Sistemas de Base de Dados

PostgreSQL

Bruno Candeias (nº 42006)
Constantino Gomes (nº 41903)
David Lopes (nº 41874)

Professor:
José Júlio Alferes

Grupo: 09

SBD - 2013/2014

Conteúdo

1 Introdução

O PostgreSQL é um Sistema de Gestão de Base de Dados *open source* bastante completo, utilizado ambientes comerciais e académicos por todo o mundo e com as mais diversas configurações.

Este documento apresenta uma análise geral acerca de alguns dos componentes principais do PostgreSQL 9.3.

1.1 História do PostgreSQL

PostgreSQL, tive a sua origem no projecto Ingres, pelo professor Michael Stonebraker da Universiade da Califórnia em Berkeley. O projecto Ingres foi desenvolvido, entre 1977 e 1985, como um exercício pedagógico para a criação de um sistema de gestão de base de dados (SGBD) de acordo com a teoria clássica de base de dados relacionais.

Entre 1986 e 1994, o professor desenvolveu o projecto, chamado Postgres, com o objectivo de explorar os novos conceitos de *object relational*.

Em 1995, dois estudandes do professor Michael Stonebraker, Andrew Yu e Jolly Chen, substituiram a linguagem de interrogação *POSTQUEL* por uma versão extendida do SQL (*Structured Query Language*) e o projecto foi renomeado para Postgres95.

Em 1996, o projecto foi novamente renomeado e divulgado sobre o nome de PostgreSQL e passou a fazer parte da comunidade *open source*, deste então o tem mantido com contribuições de diversos programadores por todo o mundo.

Actualmente o PostgreSQL está na sua versão 9.3 e destaca-se pelas seguintes funcionalidades: *queries* complexas, chaves estrangeiras; *triggers*; *views*; integridade transacional; controlo de concorrência multi-versões.

Recentemente foi implementada a replicação de dados, suportando assim bases de dados distribuídas.

1.2 Organização do documento

Este documento está organizado da seguinte maneira: a secção 2 apresenta a organização interna de ficheiros no PostgreSQL. A secção 3 descreve os tipos de indexação e *hashing* existentes. A secção 4 analisa o processo de tradução e execução de uma *query* bem como os mecanismos de optimização implementados pelo PostgreSQL. A secção 5 descreve a gestão de transações e controlo de concorrência que o PostgreSQL implementa. Na secção 6 analisamos o suporte que o PostgreSQL oferece a base de dados distribuídas. Por fim, na secção 7 descrevemos outras características do PostgreSQL que achamos importante referir.

2 Armazenamento e Estrutura de Ficheiros

2.1 Sistema de Ficheiros

A abordagem do PostgreSQL para o sistema de ficheiro e armazenamento foi utilizar o sistema de ficheiros do sistema operativo, contrariamente ao sistema Oracle 11g que implementa o seu próprio sistema de ficheiros. Este facto faz com que o seu desempenho possa diminuir devido a duplicações dos sistema de caches entre o PostgreSQL e o sistema operativo. Deste design resulta o uso de *double buffering*, onde os blocos de disco são primeiro colocados pelo sistema de operativo para a sua cache, no *kernel*, e depois copiados para o *buffer* do PostgreSQL.

2.1.1 Organização de Ficheiros

O PostgreSQL costuma criar uma diretoria `PGDATA` para guardar quer informação de configuração quer os dados para aquele *database cluster*. Um *database cluster* é uma coleção de bases de dados geridas por uma única instancia do servidor de PostgreSQL. No contexto do sistema de ficheiros *database cluster* é uma única diretoria sobre a qual todos os dados serão armazenados. Nos sistemas baseados no sistema UNIX esta costuma-se encontrar localizada no caminho absoluto `/var/lib/pgsql/data`, mas pode ser criada noutra directoria com o comando `initdb`. A organização dos ficheiros é feita através de subdirectorias que são apresentadas na tabela 1.

O PostgreSQL permite aos administradores definirem as localizações deste *database cluster* no sistema de ficheiro que representam a base de dados (tabelas, índices, ...) pelo do uso de *tablespaces*. Este conceito adiciona flexibilidade à base de dados uma vez que diferentes tabelas podem estar associadas a diferentes *tablespaces*. Por exemplo, quando uma partição do disco onde existe um *database cluster* ficar sem espaço, o comando *tablespace* pode ser criado numa partição diferente e usado até o sistema ser reconfigurado. Por outro lado, permite aos administradores usar os seus conhecimentos da utilização dos objectos da base de dados para a otimizar.

Por omissão, as base de dados e tabelas estão associadas ao *tablespace* *pg_default*, este parâmetro pode ser mudado alterando o *default_tablespace* na configuração do servidor.

Item	Descrição
`PG_VERSION`	Ficheiro que contém o número da versão do PostgreSQL
`base`	Subdiretoria que contém as subdiretorias de cada base de dados
`global`	Subdiretoria que contém as tabelas comuns do *cluster*
`pg_clog`	Subdiretoria que contém informação sobre o estado do *commit* das transações
`pg_multixact`	Subdiretoria que contém informação sobre o estado de multi-transações
`pg_notify`	Subdiretoria que contém informação sobre o sistema de notificações
`pg_serial`	Subdiretoria que contém informação sobre transações surrealizáveis já submetidas
`pg_snapshots`	Subdiretoria que contém informação sobre os *snapshots* exportados
`pg_stat_tmp`	Subdiretoria que contém ficheiros temporários utilizados pelo subsistema de estatísticas
`pg_subtrans`	Subdiretoria que contém dados referentes ao estado de subtransações
`pg_tblspc`	Subdiretoria que contém *symbolic links* para *tablespaces*
`pg_twophase`	Subdiretoria que contém ficheiros de estado para transações preparadas
`pg_xlog`	Subdiretoria que contém ficheiros WAL (*Write Ahead Log*) referentes ao *logging* das transações
`postmaster.opts`	Ficheiro que contém as configurações com que o servidor foi inicializado na última vez
`postmaster.pid`	Ficheiro de *lock* que contém o PID corrente, diretoria corrente do *database cluter*, tempo inicio do processo servidor, endereço porta de rede do processo servidor e ID da memória partilhada

Tabela 1: Conteúdo da diretoria `PGDATA`

O PostgreSQL limita o tamanho das tabelas e índices a 1GB, valor este que pode ser alterado com a opção `with-segsize`, quando o tamanho é ultrapassado o ficheiro contendo a informação tabelas e índices é dividido em segmentos de 1GB, para evitar assim problemas de compatibilidade com outras plataformas, no que diz respeito as restrições de tamanho.

2.2 Organização de Tuplos e Tabelas

No que diz respeito a representação de tabelas, o PostgreSQL utiliza um esquema de páginas com tamanho fixo por cada página de 8KB, no entanto este tamanho pode ser modificado a quando da compilação, e não permite que um tuplo se encontre dividido em mais que uma página, impedindo assim que sejam guardados campos com grandes dimensões. As páginas seguem a estrutura do tipo *slotted-page*, representada na figura 1, o que permite a organização dos registos armazenados em cada página. Para guardar os índices a primeira página e geralmente guardada para guardar informação de controlo, sendo que as restantes poderão ser de diferentes tipos.

Os tuplos estão organizados num *heap* e são identificados por tuplo ID (TID), que especifica a página dentro do ficheiro que contém a relação e ainda o índice do *line pointer* dentro da página. O comprimento dos tuplos é normalmente limitado pelo tamanho das páginas, o que dificulta o armazenamento de grandes tuplos. Para contornar esta limitação, os tuplo podem ser comprimidos e/ou divididos em múltiplas linhas, usando a técnica TOAST.

O PostgreSQL à semelhança do Oracle 11g implementa *slotted pages*, com o objectivo de permitirem registos de tamanho variável.

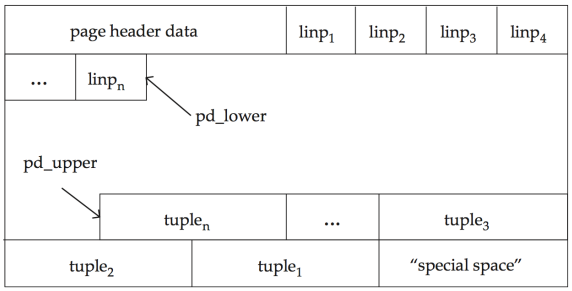

Figura 1: Formato *Slotted Page* PostgreSQL

2.2.1 *Free Space Map* e *Visibility Map*

Cada *heap* e índice tem associado um *free space map*, para monitorizar o espaço livre dentro destes. Esta estrutura consiste numa árvore binária em que cada folha aponta para o espaço livre. O *visibility map* mapeia as páginas que se encontram visíveis para todas as transações a ocorrer em determinado momento. Utiliza apenas um bit por *heap page*, que caso esteja a um todos os tuplos dessa página se encontram visíveis para o conjunto de transações, significando que a página não contém tuplos que devem ser apagados pelo *garbage collector VACCUM*.

2.2.2 *The Oversized-Attribute Storage Technique* (TOAST)

Devido ao PostgreSQL usar páginas de tamanho fixo e não permitir que tuplos estejam dividir por várias páginas, impossibilita o armazenamento de grandes tuplos. Por forma a superar esta limitação, foi necessário implementar este mecanismo TOAST, que para atributos de tuplos de grandes dimensões estes são comprimidos e/ou divididos em múltiplas linhas físicas, de forma transparente para o utilizador. Ainda assim o TOAST impõe que os dados tenham o tamanho máximo da slotted-page, normalmente 1GB.

Os atributos que podem utilizar TOAST que podem ser configurados para utilizar uma de quatro estratégia fornecidas:

- `PLAIN` - previne tanto a compressão como a decomposição, sendo a única opção para colunas com tipos de dados que não necessitem de TOAST;
- `EXTENDED` - permite a compressão e a decomposição, sendo a estratégia utilizada por omissão para a maioria dos tipos de dados compatíveis com TOAST. A compressão será o primeiro mecanismo utilizado e se o tuplo ainda for demasiado grande é utilizada a decomposição em multiplas linhas;
- `EXTERNAL` - permite a decomposição mas não a compressão, sendo a estratégia que otimiza a manipulação de dados de texto com custo na utilização de mais espaço em disco;
- `MAIN` - efetua a compressão mas só em ultimo recurso efetua a decomposição dos dados.

.

2.2.3 *Clustering*

A técnica de *clustering* pode ser definida como a junção de dados em espaços contínuos dos componentes de armazenamento, tornando assim as operações de acesso à informação mais rápidas, por exigirem menos operações de *I/O*.

O PostgreSQL implementa apenas o *table clustering* que permite fazer *cluster* a uma determinada tabela recorrendo a um índice, através do comando `CLUSTER`. Quando a tabela é *clustered*, é reordenada em disco pelo índice atribuído. Qualquer atualização realizada à tabela após o *clustering*, deixa de usufruir desta organização.

```
CLUSTER index_name ON table_name
```

Listing 1: Sintaxe comando *Cluster*

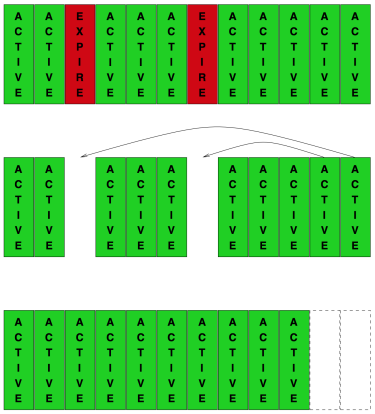

Figura 2: Representação execução `VACCUM FULL` sobre uma tabela

O PostgreSQL não suporta *multitable clustering*, no entanto o Oracle 11g suporta, permitindo uma melhor performance em *queries* que frequentemente usam juncões de tabelas que estão *clustering*.

2.2.4 *Garbage-collect* VACUUM

No PostgreSQL os tuplos que ficam obsoletos devido a uma atualização ou que são removidos através de simples operações, não são fisicamente removidos da tabela correspondente. O mecanismo VACUMM recupera o espaço de armazenamento ocupado por esses tuplos mortos, permitindo fazer uma limpeza (*garbage-collector*) e opcionalmente uma análise à base de dados. Como o espaço ocupado por tuplos mortos apenas e reaproveitado após ser aplicado o VACUMM este deve ser aplicado periodicamente, especialmente em tabelas que são atualizadas frequentemente.

O comando `VACCUM` oferece dois modos de operação:

- `Plain VACCUM` - esta versão simplesmente procura por tuplos que já não são necessários e liberta o seu espaço. Este comando pode ser executado em paralelo com operações de escrita e leitura sobre a tabela, uma vez que um *exclusive lock* não é obtido.

- `VACCUM FULL` - esta versão faz uma limpeza mais exaustiva sendo que ao apagar tuplos pode mover outros tuplos da tabela para a compactar e fazer com que esta ocupe menos blocos de disco possíveis. Esta versão é mais lenta e requer que seja obtido o *lock* das tabelas sobre as quais opera.

2.2.5 Partições

Em PostgreSQL é possível particionar uma tabela em sub-tabelas, passando a existir uma tabela lógica que representa a tabela completa e um conjunto de sub-tabelas físicas onde estão guardados os dados. Atualmente o PostgreSQL suporta o particionamento de tabelas através de herança, ou seja, cada partição deve ser criada como tabela filha de uma tabela pai, que normalmente esta vazia, existindo apenas para representar todos os dados.

Esta funcionalidade é principalmente útil para dividir tabelas de grandes dimensões, trazendo benefícios no desempenho das *queries*, especialmente nos casos em que os dados acedidos com mais frequência estão numa partição ou num pequenos conjunto destas; reduz o tamanho dos índices tornando mais provável que estes caibam em memória. Quando as *queries* ou *updates* são sobre um grande parte da partição a performance pode ser melhorada utilizando *sequential scan* em vez de índices e acessos aleatórios sobre toda a tabela. Permite ainda que dados não usados regularmente sejam transferidos para uma partição num disco mais lento.

São suportadas duas forma de particionameto no PostgreSQL:

- *Range Partitioning* - A tabela é particionada em intervalos definidos por uma coluna chave ou um conjunto de colunas, sem sobreposição entre os intervalos de valores atribuídos às diferentes partições. Um exemplo possível deste particionamento é com base num intervalo de datas;

- *List Partitioning* - A tabela é dividida explicitamente por uma lista de valores chave que devem aparecer em cada partição. Um exemplo possível deste particionamento é dividir um tabela com base numa lista de países.

Relativamente ao Oracle 11g, este permite adicionalmente a partição de tabelas por *Hash Partitioning* e *Composite Partitioning*.

2.3 *Buffer Management*

O PostgreSQL tem a sua própria estão de *buffers*, mas ao contrário de outras base de dados, não prefere que a maioria da memória do sistema seja alocada para si. A maioria das leituras e escritas na base de dados são feitas utilizando chamadas ao sistema do sistema operativo, permitindo assim que o sistema operativo utilize o seu próprio modelo de cache, normalmente do tipo *Least Recently Used* (LRU).

Apesar do PostgreSQL confiar no sistema de cache do sistema operativo para gerir as páginas que são escritas e lidas regularmente, por exemplo para os ficheiros `pg_clog` que contém o estado do *commit* as transações, este utiliza também um *buffer* de memória partilhada. A principal razão

para a utilização deste *buffer* é que assim consegue ter melhor performance que o sistema operativo.

O algoritmo utilizado para a gestão dos *buffers* é chamado *clock-sweep algorthim*. Este algoritmo mantém um contador da utilização de cada *buffer* pela sua "popularidade". Quanto maior o contador menor a probabilidade desses *buffers* deixarem a cache.

Se a base de dados tiver dados com um elevado uso, é provável que os dados serão melhor utilizados se ficarem no *buffer* da base de dados do que no sistema operativo, porque o LRU não considera o número de vezes que uma *cache entry* é acedida, e em pesquisas sobre grandes tabelas pode começar a descartar *cache entry* úteis.

O *buffer* é um vector composto por várias entradas, em que cada entrada aponta para um bloco de dados de tamanho 8KB e possui uma etiqueta que identifica o ficheiro que essa entrada esta a fazer *buffering* e qual o bloco do ficheiro que ele contém. O *buffer* também e composto por uma série de *flags* que identifica o estado da informação de cada bloco: *pinned* que indica que o bloco está a ser usado por um processo e este só fica livre quando o processo acabar a sua execução e *dirty* que indica que a informação já foi modificada desde que foi lida do disco (útil para se saber que páginas necessitam de ser atualizadas).

3 Indexação e *Hashing*

No PostgreSQL um índice é uma estrutura de dados que fornece um mapeamento entre um determinado predicado e um conjunto de tuplos duma tabela. Neste sistema a sintaxe geral para a criação de um índice é a seguinte:

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

3.1 Tipos de Índices

O PostgreSQL apresenta os seguintes tipos de índices:

B-Tree

Estes índices são os índices utilizados por defeito, são baseados nas *B-link trees* de *Lehmanand e Yao*.

O *planner* de *queries* íra considerar utilizar estes índices em *queries* de igualdade ou de intervalos sobre dados ordenados, por exemplo envolvendo os operadores: `<, <=, =, >=, >, BETWEEN, IN, IS NULL, IS NOT NULL`

Podem ainda ser utilizados em *queries* envolvendo operadores `LIKE` e `~` se o padrão procurado for constante ou corresponder ao ínicio da *string*, por exemplo `coluna LIKE "foo%"` ou `coluna ~ "^foo"`.

Hash

Este tipo de índices em PostgreSQL é implementado usando *linear hashing*. Estes índices apenas lidam com comparações simples de igualdade, ou seja, o optimizador apenas íra considerar utilizar este índice em comparações utilizando o operador `=`.

Estes índices demonstraram um desempenho de pesquisa equivalente as B-trees, mas possuem um tamanho e custos de manutenção consideravelmente maiores, são os únicos índices no PostgreSQL que não suportam recuperação de falhas, devido a não utilizarem o mecanismo WAL.

Por estes motivos é quase sempre preferível utilizar índices B-trees em vez de índices de hash.

GiST

GiST significa *Generalized Search Tree* e trata-se de uma estrutura de dados e uma *API* que permite que vários tipos de índices sejam

implementados, por exemplo *B+ tree*, *R-tree* entre outros. Este tipo de índice permite que os especialistas possam efetuar afinações nas bases de dados, de forma a melhorar o desempenho desta sem terem de se preocupar com detalhes internos, como o gestor de *locks* e WAL. Este tipo de índice proporciona ainda a possibilidade de implementar índices que suportem pesquisas ordenadas (*nearest-neighbor searches*).

SP-GiST

SP-GiST é a abreviatura de *space-partitioned GiST*. Tal como GiST oferecem suporte a implementações de várias estratégias de pesquisa em estruturas de dados não equilibradas, diferenciado-se deste o suporte a arvores não equilibradas como *radix trees*, *quadtrees*, entre outras.

GIN

Os índices GIN, *Generalized Inverted Index*, lidam com valores aos quais corresponde mais do que uma chave (como por exemplo, vectores). São frequentemente usados para pesquisa em documentos, onde uma dada palavra assume o papel de chave no índice. Tal como o GiST, o GIN suporta diferentes estratégias de indexação definidas pelo utilizador.

O PostgreSQL oferece uma implementação que suporta a execução em *queries* que utilize vectores uni-dimensionais em que os operadores sejam: `<@, @>, =, &&`.

3.2 Variações de Índices

O PostgreSQL para além dos índices acima apresentados, implementa algumas variações mais complexas que são úteis, para aumentar o desempenho das *queries*.

Índices Multicoluna

Os índices multicoluna são uteis para utilização em *queries* que sejam efetuadas sobre vários atributos, por exemplo quando um atributo representa uma categoria e outro a respectiva subcategoria. Este tipo de índices são suportadas por índices B-tree, GiST e GIN e podem indexar até 32 colunas.

Retomando o exemplo anterior podemos criar um índice da seguinte forma:

```
CREATE INDEX i_category ON t (category,subcategory);
```

Este índice poderia ser utilizado para as seguintes *queries:*

```
SELECT * FROM t WHERE category='x' and subcategory='y';
SELECT * FROM t WHERE category='x';
```

Mas não é possível utiliza-lo para uma *querie* que onde se procura informação pela subcategoria, como por exemplo:

```
SELECT * FROM t WHERE subcategory='y';
```

Índices únicos

Estes índices garantem que numa coluna de uma tabela não existe mais de um índice com o mesmo valor. Este tipo de índice é útil por duas razões: integridade dos dados, evitando duplicação do atributo, e desempenho do tempo das consultas. Um caso particular deste tipo de índices são as chaves primárias. No PostgreSQL apenas os índices B-tree podem ser utilizado com indices únicos.

Este tipo de indice pode ser instanciado sobre uma tabela da seguinte forma: `CREATE UNIQUE INDEX name ON table (column [, ...]);`. De notar que este comando é considerado um má forma de criação deste índice e deve ser evitada. A maneira preferida para adicionar um índice único é adicionar à tabela uma restrição da forma:

```
ALTER TABLE table ADD CONSTRAINT name_constraint
UNIQUE (column [, ...]);
```

Índices em Expressões

São índices ao qual se efetua a aplicação de uma computação sobre um ou mais atributos dos tuplos. Estes índices são úteis quando as expressões são pesadas e consomem algum tempo na sua execução, permitindo o acesso rápido aos dados pretendidos com base em valores pre-calculados. Este tipo de índice torna algumas leituras mais rápidas mas é de manutenção custosa, porque a expressão tem que ser avaliada cada vez que uma linha é inserida ou modificada.

Um exemplo da utilização destes índices é o suporte para comparações *case-insensitive*.

```
SELECT * FROM test1 WHERE lower(col1) = 'value';

CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

Índices Parciais

Um índice parcial é um índice mantido apenas para um subconjunto de colunas de uma tabela. Estes índices são definidos utilizando uma expressão denominada predicado e normalmente são utilizados para

diminuir o tamanho do índice de forma a aumentar a velocidade de pesquisa.

Estes índices normalmente não indexam toda a tabela, devido ao facto de utilizarem a cláusula `WHERE`, por exemplo considerando pretendemos seguir um certo número de contas de clientes para um processamento especial (clientes VIP), podemos ter uma *flag* "vip"*boolean* nos seu tuplos. Sendo assim podemos criar um índice parcial da seguinte forma:

```
CREATE INDEX accounts_interesting_index ON accounts
WHERE vip IS true;
```

A partir deste momento quando quisermos fazer algum processamento sobre estes clientes o *planner* poderá utilizar este índice em vez de verificar para todos os tuplos se são clientes "vip".

3.3 Combinação de Índices

Numa *querie* que utiliza apenas um índice, apenas podemos usar cláusulas `WHERE` cujas interrugações estejam juntas pela cláusula `AND`, por exemplo `WHERE a=4 AND b=6`, mas não conseguimos utilizar o índice directamente para `WHERE a = 5 OR b = 6`.

O PostgreSQL consegue combinar vários índices para resolver casos que não podem ser implementados com um só índice. Estas combinações de índices podem ser entre índices diferentes ou entre o mesmo índice. Por exemplo, no caso de uma pergunta ter uma cláusula `WHERE x = 42 OR x = 47 OR x = 53 OR x = 99` é possível executar um *scan* no índice por cada igualdade e, posteriormente, juntar os resultados com uma operação *OR*, de modo a obter o resultado final. Outro caso de interesse é aquele em que as colunas x e y tem um índice cada uma e é feita uma pergunta com a cláusula `WHERE x=5 AND y=6`. Neste caso seriam utilizados separadamente os índices de cada coluna, juntando-se o resultado final com uma operação *AND*.

Para combinar os índices, o PostgreSQL utiliza *bitmaps* que indicam as localizações das linhas das tabelas que estão de acordo com a condição da pesquisa. Os *bitmaps* são então cruzados através de operações *AND* ou *OR*, obtendo-se todas as localizações das linhas do resultado final. Para obter os dados, as linhas são visitadas por ordem física porque os *bitmaps* não conservam a ordenação dos índices. Caso o resultado tenha que ser ordenado devido a uma cláusula `ORDER BY`, será necessário efetuar uma operação de ordenação.

3.4 Diferenças ao Oracle SQL

A grande diferença entre Oracle SQL e PostgreSQL na indexação é a incorporação dos índices GIN e GiST, que não existem no Oracle SQL.

4 Processamento e optimização de perguntas

Esta secção oferece uma visão geral acerca da estrutura interna do *backend* do PostgreSQL. São identificadas e analisadas todas as fases do processamento de uma *query*, desde o momento que é invocada pelo utilizador, até ao momento da sua execução. São também identificados os mecanismos que o PostgreSQL oferece para a execução de queries e para a sua optimização.

4.1 Processamento de perguntas

Quando um utilizador executa uma *query*, esta é enviada pela aplicação cliente para o servidor[1], onde será então processada. O pedido será entregue pelo processo `postgres` a um processo alternativo, que ficará responsável pela conexão estabelecida. É a partir desse processo que a *query* será processada. O processamento de uma *query* dentro do servidor de PostgreSQL passa por várias etapas, representado na figura 3. Cada fase será brevemente discutida de seguida.

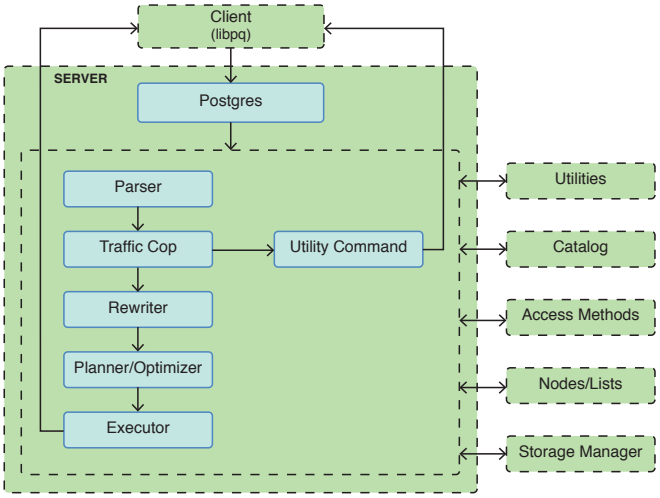

Figura 3: Etapas do processamento de uma query.

4.1.1 Parser

A fase de *parsing* consiste em duas partes: i) *parsing* da expressão, ii) transformação da estrutura de dados criada na primeira parte . Na primeira parte, é feita uma análise lexográfica à *query* SQL, onde são identificadas as *keywords* e os identificadores do SQL. O *parser* recebe o resultado

[1]O servidor pode ser local ou não; caso não seja, o pedido é enviado através da rede. A biblioteca `libpq` é responsável por enviar o pedido e receber os resultados.

devolvido pelo *lexer* e, através de um conjunto de regras gramaticais, determina se a síntaxe da expressão está correcta. Caso a expressão esteja sintaticamente correcta, é criada uma *parse tree* que irá ser usada de seguida. Na segunda parte, são consultados os catálogos do sistema para aferir que tabelas, funções ou operadores são referidos pela *query*. O resultado deste processo é representado através de uma estrutura chamada *query tree*. Uma *query tree* é meramente uma representação interna de uma expressão SQL, onde são guardadas as componentes da *query*: tipo de comando (`SELECT`, `INSERT`, ...), a lista de relações usadas na *query*, etc. O PostgreSQL oferece a possibilidade de ver as estruturas das *query trees* no *log* do servidor através dos parâmentros `debug_print_parse`, `debug_print_rewritten`, ou `debug_print_plan`.

Esta divisão do processo em duas partes é benéfica porque, a consulta dos catálogos do sistema é feita com recurso a transacções. Assim, a verificação de que a *query* está bem escrita – sintaticamente correcta – functiona como um mecanismo de segurança; não era desejável que expressões incorrectas escritas por utilizadores e que incorressem em erros, sobrecarregassem o servidor.

4.1.2 *Traffic Cop*

Antes de ser passada à próxima fase, a *query* é analisada por um módulo, *Traffic Cop*, determina se se trata de uma *query* utilitária, i.e. que não necessita de um tratamento complexo, tais como `CREATE TABLE`, `ALTER`, ou `COPY`, ou se se trata de uma *query* complexa, que envolva escritas ou leituras de dados – `SELECT`, `INSERT`, `UPDATE` ou `DELETE`. Caso o processamento da *query* seja trivial, este é executado de imediato por um módulo destinado ao tratamento deste tipo de queries. No caso de uma *query* que exija um tratamento mais complexo, a *query tree* é passada ao *rewrite system*, que irá consultar os catálogos do sistema para saber se existem regras definidas pelo utilizador que deverão ser aplicadas à *query*.

4.1.3 *Rewrite System*

É neste sistema que o PostgreSQL implementa o seu suporte para regras (*rules*). Nos outros sistemas de bases de dados, as regras são habitualmente procedimentos ou *triggers* armazenados. No entanto, o PostgreSQL funciona de maneira diferente, uma vez que modifica a *query* do utilizador de acordo com as regras definidas pelo utilizador.

O PostgreSQL implementa as *views* usando o *rewrite system*. De facto, para o *parser* uma *view* e uma tabela são a mesma coisa – uma relação – o que significa que no catálogo se sistema, é mantida a mesma informação. Como resultado, para o PostgreSQL não existe diferenças entre as duas *queries* abaixo apresentadas (assumindo que `myview` tem os mesmos atributos

que `mytab`).

```
CREATE VIEW myview AS
SELECT * FROM mytab;
```

```
CREATE TABLE myview ;
CREATE RULE "_RETURN" AS ON
    SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

4.1.4 *Planner/Optimizer*

Depois de a *query tree* ser reescrita, é sujeita à fase de planeamento e optimização. Cada *query* SQL pode ser executada de várias maneiras, portanto são gerados todos os planos de execução possíveis, i.e. que conduzam ao mesmo resultado. Cada plano é representado por uma estrutura interna, uma *plan tree*, que é essencialmente uma árvore de operações relacionais, como está demonstrado na figura 4. Para o fazer, o *Planner* usa estatísticas relativas à base de dados, mantidas pelo sistema, que depende de factores como o número de tuplos das relações e as operações de I/O – abordado com mais detalhe na secção 4.4 – para seleccionar qual o plano que incorre no custo mais reduzido.

No entanto, o plano de execução criado pode não ser óptimo; se a examinação de todas as possíveis execuções de uma *query* consumir muito tempo ou espaço. Para o executar numa quantidade de tempo aceitável, o PostgreSQL usa um *Genetic Query Optimizer*. Habitualmente isto acontece sempre que o número de junções excede um dado threshold (configurável através da varíavel `geqo_threshold`) – ver anexo A. A geração de planos será abordada mais em detalhe na secção 4.2.

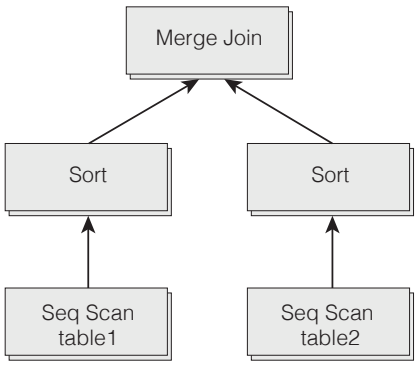

Figura 4: Exemplo de um plano de execução simples.

4.1.5 *Executor*

O *executor* recebe o plano gerado pelo optimizador e processa-o recursivamente até chegar ao resultado final. O seu modo de funcionamento baseia-se num mecanismo *demmand-pull pipeline*, ou seja, cada vez que um nó do plano é invocado, deverá devolver um tuplo adicional, ou notificar que já não tem mais tuplos para devolver. Aproveitando o exemplo da figura 4, antes que qualquer *merge* seja feito, é necessário obter dois tuplos (um de cada sub-árvore). Então o executor vai invocar-se a si próprio para processar as sub-árvores (começando pela esquerda). Quando o `MergeJoin` tiver dois tuplos, testa se os pode juntar e, caso seja possível, devolve de imediato *Eventually*[2], uma das sub-árvores (ou ambas) deixaram de ter tuplos para devolver, e o `MergeJoin` irá devolver `NULL`, indicando assim que não poderão ser formadas mais junções.

4.2 Geração de planos possíveis

Como já foi dito, cada plano de execução gerado pelo *planner* é uma estrutura onde cada nó é, na realidade, uma operação relacional. A maneira como a operação será executada na base de dados tem também de ser especificada, uma vez que diferentes abordagens poderão incorrer em diferentes custos. O PostgreSQL implementa vários algoritmos de leitura, junção e ordenação.

4.2.1 Métodos de acesso

Para adquirir os dados do disco, o PostgreSQL usa os seguintes métodos:

Sequential Scan
: Leitura sequencial do ficheiro, desde o bloco inicial ao final. Neste caso, a leitura pode ser sujeita a um filtro: para cada tuplo que lê, testa uma qualquer condição e só devolve os tuplos que a satisfizerem. Isto acontece quando existem cláusulas `WHERE`.

Index Scan
: Permite a pesquisa de acordo com predicados de igualdade ou de intervalo. O operador processa um tuplo de cada vez, lendo uma entrada do índice e obtendo o correspondente tuplo. No pior caso, este método pode resultar em leituras aleatórias no disco.

Bitmap Index Scan
: Esta abordagem reduz o perigo de leituras aleatórias. Isto é conseguido porque a abordagem tem duas fases: primeiro lê todos os índices e guarda os identificadores dos tuplos da *heap* num *bitmap*; na segunda

[2] A não ser confundido com *eventualmente*, que no seu sentido infere incerteza.

fase, adquire os correspondentes tuplos da *heap* sequencialmente. Isto garante que cada página da *heap* é acedida apenas uma vez.

4.2.2 Métodos de junção

O PostgreSQL suporta três métodos de junção:

Nested Loop Join
Apenas usado se a relação de "dentro"puder ser lida com o método *index scan*

Merge Join
Cada relação é ordenada pelos atributos antes de a operação de junção começar. Depois as relações são lidas em paralelo e os tuplos que corresponderem são combinados. Este tipo de join é mais atractivo porque cada relação tem de ser lida apenas uma vez. A ordenação pode ser alcançada usando uma fase de ordenação dos tuplos lidos, ou pela leitura de forma ordenada, usando um índice ou a chave do join.

***Hash Join*:**
A relação é lida e carregada para uma *hash table*, usando os atributos da junção como chaves.

4.2.3 Métodos de ordenação

O PostgreSQL usa o algoritmo *quicksort* para ordenar pequenas quantidades de dados em memória. Caso não seja possível, é usado o external-merge, baseado no algoritmo proposto por Knut [9]. O *input* é dividido em *runs*, onde será ordenado, e depois juntado em várias durante várias fases. As *runs* iniciais são criadas usando *replacement selection* recorrendo a uma árvore de prioridades (em vez de uma estrutura de dados que fixa o número de registos em memória), porque o PostgreSQL lida com tuplos que variam consideravelmente em tamanho e, desta forma, procura garantir a máxima actualização do espaço de memória configurada para realizar a ordenação. Este espaço de memória pode ser configurado com o parâmetro `work_mem`.

4.3 Visualização do plano de execução

É possível ver o plano que o PostgreSQL gera para resolver uma *query*. Para o fazer basta usar o comando `EXPLAIN` antes de cada *query*.

Vejamos o seguinte exemplo:

```
EXPLAIN SELECT * FROM tenk1;

                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

Como a *query* não inclui nenhuma cláusula `WHERE`, é necessário ler todos os tuplos da relação, então o *planner* escolheu um *sequential scan*. Caso existisse uma cláusula `WHERE`, era aplicado um `FILTER` no `Seq Scan`[3]. Os valores apresentados pelo plano são os seguintes: *i*) custo do arranque (i.e. tempo dispendido para que a fase de output seja iniciada), *ii*) custo total estimado, *iii*) número linhas que vão ser devolvidas por este nó (estimado) e *iv*) comprimento estimado dos tuplos devolvidos por este nó (estimado, em bytes) .

Os custos são calculados com base na parameterização do *planner* (ver o anexo A). Habitualmente, os custos são calculados com base no custo de obter uma página do disco: o parâmetro `seq_page_cost` é definido a `1.0` e todos os outros parâmetros são relativos a este.

É possível verificar qual a precisão das estimativas do *planner* através da opção `ANALYZE`. Com esta opção, o comando `EXPLAIN` executa realmente a *query* a *query* e mostra os verdadeiros valores obtidos:

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;

                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7000 width=244) (actual
 time=0.016..5.107 rows=7000 loops=1)
   Filter: (ten < 7)
   Rows Removed by Filter: 3000
 Total runtime: 5.905 ms
```

É relevante notar que os tempos calculados pelo `EXPLAIN ANALYZE` podem divergir da execução normal da *query*. Como não são enviados dados para o client, os custos de operações de I/O ou de transmissão de rede não são contabilizados. Para além disso, pode verifiar-se algum *overhead* em máquinas com um `gettiemofday()` lento. Para medir este *overhead* no sistema, o PostgreSQL oferece a ferramenta `pg_test_timing`.

4.4 Estatísticas usadas pelo *Planner*

Como já vimos, o *planner* precisa de estimar o custo das operações, de forma a que seja possível optimizar o custo do plano de execução.

Um componente das estatísticas que o sistema usa é o número total de entradas em cada tabela e índices, bem com o número de blocos ocupados por cada tabela e índices. Esta informação é mantida na tabela `pg_class`, nos atributos `reltuples` e `relpages`. Tomemos o exemplo que está em [3]:

[3]Notar que uma filtragem não invalida a necessidade de ler todos os tuplos, portanto apesar de o número de tuplos devolvidos poder baixar, o custo permanecer-se-ia inalterável.

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';
```

```
       relname        | relkind | reltuples | relpages
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      358
 tenk1_hundred        | i       |     10000 |       30
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30
(5 rows)
```

Podemos ver que a relação `tenk1` possui 10000 tuplos, como os seus índices, mas que no entanto (e sem surpresa), ocupam significativamente menos.

É importante notar que por questões de eficiência, estas estatísticas não são actualizadas *on-the-fly*; somente as operações `ANALYZE`, `VACUM` e outros como o `CREATE INDEX` provocam esta actualização.

As *queries* habitualmente querem apenas tuplos que satisfaçam uma dada propriedade, restrindo assim os tuplos que deverão ser examinados. Esta selectividade incorre também em custos. A informação usada nesta tarefa está no catálogo `pg_statistic` e só pode ser actualizada pelas operações `ANALYZE` e `VACCUM ANALYZE` e só pode ser lida em modo *superuser*. Existe uma alternativa mais legível – a view `pg_stats`.

4.5 Mecanismos de avaliação de operações complexas

4.5.1 *Pipelining*

O PostgreSQL recorre a *pipelining* sempre que possível. O esquema de *pipelining* que o PostgreSQL implementa é *demand driven*, ou seja, o sistema realiza pedidos do topo do *pipeline* – como vimos na secção 4.1.5.

4.5.2 Paralelização

Apesar do PostgreSQL suportar paralelismo do lado do cliente (uma aplicação pode ter várias conexões simultâneas e geri-las assincronamente), do lado do servidor o nível de paralelismo é mais baixo, estando maioritariamente relacionado com o processamento de pedidos e mecanismos de gestão de I/O. A versão PostgreSQL 9.3 introduziu um conceito de *Dynamic Background Workers*, processos que correm em backgound – daemons – e que não se ligam a um cliente e podem ser invocados dinamicamente para processamento. Uma vez que o PostgreSQL usa modelo baseado em processos (e não em *threads*), os dados não são partilhados. Para partilhar dados entre os *backends*, o servidor mapeia uma secção de tamanho fixo de memória quando o servidor arranca, como memória partilhada. Versões

posteriores do PostgreSQL irão suportar *Dynamic Shared Memory*, que irá consistir num mapeamento dinâmico, o que irá dar ao sistema a habilidade de realizar, por exemplo, operações de ordenação em memória que envolvam mais tuplos do que a versão actual [8].

4.5.3 Materialização

A versão PostgreSQL 9.3 introduziu a materialização de *views*. As *materialized views* usam o sistema de regras, tal como as outras *views*, contudo os resultados persistem num *form* semelhante a uma tabela. Para criar uma vista materializada usa-se o comando

```
CREATE MATERIALIZED VIEW <view_name> AS <query>;
```

A principal diferença é que as *views* normais (`CREATE VIEW ...`) têm de ser reescritas sempre que uma *query* recorre a uma, o que por um lado garantia que em todas as ocasiões, a vista estava no seu estado mais actualizado. Com as vistas materializadas, isto não acontece porque os resultados da vista ficam guardados. Subsequentemente, a vista materializada não pode ser actualizada, portanto para que a vista seja actualizada com o estado mais recente, é necessário usar o comando `REFRESH MATERIALIZED VIEW`. A *performance* ganha usando este método poderá ser substancial nalguns casos.

5 Gestão de transacções e controlo de concorrência

Ao longo desta secção serão apresentados vários exemplos para ilustrar os diferentes comportamentos do PostgrSQL na gestão de transações. Todos os exemplos vão basear-se numa base de dados bastante simples, apresentada na listagem 2.

```
1  create table t1 (a int);
2  create table t2 (b int);
3
4  insert into t1 values(5);
5  insert into t1 values(10);
6  insert into t1 values(15);
7
8  insert into t2 values(2);
9  insert into t2 values(4);
10 insert into t2 values(6);
```

Listing 2: Base de dados de referência

5.1 Transações

No PostgreSQL cada comando SQL é interpretado como uma transação. No entanto, é também possível tratar um conjunto de comandos SQL como uma transação com as *keywords* `BEGIN` e `END`, antes e depois do conjunto de comandos respectivamente. Para anular os efeitos de uma transação utiliza-se o comando `ROLLBACK`, e para tentar terminá-la o comando `COMMIT`. De notar que o comando `END` tenta sempre fazer `COMMIT` da transação, mas o resultado pode tanto ser *committed* como *rolled back*. Apesar de não suportar *nested transactions*, a utilização de *savepoints* permite simular esse comportamento. Um *savepoint* apenas se pode usar no contexto de uma transação e é utilizado para permitir fazer `ROLLBACK` parcial de uma transação. Ao fazer `ROLLBACK` para um *savepoint*, todas as alterações feitas antes do *savepoint* são mantidas, enquanto que as alterações posteriores são anuladas. Na listagem 3 exemplificamos a utilização de *savepoints*. Inicialmente a tabela *t1* apenas contém os três tuplos iniciais. No exemplo é inserido um tuplo antes e um tuplo depois do *savepoint*, seguido do `ROLLBACK` para o mesmo, e no fim é feito o `COMMIT` da transação. A consulta final confirma o comportamento esperado, em que apenas um dos tuplos é inserido na tabela.

Os mecanismo acima descritos são semelhantes aos encontrados no Oracle SQL embora existam algumas ligeiras diferenças ao nível da sintaxe.

O controlo de concorrência aos dados é analisado em algum detalhe nas próximas secções.

```
BEGIN;

INSERT into t1 values (20);
SAVEPOINT save1;

INSERT into t1 values (22);
# afinal só quero inserir múltiplos de 5 na tabela t1
ROLLBACK to save1;

END;

# fim da transação

SELECT * from t1;
  a
----
  5
 10
 15
 20
(4 rows)
```

Listing 3: Utilização de *savepoints*

5.2 Controlo de concorrência

O PostgreSQL utiliza um modelo de multiversão de forma a garantir a consistência dos dados nas suas base de dados. O Multi-version Concurrency Control (MVCC) é um mecanismo de controlo de concorrência caracterizado por uma contenção de *locks* bastante modesta, permitindo deste modo um bom desempenho mesmo em ambientes com muitos acessos em simultâneo. Ao manter várias versões de cada tuplo, o MVCC evita que os *locks* de leitura entrem em conflito com os *locks* de escrita e vice-versa. A par deste mecanismo, o PostgreSQL dá ainda a possibilidade de utilização de *locks* explícitos, ficando a cargo do progamador gerir as situações de conflitos. Embora disponível, a gestão explicíta de *locks* é desaconselhada por i) ser normalmente menos eficiente e ii) ser mais susceptível a erros de progamação.

Ambos os mecanismos serão abordados nas próximas secções.

5.3 *Multiversion Concurrency Control*

Como descrito na secção anterior, no modelo MVCC são permitidas várias versões do mesmo tuplo na base de dados. Embora seja benéfico em termos de performance, levanta um novo problema: qual versão do tuplo deve uma transação ler? O conceito de *tuple visibility* introduz um conjunto de regras que definem em cada momento que versão do tuplo deve uma transação ler. Este conceito é utilizado para definir o *snapshot* de uma transação. O *snapshot* de uma transação é a visão que esta tem da base de

dados num dado instante. Quando uma transação começa é-lhe atríbuida um *id* que a identifica de forma inequívoca. Os *ids* de transações têm a seguinte propriedade:

$$T_{id}(A) > T_{id}(B) \Leftrightarrow T_0(A) > T_0(B) \tag{1}$$

onde T_0 representa o instante inicial da transação.

Quando o *snapshot* de uma transação é feito, esta guarda uma lista de *ids* das transações activas naquele momento. Para decidir que versão do tuplo deve usar, além dos *ids* acima descritos, uma transação necessita de saber o par $(xmin, xmax)$ associado ao tuplo. Este par encontra-se no cabeçalho de cada tuplo, e é independente entre versões, isto é, cada versão do tuplo tem o seu próprio par. Este par têm o seguinte significado:

- **xmin** - id da transação que criou esta versão do tuplo. Este campo nunca pode ser alterado.
- **xmax** - id da transação que expirou (eliminou/actualizou) esta versão do tuplo.

A seguir apresentamos os comandos SQL autorizados a alterar estes valores, e de que forma o fazem:

- `INSERT` - cria uma nova versão do tuplo com o par $(T_{id}, null)$.
- `DELETE` - altera o par $(xmin, null)$ para $(xmin, T_{id})$. Esta acção designa-se de "expirar" o tuplo.
- `UPDATE` - é traduzido num `DELETE` seguido de um `INSERT`.

onde T_{id} representa o *id* da transação que executa o comando. O cabeçalho contém ainda um apontador para uma versão mais recente do tuplo, caso esta exista.

A regra para determinar se um tuplo é visível para uma transação pode ser resumida no algoritmo 0.

As listagem 4 e 5 permitem observar as alterações aos pares $(xmin, xmax)$ de acordo com o descrito anteriormente. Seja a transação da esquerda $T1$ e a da direita $T2$, com *ids* 1137 e 1138 respectivamente. O primeiro `SELECT` em $T1$ mostra o estado inicial da tabela $t1$. O `INSERT` na linha 13 cria um novo tuplo $a = 20$ e o `UPDATE` na linha 14 altera o tuplo $a = 10$ para $a = 50$. O `SELECT` na linha 15 mostra o novo estado da tabela t1. O tuplo inserido tem associado o par $(1137, 0)$ como seria de esperar. Mais interessante de observar é o comportamento do `UPDATE` (`DELETE` seguido de `INSERT`). O tuplo $a = 50$ tem agora $xmin = 1137$, o que prova necessariamente que este tuplo é outra versão (lembrar que o $xmin$ nunca pode ser alterado). Do lado da $T2$ podemos também confirmar este comportamento. O `SELECT` executado por esta transação mostra-nos exactamente o estado inicial da tabela

Algorithm 1 Check tuple-validity

```
1: activeTx ← txIdsList
2: currentTx ← currentTxId

3: if xmin < currentTx then
4:     if T_id(xmin) = COMMITTED then
5:         if (xmax = null) ∨ (xmax ∈ activeTx) then
6:             return TRUE
7:         else
8:             return FALSE
9:         end if
10:    else
11:        return FALSE
12:    end if
13: else
14:    return FALSE
15: end if
```

$t1$. Olhando para o tuplo $a = 10$ (aquele que foi alvo do `UPDATE`) podemos verificar duas coisas: i) $xmin = 1335$, ou seja trata-se da versão antiga do tuplo, e ii) $xmax = 1137$, o que significa que esse tuplo foi expirado por $T1$. Apesar disso, o tuplo continua visível para $T2$ uma vez que a transação $T1$ ainda não fez *commit* e encontrava-se activa quando *T2* começou. Pela mesma razão, o tuplo $a = 20$ não aparece (ver condição na linha 4 do algoritmo).

```
BEGIN;

select *, xmin, xmax from t1;

# OUTPUT
 a  | xmin | xmax
----+------+------
  5 | 1134 |    0
 10 | 1135 |    0
 15 | 1136 |    0
(3 rows)

insert into t1 values (20);
update t1 set a=50 where a=10;
select *, xmin, xmax from t1;

# OUTPUT
 a  | xmin | xmax
----+------+------
  5 | 1134 |    0
 15 | 1136 |    0
 20 | 1137 |    0
 50 | 1137 |    0
(4 rows)

END;
```

Listing 4: $T1$ - *id* 1337

```
BEGIN;









select *, xmin, xmax from t1;

# OUTPUT
 a  | xmin | xmax
----+------+------
  5 | 1134 |    0
 10 | 1135 | 1137
 15 | 1136 |    0
(3 rows)


END;
```

Listing 5: $T2$ - *id* 1338

O MVCC obtém todos os *locks* necessários automaticamente, no entanto é também possível obter *locks* explicitamente. Na próxima secção vamos apresentar os tipos de *locks* existentes e o seu comportamento.

5.4 Locks

No PostgreSQL, os *locks* podem ser de duas granularidades diferentes: *table-level locks* e *row-level locks*. Em ambos os casos os *locks* são libertados apenas no final da transação, de acordo com o protocolo *two-phase locking*. Caso seja feito um *rollback* para um *savepoint*, todos os *locks* pedidos após o mesmo são libertados. Este comportamento é um requisito para garantir que todas as acções da transação são revertidas em caso de *rollback* da transação (neste caso, um *rollback* parcial). Os *locks* são implementados recorrendo a semáforos e a sua gestão é feita numa zona de memória partilhada, que mantém uma tabela de dispersão com a informação dos *locks* activos.

É possível monitorizar os *locks* activos através de uma consulta SQL. A listagem 6 mostra uma transacção a executar um `SELECT` e um `INSERT`. A listagem 7 mostra os *locks* associados a ambos os comandos: *AccessShareLock* e *RowExclusiveLock*, respectivamente (uma explicação sobre o significado de cada tipo de *lock* encontra-se mais à frente). A consulta foi feita às tabelas *pg_locks* e *pg_stat_all_tables*, duas tabelas geridas pelo PostgreSQL.

```
BEGIN;

select * from t1;
insert into t1 values(5);

END;
```

Listing 6: Obtenção dos *locks*

```
select t.relname,l.locktype,page,virtualtransaction,pid,
    mode,granted from pg_locks l, pg_stat_all_tables t
    where l.relation=t.relid order by relation asc;

# OUTPUT

   relname     | locktype |        mode
---------------+----------+---------------------
 pg_class      | relation | AccessShareLock
 pg_index      | relation | AccessShareLock
 pg_namespace  | relation | AccessShareLock
 t1            | relation | AccessShareLock
 t1            | relation | RowExclusiveLock
(5 rows)
```

Listing 7: *Locks* activos

5.4.1 *Table level locks*

Os *table locks* disponíveis no PostgreSQL são:

- ***ACCESS SHARE*** - *lock* usado pelo comando `SELECT`. Tipicamente usado quando apenas se pretende ler uma tabela.
- ***ROW SHARE*** - *lock* usado pelo comando `SELECT FOR [UPDATE | SHARE]`
- ***ROW EXCLUSIVE*** - os comandos que modificam uma tabela (`INSERT`, `DELETE` e `UPDATE`) obtém este *lock*
- ***SHARE UPDATE EXCLUSIVE*** - *lock* que previne a alteração concorrente ao esquema da base de dados (`ALTER TABLE`) e execuções *VACUUM*.
- ***SHARE*** - previne a alteração concorrente de dados numa tabela
- ***SHARE ROW EXCLUSIVE*** - semelhante ao de cima, mas com a garantia de ser exclusivo.
- ***EXCLUSIVE*** - previne todo o acesso que não seja de leitura à tabela em questão.

- ***ACCESS EXCLUSIVE*** - garante que o "dono"do *lock* seja o único com acesso à tabela. Este *lock* é usado para fazer alterações ao esquema da tabela (`[ALTER | DROP] TABLE`) e é também usado pelas execuções *Vacuum Full*, *Reindex* e *Cluster*.

Os conflitos entre os vários *locks* são apresentados pela tabela 2 (retirada directamente da documentação do PostgreSQL). Quando dois *locks* são conflituosos entre si, significa que não podem haver duas transações cada uma com um desses *locks* em simultâneo.

Lock	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
(1) ACCESS SHARE								X
(2) ROW SHARE							X	X
(3) ROW EXCLUSIVE					X	X	X	X
(4) SHARE UPDATE EXCLUSIVE				X	X	X	X	X
(5) SHARE			X	X		X	X	X
(6) SHARE ROW EXCLUSIVE			X	X	X	X	X	X
(7) EXCLUSIVE		X	X	X	X	X	X	X
(8) ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

Tabela 2: Conflitos entre *locks*

5.4.2 *Tuple level locks*

Os *locks* ao nível dos tuplos podem partilhados ou exclusivos. Quando um tuplo é modificado ou apagado por uma transação, esta obtém automaticamente um *lock* exclusivo sobre o tuplo. Estes *locks* seguem a mesma filosofia que os *locks* ao nível da tabela, e são apenas libertados no fim da transação. Isto significa que uma transação a partir do momento em que obtém este *lock* tem a garantia de poder alterar o tuplo quantas vezes quiser sem haver o perigo de conflitos. O comando `SELECT FOR UPDATE` permite obter este *lock* ainda antes de modificar o tuplo.

Relativamente aos *locks* partilhados, estes podem ser obtidos através do comando `SELECT FOR SHARE`. Uma vez que são partilhados, múltiplas transações podem obter este tipo de *lock* sobre o mesmo tuplo. No entanto, o tuplo não poderá ser alterado enquanto houver uma transação com um *lock* partilhado sobre o tuplo.

5.4.3 *Advisory locks*

Este tipo de *locks* são dependentes da semântica da aplicação, pelo que a sua utilização fica completamente a cargo do progamador. Podem ser obtidos a nível da transação mas também ao nível de sessão. Os *locks* obtidos ao nível de sessão têm um comportamento bastante distinto do tradicional: i) têm sempre que ser libertados explicitamente e ii) pode ser obtido várias vezes; neste caso cada *lock* terá de ser libertado independentemente.

5.4.4 *Deadlocks*

Embora *deadlocks* sejam mais frequente quando se utiliza *locking* explicíto, estes podem acontecer em qualquer ambiente. Para detectar e resolver situações de *deadlock*, o PostgreSQL mantém um grafo que contém informação sobre os processos que estão bloqueados à espera que um *lock* lhe seja atribuído. Um ciclo neste grafo significa uma situação de *deadlock*. Por isso, a abordagem do PostgreSQL é executar, periodicamente, um algoritmo para detectar ciclos neste grafo. No caso de detectar um ciclo, o sistema aborta uma das transações responsáveis pelo ciclo de modo a resolver a situação de *deadlock*.

5.5 Níveis de isolamento

A sintaxe do PostgreSQL aceita os quatro níveis de isolamento definidos no *standard* do SQL. No entanto, internamente o nível *Read Uncommitted* é transformado no nível *Read Committed*. Para definir o nível de isolamento para uma transação utiliza-se o comando `BEGIN TRANSACTION ISOLATION LEVEL` seguido do nível de isolamento pretendido. Nas próximas secções iremos descrever cada um dos níveis de isolamento, apontando as diferenças fundamentais entre eles.

5.5.1 *Read Committed*

O *Read Committed* é o nível de isolamento que o PostgreSQL usa por *default*. A regra deste nível de isolamento é simples: cada comando SQL trabalha sobre os dados *committed* no instante da execução do mesmo. Ou seja, cada comando SQL trabalha sobre um *snapshot* diferente da base de dados. Este nível de isolamento também está disponível no *Oracle SQL* e o seu comportamento é semelhante.

5.5.2 *Repeatable Read*

A implementação deste nível de isolamento no PostgreSQL é mais restritivo que o definido no *standard SQL*. Na verdade, este nível de isolamento implementa *Snapshot Isolation*. Ao invés de efectuar um *snapshot* em cada comando *SQL* como o *Read Committed* faz, neste nível o *snapshot* é feito uma única vez no início da transação, e toda ela executa sobre esse *snapshot*. Neste nível de isolamento não há a garantia que as transações sejam serializáveis uma vez que não detecta conflitos *read-write*. Por isso, neste nível de isolamento *rollbacks* podem acontecer. Nesse caso o PostgreSQL informa o utilizador com uma mensagem de erro: *"could not serialize access due to concurrent update"*. Este nível de isolamento não está disponível no *Oracle SQL*.

5.5.3 *Serializable*

Este nível de isolamento é o mais restrito possível e garante a seguinte propriedade: se uma transação executa sem anomalias num ambiente sequêncial então também o faz em ambiente concorrente. Tradicionalmente, os Sistemas de Gestão de Base de Dados para implementar este nível de isolamento recorriam ao protocolo *strict two-phase locking* conjuntamente com *predicate locks*. No entanto, esta implementação tem bastantes problemas de performance. O PostgreSQL implementa este tipo de isolamento com *Snapshot Isolation* em conjunto com algumas verificações em *runtime* para detectar possíveis anomalias, nomeadamente conflitos *read-write*. Caso alguma anomalia seja detectada, uma das transações é abortada pelo sistema. No *Oracle SQL* este nível de isolamento corresponde ao *Snapshot Isolation* descrito na secção anterior.

5.6 Consistência

O PostgreSQL é bastante flexível na verificação de restrições e permite que o progamador configure quando estas verificações são feitas. Com excepção das restrições do tipo `CHECK` e `NOT NULL`, a verificação das outras restrições podem ser "adiadas"no final da transação. Para manipular estas configurações, o PostgreSQL oferece os seguintes comandos:

- ***SET CONSTRAINT IMMEDIATE*** - nesta configuração a verificação da restrição é feita no final de cada comando *SQL*.
- ***SET CONSTRAINT DEFERRED*** - nesta configuração a verificação é adiada para o final da transação por *default*. No entanto, é possível alterar a altura em que as validações são feitas para *IMMEDIATE* em tempo de execução, bastando para isso utilizar o comando `SET CONSTRAINT IMMEDIATE`. Neste caso todas as restrições em "espera"são validadas e caso alguma delas seja violada o comando dá erro.

5.7 Durabilidade

Para garantir a durabilidade das transações, o PostgreSQL utiliza o mecanismo de Write-Ahead Log (WAL). Este mecanismo garante que quando alguma alteração é feita à base de dados, toda a informação necessária para concluir (ou reverter) a operação (neste caso transação) já está escrita em disco, num ficheiro de *log*. Em caso de *crash* do sistema, uma vez que a informação do *log* é mantida, é possível utilizá-la para aplicar as alterações que ainda não tinham sido feitas na base de dados. As garantias que este ficheiro de *log* oferece evitam que as transações necessitem de fazer *flush* das alterações para disco aquando o seu *commit*. Isto resulta num aumento substancial de performance uma vez que durante uma transação todas as

escritas em disco são feitas no mesmo ficheiro (*logfile*), minimizando por isso o número de *seeks*.

6 Suporte para bases de dados distribuídas

É comum haver sistemas em diversos servidores de bases de dados funcionam em conjunto para garantir questões como a disponibilidade (e.g. se um servidor principal falhar, ser possível um secundário assumir as funções do primeiro) ou a distribuição de carga. No entanto, tal cenário representa um problema de sincronização, uma vez que é necessário garantir que, apesar de existirem vários servidores, as bases de dados deverão estar consistentes em todos os instantes – individualmente ou como um todo.

Existem bastantes soluções que lidam com o problema de maneira diferente, no entanto cada solução insere-se especificamente num cenário, resolvendo muitas vezes apenas um subconjunto do problema. Uma das abordagens é limitar a modificação de informação a apenas um servidor. O PostgreSQL é um exemplo de um sistema que implementa essa abordagem.

É importante notar que existem bastantes sistemas (comerciais e *open source*) que implementam suporte para replicação, escalabilidade, etc. sobre o PostgreSQL (Slony, pgPool, Bucardo, etc), mas que no entanto não recorrem aos mecanismos internos do PostgreSQL para o fazer. Esta secção focar-se-á apenas no suporte para bases de dados distribuídas que o PostgreSQL oferece *per se*.

6.1 *Log-Shipping*

Log-Shipping é uma solução de replicação, onde réplicas de um servidor principal são mantidas actualizadas através da transferência dos registos WAL. O PostgreSQL adopta uma abordagem baseada em ficheiros, i.e. envia todos os registos WAL num único ficheiro, tipicamente de $16MB$. O facto de este ficheiro ter um tamanho relativamente reduzido, torna possível que a sua transferência a qualquer distância – para um sitema adjacente ou geograficamente distante – seja feita de forma rápida. Naturalmente que a *bandwidth* necessária para implementar esta técnica depende da taxa de transacções que o servidor primário precisa de atender.

Uma vez que é assíncrona, i.e. os registos WAL são enviados assim que uma transacção terminar com sucesso (`COMMIT`), existe a possibilidade de perda de informação, caso o servidor primário sofra uma falha e as transacções não tiverem sido enviadas. Para diminuir a janela de erro, é possível definir a frequência com que o servidor irá criar segmentos WAL através do parametro `archive_timeout`: um valor mais baixo fará com que as transferências sejam mais frequentes, no entanto poderá incorrer num aumento de *bandwidth* necessária. *Streaming replication* permite ter uma janela de perda de dados muito mais pequena – será analisado mais à frente.

Habitualmente, é sensato que o servidor primário e as réplicas sejam o mais semelhantes possível, pelo menos em termos de sistemas de bases de dados. Em particular, as *tablespaces* passam do servidor primário para

as réplicas sem ser modificados, portanto tanto o servidor primário como as réplicas precisam de ter os mesmos caminhos montados. O hardware não precisa de ser igual, no entanto o sistema seria mais fácil de manter. Embora haja alguma flexibilidade relativamente ao hardware, o mesmo não se verifica para a arquitectura – não é possível transferir de sistemas a 32 bits para sistemas a 64 bits. Neste aspecto, o PostgreSQL necessita que os seus servidores sejam homogéneos.

6.2 Servidor principal e réplicas *standby*

Quando está no modo *standby*, uma réplica aplica continuamente os WALs recebidos do servidor principal. Isto pode ser feito lendo de um arquivo ou directamente do servidor principal, através de uma conexão TCP. O administrador pode ainda colocar manualmente cópias dos WALs na directoria `pg_xlog`.

6.2.1 Configuração de uma réplica *Standby*

É necessário criar um ficheiro `recovery.conf` na directoria de dados do cluster da réplica, e accionar o modo `stanby_mode`. É também necessário configurar outros parâmetros que são essenciais para o funcionamento da réplica, como a ligação ao servidor principal e credenciais de autenticação.

Este é um exemplo simples de um ficheiro `recovery.conf`:

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
restore_command = 'cp /path/to/archive/%f %p'
archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'
```

6.3 *Streaming Replication*

Streaming replication garante que as réplicas estão o mais actualizado possível, uma vez que estas se ligam directamente ao servidor principal, que por sua vez lhes transmitem registos WAL assim que estes são gerados – sem ter de esperar pelo preenchimento de um ficheiro. A supressão da espera faz com que o impacto do problema referido anteriormente seja mitigado – a janela de perda de dados é reduzida.

Para testar a eficiência da replicação por *stream*, pode-se analisar qual a quantidade de registos WAL gerados no servidor principal, e quais foram aplicados pela réplica, num dado instante de tempo. Para tal, pode recorrer-se à rotina `pg_current_xlog_location` no servidor principal e a `pg_last_xlog_receive_location` na réplica. Uma diferença significativa entre o último registo gerado e o último registo enviado, pode aferir-se que existe algum *overhead* no servidor principal; da mesma maneira, uma diferença significativa entre o último registo enviado pelo principal e o último

registo aplicado pela réplica, pode aferir-se que se verifica algum *overhead* na réplica.

6.4 *Cascading Replication*

Cascading replication permite que apenas algumas das réplicas necessitem de estabelecer uma conexão com o servidor principal para receberem os registos WAL; estas, por sua vez, retransmitem para as restantes réplicas. Esta abordagem é vantajosa porque não só diminui o número de conexões directas que o servidor principal precisa de manter, como minimiza o *overhead* de *bandwidth* nas comunicações. Neste modelo, apesar de não existirem limitações quanto ao número ou disposição de réplicas receptoras (réplicas que apenas recebem actualizações), cada réplica apenas pode receber actualizações de um emissor.

6.5 *Synchronous Replication*

A replicação síncrona reforça o nível de durabilidade oferecido pelo *commit* de uma transacção, uma vez que oferece a garantia de que todas as alterações feitas por uma transacção foram transmitidas para uma *réplica síncrona – 2-safe replication*.

Com esta abordagem, cada *commit* de uma transacção de escrita terá de esperar até que seja recebida uma confirmação de que foi registado o *log* da transacção em disco, tanto no servidor primário como na réplica. A única maneira de haver perdas de informação, é se ambos sofrem um *crash* simultaneamente. Naturalmente, esta abordagem tem impacto no tempo de resposta de uma transacção, sendo o tempo mínimo o RTT entre o servidor primário e a réplica.

7 Outras características do sistema estudado

Nesta secção abordamos características que não se enquadram em nenhum tópico abordado até agora.

7.1 Suporte a outras linguagens

Neta secção descrevemos o suporte oferecido pelo PostgreSQL a outras linguagens.

7.1.1 Linguagens procedimentais

O SQL é *Turing complete* (a demonstração está fora do contexto desta cadeira). Isto significa que o suporte para linguagens procedimentais não adiciona expressividade ao PostgreSQL. No entanto, o suporte a outras linguagens facilita (bastante) a implementação de certos procedimentos por parte do progamador.

As linguagens procedimentais suportadas pelo PostgreSQL são as seguintes:

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

Existem ainda outras linguagens que são suportadas, embora não estejam incluídas na distribuição oficial do PostgreSQL:

- PL/Java
- PL/PHP
- PL/Py
- PL/R
- PL/Ruby
- PL/Scheme
- PL/sh

7.1.2 Drivers

Além do suporte interno a várias linguagens, existem *drivers* que permitem a comunicação de aplicações externas com o PostgreSQL. O papel destes *drivers* é traduzir instruções na linguagem da aplicação em instruções que o PostgreSQL compreende e vice-versa.

Np PostgreSQL existem *drivers* para as seguintes linguagens:

- Perl
- Java
- C++
- Python
- PL/Ruby

– PL/Scheme
– PL/sh

7.2 Suporte a XML

O suporte para XML é feito pela biblioteca *libxml*. Por questões de performance e para garantir o funcionamento correcto de algumas funções é aconselhável que este tipo de dados utilize o formato UTF-8. Além disso, um valor do tipo XML tanto pode ser um documento completo como um fragmento XML.

Para manipular dados do tipo XML o PostgreSQL disponibiliza os seguintes comandos:

- **XMLPARSE ([DOCUMENT | CONTENT]) value** - transforma um documento ou fragmento XML num atributo do tipo XML.
- **XMLSERIALIZE ([DOCUMENT | CONTENT]) value AS type** - transforma um atributo do tipo XML no tipo especificado por *type*. Os valores possíveis para *type* são: *character*, *character varying* e *text*.

A comparação de valores de tipo XML não é possível, pelo que consultas baseadas no conteúdo destes atributos não são permitidas pelo PostgreSQL. Pela mesma razão não é possível criar índices sobre atributos deste tipo.

7.3 Autenticação

A autenticação de clientes é controlada pelo ficheiro de configuração *pg_hba.conf*. Este ficheiro contém várias entradas com o seguinte formato:

```
local   DATABASE  USER  METHOD  [OPTIONS]
host    DATABASE  USER  ADDRESS METHOD  [OPTIONS]
```

A primeira entrada refere-se a clientes que estejam na mesma máquina que o servidor, enquanto que a segunda refere-se a clientes remotos. O significado de cada atributo é:

- **DATABASE** - a quais bases de dados a entrada se refere.
- **USER** - o user a que entrada se aplica.
- **METHOD** - modo de autenticação utilizado. Este atributo pode tomar os seguintes valores: trust, reject, md5, password, gss, sspi, krb5, ident, peer, ldap, radius, cert e pam.
- **ADDRESS** - no caso de clientes remotos este atributo especifica o endereço da máquina de onde o cliente.

- **OPTIONS** - opções avançadas de autenticação. Ver a documentação para mais informações.

Referências

[1]

[2] Memory architecture.

[3] Postgresql 9.3.4 documentation. http://goo.gl/oU4wxv.

[4] Postgresql wiki. http://goo.gl/9dzrwM.

[5] Buffer management README - github. http://goo.gl/DWv5TK, 2014.

[6] Fabien Coelho. Turing machine in sql. http://goo.gl/rCzyxi, 2013.

[7] Korry Douglas. PostgreSQL SQL syntax and use.

[8] Robert Haas. Parallelism progress. http://goo.gl/ow8h0A, 2013.

[9] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[10] Brune Momjian. Mvcc unmasked. http://goo.gl/ZQYi1p, 2013.

[11] Dan R. K. Ports and Kevin Grittner. Serializable snapshot isolation in postgresql. *CoRR*, abs/1208.4179, 2012.

[12] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 6 edition, 2010.

[13] G. Smith. *PostgreSQL 9.0: High Performance*. Open source : community experience distilled. Packt Publishing.

A Parameterizações do Planeamento de Perguntas

O PostgreSQL possui bastantes parâmetros que podem ser afectados para que seja possível influenciar a geração de planos de execução. A melhor maneira de melhorar a qualidade dos planos seria deixar que o optimizador tomasse a sua decisão com base nas estatísticas do sistema. Para melhorar a qualidade das estatísticas poder-se-ia correr o `ANALYZE` manualmente, ou mesmo adicionar mais informação ao conjunto já existente. Tal poderia ser feito com `ALTER TABLE SET STATISTICS`. Aqui listamos alguns dos comandos mais importantes para a configuração do sistema e para o planeamento. Para alterar o valor de um parâmentro usar o comando `SET`:

```
SET enable_sort = off;
```

Activação de algoritmos suportados

Estes parâmetros especificam os algoritmos que o *planner* poderá usar quando gerar os planos de execução possível. É importante notar que não é possível suprimir completamente alguns, no entando afectando o parâmetro para *off* diz ao *planner* para evitar aquele modo, desde que hajam outros disponíveis. O valor por defeito para todos é *on*.

```
enable_bitmapscan (boolean)

enable_hashagg (boolean)

enable_hashjoin (boolean)

enable_indexscan (boolean)

enable_indexonlyscan (boolean)

enable_material (boolean)

enable_mergejoin (boolean)

enable_nestloop (boolean)

enable_seqscan (boolean)

enable_sort (boolean)

enable_tidscan (boolean)
```

Constantes de custo

Como já foi dito, os custos são relativos ao custo da obtenção de páginas do disco. Aqui apresentamos alguns parâmetros relevantes onde é possível especificar custos de operação.

`seq_page_cost (floating point)`
Sets the planner's estimate of the cost of a disk page fetch that is part of a series of sequential fetches. The default is 1.0. This value can be overridden for tables and indexes in a particular tablespace by setting the tablespace parameter of the same name (see ALTER TABLESPACE).

`random_page_cost (floating point)`
Determina o custo de ler uma página do disco de maneira não sequencial. O *default* é `4.0`.

`cpu_tuple_cost (floating point)`
Determina a estimativa do *planner* do custo de processar um tuplo durante uma query. O *default* é `0.01`.

`cpu_index_tuple_cost (floating point)`
Determina a estimativa do *planner* do custo de processar um tuplo durante um *index scan*. O *default* é `0.05`.

`cpu_operator_cost (floating point)` Determina a estimativa do *planner* do custo de processar cada operador ou função durante uma *query*. O *default* é `0.0025`.