enquadram de forma estranha sobre o modelo MVCC (*multiversion concurrency control*). Por exemplo, um uso comum dos *advisory locks* é emular estratégias de bloqueio pessimistas típicas dos chamados sistemas de gerenciamento dos ‘’flat file’’.

Vantagem de *advisory locks* são os seguintes:

1. São mais rápidas
2. Evita o MVCC bloat
3. E pode ser limpo automaticamente pelo servidor no final da sessão.

5.4 Deadlock detection

O uso excessivo de *locks* pode levar ao *deadlock*, quando existem duas ou mais transações, onde cada uma delas mantêm um *lock* que a outra requere. Por exemplo,

se a transação 1 adquire um exclusive *lock* sobre a tabela A e tenta adquirir um lock exclusivo sobre a tabela B, sabendo que a transação 2 já tem o exclusive *lock* sobre a tabela B e agora pretende um exclusive *lock* sobre a tabela A, então nenhuma delas pode proceder. O PostgreSQL deteta automaticamente situações de *deadlock* e resolve-as abortando uma das transações envolvidas e deixando uma completar.

A melhor defesa contra os *deadlocks* na sua generalidade é evita-los, tendo a certeza de que todas as aplicações que usam base de dados que adquiram *locks* múltiplos sobre objetos numa ordem consistente. Se duas transações atualizarem duas linhas na mesma ordem, nenhum *deadlock* poderia ter ocorrido.

É preciso também garantir que o primeiro bloqueio adquirido num objeto numa transação é o modo mais restritivo que será necessário para esse objeto. Se não for possível verificar isso com antecedência, então os *locks* podem ser manipulados *on-the-fly* por transações. Tentando que abortem devido a *locks*.

5.5 Locking e Índices

PostgreSQL fornece acesso *nonbloking read/ writes* a dados da tabela, estes acessos não são fornecidos para todos os métodos índice implementados no PostgreSQL.

Though PostgreSQL provides *nonblocking* read/write access to table data, *nonblocking* read/write access is not currently offered for every index access method implemented in PostgreSQL. Os vários tipos de índice são tratados como se segue:

Índices B-tree, GiST e SP-GiST.

Short-term share/exclusive page-level locks são usados para acessos de *read*/*write*. *Locks* são libertados de imediato depois de cada linha de índice ser procurado ou inserido. Estes índices fornecem maior concorrência sem condições de *deadlock*.

Índice Hash

Share/exclusive hash-bucket-level locks são usados para acesso de *read*/*write*. *Locks* são libertados depois do *bucket* ser processados na sua totalidade. Os Bucket-level *locks* fornecem melhor concorrência do que os index-level, mas o *deadlock* é possível porque os *locks* são mantidos durante mais tempo que as operações em índice.

Índice GIN

Short-term share/exclusive page-level locks são usados para acesso de *read*/*write*. Os *locks* são libertados de imediato depois de cada linha de índice ter sido procurada ou inserida. Mas deve ter em conta que que a inserção de um valor GIN-Indexed normalmente produz algumas inserções de chave índices por linha, então o GIN poderá fazer um trabalho substancial para a inserção de um único valor.

Neste momento, os índices B-tree oferecem a melhor performance em aplicações concorrentes, uma vezes que tem mais *feature* em relação aos índices *hash*, estes são recomendados para aplicações simultâneas que necessitam de dados de índice escalares. Quando se lida com dados não escalares, as B-trees não são muito uteis, e as GiST, SP-GIST ou GIN deverão de ser usados.

5.6 Verificação no final da transacção

O PostgreSQL permite ao utilizador usar o comado SET CONSTRAINTS, este define o comportamento de verificação de restrição dentro da transação corrente. Constrangimentos imediatos são verificados ao final de cada comando. Restrições diferidas não são verificadas até *commit* de transações. Cada restrição tem o seu próprio modo de imediata ou diferida.

Sintaxe:

SET CONTRAINT {ALL |name[,...]}{DEFERRED | IMMEDIATE}

5.7 Mesmo que o sistema falhe, será ele mantém a sua atomicidade e durabilidade?

Se a transação não for *committed*, esta não será visível a nenhuma outra transação.

Uma transação que aborta, normalmente colocara o seu pg_log para *aborted*. Mas, mesmo que o sistema falhe sem ter colocado o estado do pg_log para *aborted*, será seguro. Na próxima vez algum *backend* verifica o estado da transação, e observara se a *transação* foi marcada em progresso, mas que não corra em *backend*, deduzindo que se *crashou*, e coloca-se o estado para *aborted*.

As transações são só garantidas atomicamente se a página de disco escrita é uma ação atômica. Nos *hard drives* mais modernos isso é verdade se a página for um sector físico, mas a maioria das pessoas correm com páginas de disco configurados a 8K, que torna um pouco duvidoso mesmo que a página escrita de tudo ou nada.

O pg_log é seguro de qualquer maneira, já que lançamos apenas os bits, e os dois bits de status de uma transação deve ser no mesmo sector.Mas quando os tuplos se movem em uma página de dados, há um potencial para corrupção de dados em caso de falha de energia deve conseguir abortar a página de escrita a meio. Esta é uma razão para manter a página tamanhos pequeno.

Nota: O ficheiro de controlo pg_log contém 2 bits de estado por ID transação, com os possíveis estados, in *progress*, *commited*, *aborted*. O estabelecimento destes dois bits com o valor empenhado é a ação atômica que marca uma transação confirmada.

5.8 Savepoints

É possível controlar afirmações nas transações numa forma mais granular, usando os *savepoints*. Os *savepoints* permitem descartar de forma seletiva partes da transação,

enquanto se faz *commit* do resto. Depois de definir o *savepoint* com SAVEPOINT, pode-se fazer *roll back* para a *savepoint* com ROLLBACK TO, se necessário.

Depois do *rolling back* para um determinado *savepoint*, este último continua a ser definido, significando que se pode fazer *rollback* várias vezes até esse ponto.

Por outro lado, se houver a certeza que não vai ser preciso reverter para um ponto de salvamento novamente, este poderá ser liberado, podendo o sistema libertar alguns recursos.

Comando:

SAVEPOINT savepoint_name

A sintaxe do comando ROLLBACK TO SAVEPOINT é a seguinte:

ROLLBACK [WORK | TRANSACTION] TO [SAVEPOINT] savepoint_name

Em caso de *lock*, este normalmente é mantido até ao fim da transação. Mas caso esse *lock* seja criado depois de um *savepoint*, o *lock* será libertado de imediato se o *savepoint* for *rolled back* to.

Isto é consistente com os princípios do ROLLBACK, pois este cancela todos os efeitos dos comandos criados após o *savepoint* em causa.

Exemplo:

```
begin;
        insert into alunos values (13, 'Soraia Santos', '1993-11-21',"F", 1);
        savepoint first_savepoint;
        insert into alunos values (14, 'Miguel Antunes', '1993-11-29',"M", 1);
        rollback to savepoint first_savepoint;
        insert into alunos values (15, 'Jéssica Chaves', '1993-10-20',"F", 3);
commit;
```

Neste exemplo são adicionados os alunos Soraia Santos e Jéssica Chaves, deixando o Miguel Antunes uma vez que se fez rollback até ao savepoint antes da sua adição.

6 Bases de dados distribuídas

O sistema PostgresSQL não suporta bases de dados distribuídas, no entanto existem algumas soluções que implementam sistemas de bases de dados distribuídas como é o caso do Postgres-XC, Bucardo, PgCluster, Pgpool, Slony, entre outros.

Neste relatório o sistema de estudo será o Postgres-XC, que se trata de uma implementação assente no sistema PostgresSQL.

O Postgres-XC trata-se de um projeto *open-source*, tal como o PostgresSQL, onde é possível a utilização de um conjunto de servidores de bases de dados a funcionar agregadamente, de forma transparente para o cliente.

Do ponto de vista do cliente trata-se um servidor com um sistema de base de dados. A comunicação é feita com um servidor que tem o papel de coordenador do sistema.

Neste sistema as *queries* são realizadas de forma transparente para a aplicação. Para esta a interação com a base de dados é feita como se de um único servidor se tratasse. Uma vantagem desta abstração é que facilmente uma aplicação deixa de usar uma base de dados única para usar uma base de dados distribuída. Uma desvantagem é que a aplicação pode não estar preparada para se adaptar a eventuais falhas comuns aos sistemas distribuídos.

No caso de replicação as operações de escrita são realizadas em todos os servidores, enquanto a leitura é feita em apenas um dos servidores.

No caso de fragmentação, existe um servidor - coordenador - que é encarregue de pedir a execução das operações em cada um dos servidores, e recolher a informação por estes devolvida. Estas operações são realizadas em paralelo em cada um dos servidores. As operações de leitura e de escrita podem ser executadas em apenas um dos servidores, caso os dados a serem escritos ou lidos estejam apenas um dos servidores, não colocando assim esforço nos restantes servidores.

Este sistema garante a atomicidade das transações recorrendo à utilização do protocolo 2PC (Two-phase commit).

GTM (Global Transaction Manager)

Para o controlo de transações o Postgres-XC utiliza um componente gerir transações. O GTM executa num dos servidores.

Para cada transação o GTM cria um identificador, crescente, para distinguir a antiguidade entre transações, tem a informação de quando a transação foi iniciada, se foi *committed* ou se foi *aborted*.

Cada tuplo, a ser modificado, tem um conjunto de identificadores que indicam as transações que criaram ou apagaram o tuplo. Por exemplo se um tuplo foi criado por uma transação ativa, então esta não foi *committed* ou *aborted*, portanto a transação ignora o tuplo.

Cada transação obtém do GTM os *snapshots* necessários para garantir o nível de isolamento da transação. No caso de *Read committed* a transação obtém um *snapshot* para cada ação.

No caso de *Serializable* a transação obtém apenas um *snapshot* no início da transação.

Na execução do protocolo 2PC, além da existência de um coordenador, o GTM vai sendo notificado das transações *commited* e *aborted*.

O Postgres-XC não suporta a deteção de *deadlocks* portanto a forma de lidar com este tipo de situações é com recurso a *timeouts*. No entanto o valor do *timeout* é um valor difícil de definir, pois se for demasiado grande os desbloqueio de situações de *deadlock* é demorada, se for demasiado pequeno pode haver transações que nunca conseguem chegar ao fim por demorarem mais tempo que o estabelecido para *timeout*. Este sistema permite a consulta dos *locks* ativos através do comando *pg_locks*.

7 Outras funcionalidades

7.1 XML

O PostgreSQL permite que, através da utilização da linguagem XML, possa ser obtida informação da base de dados num formato XML, configurável pela aplicação/programador.

Desta forma é possível otimizar a construção de conteúdos direcionados para o desenvolvimento Web.

Seguem algumas funcionalidades desta integração com exemplos do seu uso.

Agregação/concatenação de linhas, criação de elementos. A criação de elementos pode ser feita de forma hierárquica como podemos verificar na *Query* 2.

Query 1:

```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;
```

Resultado: <foo>abc</foo><bar/>

Query 2:

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
    xmlelement(name abc),
    xmlcomment('test'),
    xmlelement(name xyz));
```

Resultado: <foo bar="xyz"><abc/><!--test--><xyz/></foo>

É ainda possível a utilização da linguagem XPath para obter informação XML da base de dados. Desta forma é possível tirar proveito das capacidades da linguagem XPath para fazer procesamento XML no momento da *query* à base de dados

Query 3:

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
        ARRAY[ARRAY['my', 'http://example.com']]);
```

Resultado:{test}

7.1.1 Linguagens Procedimentais

Quando existe a necessidade de executar múltiplas *queries* à base de dados, é necessária comunicação entre a aplicação e o sistema de base de dados. Esta comunicação pode ser causa de diminuição de performance da aplicação, pois introduz um *overhead*. As linguagens procedimentais são forma de evitar essa comunicação.

No PostgreSQL é possível a definição de procedimentos que podem ser executados pelo sistema de base de dados, sem necessidade de comunicação com a aplicação.

Para o PostgreSQL, atualmente, existem quatro linguagens procedimentais, em que é permitida a utilização de diferentes linguagens de programação para a implementação dos procedimentos. As linguagens são SQL (PL/pqSQL), C (PL/TCL), Perl (PL/Perl) e Python (PL/Python).

Existe ainda a possibilidade de criação de *triggers*, que permitem a invocação de funções/procedimentos em determinadas situações, previamente definidas.

A criação de *triggers* é possível através da linguagem SQL.

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [
OR ... ] }
  ON table
  [ FROM referenced_table_name ]
```

{ NOT DEFERRABLE | [DEFERRABLE] { INITIALLY IMMEDIATE | INITIALLY DEFERRED } }
[FOR [EACH] { ROW | STATEMENT }]
[WHEN (condition)]
EXECUTE PROCEDURE function_name (arguments)

Onde o event pode ser INSERT, UPDATE, DELETE ou TRUNCATE.

O padrão de acesso ODBC torna possível a compatibilidade entre diferentes sistemas de bases de dados. Desta forma não é necessário que uma aplicação implemente os métodos de acesso à base de dados, pois cada sistema tem a própria implementação que pode ser invocada através da interface padrão. Este padrão de acesso está disponível para o PostgreSQL.

7.1.2 Segurança

A nível de segurança, o PostgreSQL oferece um sistema de autenticação, com restrições de acesso para cada utilizador. É possível a restringir o acesso à base de dados para um determinado IP.

7.1.3 Envio de e-mail

Existe a funcionalidade de envio de *e-mails* a partir do sistema de base de dados. O pgMail é um sistema implementado em TCL para os sistemas PostgreSQL. Este sistema pode ser bastante interessante para o envio de *e-mail* despoletado por *triggers*, por exemplo: numa loja virtual, é possível fazer um envio de *e-mails* para clientes quando um determinado produto passou a ter *stock*.

O exemplo abaixo mostra como pode ser implementado um *trigger* que despoleta o envio de um *e-mail* quando um utilizador faz um pagamento.

```
create table orders (
        id integer,
        firstname varchar(200),
        lastname varchar(200),
        email varchar(200),
        paystatus varchar(1) default 'n'
);

create function checkordermail() returns opaque as '
DECLARE
        customerRec RECORD;
        textMessage text;
BEGIN
        select into customerRec * from orders where id = NEW.id;
        if customerRec.paystatus = ''y'' then
                textMessage := ''Thank you for paying your bill.  How sweet of you.
                        I love cake.  Dont you?'';
                perform                                                 pgmail(''Order
System<os@store.com>'',customerRec.email,''You paid.  How nice.'', textMessage);
        end if;
        return NEW;
END;' language 'plpgsql';

CREATE TRIGGER trgCheckOrderMail
        AFTER INSERT OR UPDATE ON orders FOR EACH ROW
        EXECUTE PROCEDURE checkordermail();
```

8 Conclusões

Terminamos aqui com o nosso estudo sobre o sistema PostgreSQL, este relatório serviu para ficarmos a saber sobre a capacidade e complexidade do mesmo.

Em relação a armazenamento e estrutura de ficheiros, vimos que tal como o Oracle, tem *Buffer* próprio tirando o facto de que o PostgreSQL usa o Buffer de E / S para mover dados de e para o disco.

Em termos de *partioning* o sistema da Oracle fornece a *composite* (*hash* + *range*) e a *hash*, já o PostgreSQL implementa só o *list* e range *partitioning*.

Em comparação aos índices vimos que o PostgreSQL suporta quatro tipos distintos de métodos de índice, sendo estes B-tree que por sua vez são as B+tree's, Hash, GiST e GIN, já o Oracle não implementa os dois últimos, mas implementa o índice Bitmap, o qual não existe externamente para o utilizador. O PostgreSQL suporta os índices *hash* mas desencoraja o uso devido a baixo performance, o Oracle por sua vez suporta organização com *hash* estático, mas não índices *hash*.

Ao contrário do que acontece no Oracle 11g, no PostgreSQL apenas é possível desligar o uso dos algoritmos de pesquisa e tipos de índices, sendo contudo o próprio PostgreSQL que escolhe dentro dos disponíveis qual utilizar.

Este sistema não suporta bases de dados distribuídas, ao contrário do sistema Oracle 11g. No entanto como foi mencionado o Postgres-XC suportam distribuição de bases de dados.

Este sistema suporta parte das funcionalidades que o Oracle 11g suporta para este tipo de sistemas. Este sistema suporta replicação e fragmentação de dados, em que as *queries* realizadas são transparentes para a aplicação. Ambos os sistemas garantem a atomicidade recorrendo ao protocolo 2PC. O PostgreSQL apesar de suportar *locks* globais, não tem ainda um mecanismo de deteção de *deadlocks*, no entanto permite definir *timeout* para a duração dos *locks*.

Tal como foi explicado, o Postgres-XC além dos coordenadores de cada servidor tal como nos sistemas Oracle 11g, tem ainda componente responsável por fazer a gestão de transações, GTM, que mantém a informação relativa às transações em curso e *snapshots* da base de dados.

Apesar de não suportar XQuery, como é o caso do Oracle 11g, este sistema suporta a utilização da linguagem XPath.

Tanto o PostgreSQL como o Oracle 11g, suportam a utilização de linguagens procedimentais e de *triggers*, para agilizar o tratamento da informação e o controlo da coerência dos dados.

9 Referências

http://www.PostgreSQL.org/docs/9.3

http://www.westnet.com/~gsmith/gregsmith/content/PostgreSQL/PostgreSQLBufferManagement.htm

http://www.westnet.com/~gsmith/content/PostgreSQL/InsideBufferCache.pdf

http://en.wikipedia.org/wiki/B%2B_tree

http://pgdocptbr.sourceforge.net/pg80/runtime-config.html

http://blog.csdn.net/spche/article/details/5186184

http://www.gqferreira.com.br/artigos/ver/indices-PostgreSQL

http://stackoverflow.com/questions/5083076/setting-constraint-deferrable-doesnt-work-on-PostgreSQL-transaction

http://PostgreSQLbr.blogspot.pt/2007/05/os-comandos-savepoint-rollback-to.htm

http://img1.wikia.nocookie.net/__cb20130122012448/postgresxc/images/6/66/PG-XC_Architecture.pdf