FCT-UNL

Relatório SBD

PostgreSQL

-Grupo 16-
41789, Pedro Lourenço
41930, Vasco Lopes
42076, Ian Rodrigues

1 Introdução

Este trabalho é realizado no âmbito da cadeira de Sistema de Bases de Dados. Objectivo é consolidar a matéria dada ao longo do semestre e estudar um SGBD, de modo a compreendê-lo e enquadrá-lo com o que foi estudado. O SGBD escolhido foi o PostgreSQL 9.3 e este será comparado com o Oracle 11g, que foi o SGBD utilizado ao longo das aulas.
O PostgreSQL nasceu depois de, em 1985, Michael Stonebraker ter regressado à Universidade da Califórnia para continuar o projecto Ingres. O objectivo inicial era adicionar algumas novas funcionalidades ao Ingres. Estas funcionalidades incluíam a possibilidade de definir tipos e descrever completamente relações. O PostgreSQL usou muitas das ideias do projecto Ingres, mas não usou o seu código.
Em 1986 foram publicados uma série de artigos pela equipa de desenvolvimento, descrevendo o essencial do sistema e, em 1988 surgiu a versão em protótipo. Nos anos seguintes foram sendo lançadas novas versões do SGBD em que foram adicionadas funcionalidades como o suporte para múltiplos gestores de armazenamento, também foi melhorado o processamento de queries e as regras do sistema foram redefinidas. Depois da versão 4, o projecto terminou.

2 Armazenamento e File Structure

2.1 Organização do sistema de ficheiros

O PostgreSQL utiliza o sistema de ficheiros do sistema operativo, ao invés de ter o seu próprio sistema de ficheiros. Isto por um lado diminui a complexidade e esforço da implementação do SGBD, mas o desempenho das operações vai estar associado ao sistema de ficheiros do sistema operativo.
As configurações e os ficheiros de dados, da base de dados, estão todos armazenados na pasta denominada PGDATA. Esta directoria tem várias subdirectorias e ficheiros de controlo. O conteúdo da directoria PGDATA pode ser verificado na tabela seguinte:

Items	**Descrição**
PG_VERSION	Ficheiro que contém o número da versão do PostgreSQL.
base	Subdirectoria que contém as subdirectorias de cada base de dados.
global	Subdirectoria que contém as tabelas comuns ao cluster.
pg_clog	Subdirectoria que contém informação sobre o estados dos commits.
pg_multixact	Subdirectoria que contém informação sobre o estado de multi-transações.
pg_notify	Subdirectoria que contém informação sobre o estado de LISTEN/NOTIFY.
pg_serial	Subdirectoria que contém informação sobre transacções serializáveis committed.
pg_snapshots	Subdirectoria que contém os snapshots exportados.
pg_stat_tmp	Subdirectoria que contém ficheiros temporários para o subsistema de estatística.
pg_subtrans	Subdirectoria que contém informação sobre o estado de subtransacções.
pg_tblspc	Subdirectoria que contém ligações simbólicas à tablespace.
pg_twophase	Subdirectoria que contém ficheiros de estado para transacções preparadas.
pg_xlog	Subdirectoria que contém ficheiros WAL (Write Ahead Log).
postmaster.opts	Um ficheiro que contém as opções de comando com que o servidor foi iniciado a última vez.
postmaster.pid	Um ficheiro que guarda o ID do processo, path, timestamp de inicialização, port, socket path, IP adress, ID da memória partilhada.

Quando uma tabela ou índex excede o tamanho de 1GB, é dividida em segmentos de 1GB cada. O tamanho máximo é 1GB, por defeito, mas pode ser alterado. O primeiro segmento fica com o nome filenode respectivo, os seguintes são nomeados filenode.1, filenode.2 e por ai adiante. Tudo isto serve para evitar problemas em plataformas que têm restrições no tamanho de ficheiros.

2.2 TOAST (The Oversized-Attribute Storage Technique)

Tabelas que tenham colunas com valores potencialmente muito grandes, vão ter tabelas TOAST associadas. O PostgreSQL usa tamanhos de páginas fixos (8KB) e não permite que os tuplos ocupem múltiplas páginas. O que se faz é comprimir e/ou partir os campos maiores em múltiplas entradas físicas. Isto acontece de forma transparente para o utilizador. Existem tipos de atributos que não produzem conteúdos de grande

dimensão, pelo que TOAST não é aplicado nestes casos. É ainda possível configurar a maneira como são guardadas as entradas TOAST, as estratégias são as seguintes:

- PLAIN:
  Não se aplica compressão ou separação dos dados. É a estratégia utilizada com os tipos de dados aos quais não se pode aplicar TOAST.
- EXTENDED:
  Pode aplicar-se compressão e separação de dados. É a estratégia utilizada com os tipos de dados aos quais se pode aplicar TOAST. Primeiro tentar-se-á comprimir, só se fará separação se a entrada ainda for muito grande.
- EXTERNAL:
  Permite separação de dados, mas não compressão. Optimiza a manipulação dos dados, mas ocupa mais espaço.
- MAIN:
  Permite compressão. Em relação à separação de dados, só o permite em último recurso.

2.3 Free Space Map

Cada heap e índex, excepto o hash índex tem um Free Space Map (FSM), que mantém informação sobre o espaço livre.
O Free Space Map está organizado como uma árvore de FSM. Os níveis inferiores da árvore guardam o espaço livre em cada heap ou índex, usando um byte para representar cada página. Os níveis de cima da árvore servem para agregar informação sobre os níveis inferiores.

2.4 Visibility Map

Cada heap tem um Visibility Map (VM) cujo trabalho é manter registo das páginas que contêm apenas tuplos que se sabem ser visíveis para todas as transacções activas.
O VM guarda um bit por cada página heap. Se o bit estiver activado, significa que todos os tuplos na página estão visíveis para todas as transacções.

2.5 Estrutura de páginas

Cada tabela e índex está guardado num array de páginas com tamanho fixo (8KB geralmente, a não ser que se tenha especificado o contrário na altura de compilação do servidor). Numa tabela todas as páginas são logicamente equivalentes, por isso um item dessa tabela pode ser guardado em qualquer página. Nos indexes, a primeira página normalmente é reservada para guardar metadata e, podem haver tipos diferentes de páginas para o índex. O aspecto que a página tem é o seguinte:

Items	Descrição
PageHeaderData	Tamanho 24 bytes. Contém a informação geral da página, incluindo apontadores de espaço livre.
ItemIdData	Um array com o par (offset,length) que aponta para o item. 4 bytes por item.
Free space	Espaço livre. Os novos apontadores vão ser alocados no inicio desta área, os novos items vão ser alocados no final desta área.
Items	Os items.
Special space	Vazio em tabelas normais. Espaço reservado para acessos por indexes.

2.6 Partições

Podemos particionar aquilo que é logicamente uma tabela grande em pedaços físicos mais pequenos. O particionamento começa a ter efeitos a partir do momento em que temos uma tabela muito grande, coisa que pode variar de aplicação para aplicação. Isto tem várias vantagens, de entre os quais:

- A performance do processamento de uma query pode melhorar bastante em alguns casos, como por exemplo quando os tuplos mais acedidos da tabela estão todos na mesma partição.
- Quando as queries acedem a uma percentagem grande de uma única partição. A performance pode melhorar ao tirar-se proveito de um varrimento sequencial da partição.

Em PostegreSQL podem ser implementadas duas formas de particionamento:

- Range Partitioning:
  A tabela é particionada em ranges definidos por uma determinada coluna ou por um conjunto de colunas, mas sem haver sobreposição de ranges associados a diferentes partições. Um exemplo é particionar por datas.
- List Partitioning:
  A tabela é particionada por indicar explicitamente que valores chave aparecem em cada partição. (Mais usado quando os valores variam pouco e são bem conhecidos)

Um utilizador pode definir o particionamento de tabelas da forma como está ilustrado no exemplo seguinte:

Criar a tabela pai que vai ser herdada por todas as tabelas:

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

Agora pode-se criar uma partição para a tabela anterior:

CREATE TABLE measurement_y2006m02 ( ) INHERITS (measurement);

Ao invés de criar só a partição, podemos adicionar-lhe logo as restrições, da seguinte maneira:

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

Pode criar-se também um índex nas colunas chave:

CREATE INDEX measurement_y2006m02_logdate ON measurement_y2006m02 (logdate);

Caso, queríamos destruir uma partição, basta executar a seguinte instrução:

DROP TABLE measurement_y2006m02;

Se não quisermos apagar todos os tuplos da partição, apenas fazer com que a tabela deixe de ser uma partição, mas mantendo o acesso aos tuplos, podemos fazer:

ALTER TABLE measurement_y2006m02 NO INHERIT measurement;

2.7 Clustering

Quando é feito cluster a uma tabela, o que se faz é reordenar fisicamente a mesma, segundo a informação de um certo índex previamente criado. Isto pode aumentar a performance, visto que os dados são guardados em blocos contíguos do disco.
Depois de fazer o cluster, e há alterações na tabela, essas alterações não serão clustered. Ou seja, os novos tuplos, ou tuplos actualizados não serão guardados de acordo com a ordem do índex. Pra voltar a reordenar terá de se, periodicamente, executar de novo o comando de clustering.
Exemplos de comandos a executar, caso se queira fazer cluster numa tabela usando determinado índex:

CLUSTER employees USING employees_ind;

Caso se queira fazer cluster da tabela, usando o índex previamente utilizado, basta fazer:

CLUSTER employees;

Ainda existe a possibilidade de fazer cluster de todas as tabelas que anteriormente tinham sofrido operações de cluster, bastanto executar:

CLUSTER;

O PostgreSQL não implementa multitable clustering, pelo que operações executadas frequentemente que juntam tabelas, não serão efectuadas de forma tão eficiente como seriam se existisse a possibilidade de fazer multitableclustering.

3 Indexação e Hashing

3.1 Indexes

Os indexes são estruturas de dados, que contêm referências associadas a uma chave, que é utilizada para fins de optimização, permitindo uma localização mais rápida de um registo quando efectuada uma consulta. A base de dados consegue ir buscar um tuplo ao disco de forma muito mais rápida do que se não tivesse um índex. Uma desvantagem de ter indexes é que podem inserir um overhead ao sistema, principalmente quando estão definidos múltiplos indexes sobre a mesma tabela, por isso devem ser usados com precaução.
Digamos que fazemos a seguinte tabela:

```
CREATE TABLE test1 (
id integer,
content varchar
);
```

Se existem muitas queries que usam o atributo id, talvez seja inteligente índex sobre a tabela que usa o atributo id:

```
CREATE INDEX test1_id_index ON test1 (id);
```

Ou criar um index de Hash, útil para queries que usem igualdades:

```
CREATE INDEX name ON table USING hash (column);
```

Para remover o índex basta fazer o comando DROP INDEX.
Feito index, o SGBD faz a actualização do index sempre que existe uma modificação na tabela, e vai fazer uso do index sempre que este diminua o custo da query em comparação com uma busca sequencial. De forma a ter os custos actualizados, deve se de periodicamente correr o comando ANALYZE, de modo a actualizar as estatísticas e permitir que o query planner tome a decisão acertada.

3.2 Tipos de Indexes

O PostgreSQL oferece diferentes tipos de indexes, cada um deles mais ajustado a certas ocasiões:

- B+-Tree:
  É o index criado por defeito, pois tem boa performance em muitas das ocasiões. É utilizado quando são usados os seguintes operadores: $<$; $<=$; $=$; $>=$; $>$. Operações como BETWEEN, IN, IS NULL, IS NOT NULL sobre uma coluna com index, também usaram as B+-Tree.

- Hash:
  Apenas consegue processar operações de igualdade. Este index no PostgreSQl não suportada recuperação de falhas e é menos eficiente que usar B+-Tree. Por isso a recomendação é nem utilizar este tipo de index, apesar de este estar disponível.

- GiST:
  É possível refinar este index de modo a obter melhorias na performance, suporta os seguintes operadores: «; »; =; <@; <; >.

- GIN:
  São indexes invertidos que podem ser usados para lidar com valores com mais de uma chave, arrays por exemplo. Suporta os seguintes operadores: <@; @>; =; &&.

3.3 Indexes multicoluna

Digamos que se criou a seguinte tabela:

```
CREATE TABLE test2 (
major int,
minor int,
name varchar
);
```

Um index pode ser definido sobre mais do que um atributo da tabela. Isto pode ser feito da seguinte forma:

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

Apenas as B+-Tree, GiST e GIN suportam indexes em múltiplas colunas.

3.4 Indexes parciais

Um index parcial é construído sobre um subconjunto da tabela, esse subconjunto deve ser especificado através de uma expressão. Existem algumas situações um pouco mais específicas em que o uso deste tipo de index é útil.
A principal razão para se usar este index é evitar a indexação de valores que surgem muitas vezes repetidos na tabela, pois estes acabariam por não fazer uso do index de qualquer maneira, por isso mais vale tirá-los do index. Isto vai diminuir o tamanho do index, logo quando uma query fizer uso deste, a rapidez de execução vai ser maior.
Por exemplo, se tivermos uma tabela que guarda os IP address de acessos a um servidor da empresa, a maior parte dos registos provêm de dentro da empresa, por isso esse range de IP address não necessita de ser indexado:

```
CREATE TABLE access_log (
url varchar,
client_ip inet,
...
);
```

O index seria criado, por exemplo desta forma:

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
WHERE NOT (client_ip > inet '192.168.100.0' AND client_ip < inet '192.168.100.255');
```

Uma query que usaria este index seria:

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

Uma que não usaria este index parcial seria:

```
SELECT *
FROM access_log
WHERE client_ip = inet '192.168.100.23';
```

4 Processamento e Optimização de Perguntas

O processamento de uma pergunta SQL segue as seguintes fase: conexão da aplicação ao servidor PostgreSQL, parse da pergunta SQL enviada pela aplicação, reescrita da pergunta, criação de planos de execução e execução da pergunta SQL. De seguida são descritas com maior pormenor cada uma destas fases.

4.1 Conexão ao servidor

O PostgreSQL implementa um modelo cliente servidor, onde se apresenta um processo por utilizador, isto significa que para cada processo de cliente existente, existe um processo servidor correspondente, ao qual este se liga. Visto que não é sabido qual o número de conexões que irão ser feitas, existe um processo master, que cria um novo processo servidor sempre é pedida uma nova conexão a uma dada porta TCP/IP. Os processos servidor comunicam entre si usando semáforos e memória partilhada, para garantir a integridade dos dados durante o acesso concorrente aos mesmos.
Assim que a conexão é estabelecida o cliente envia uma query ao servidor, sendo esta transmitida em plaintext, sendo que não é feito nenhum parse do lado do cliente, apenas do lado do servidor que vai criar um plano de execução, executa-o e de seguida devolve os tuplos obtidos através da conexão criada anteriormente.

4.2 Parse da pergunta SQL

Esta fase contêm duas partes, a parse da pergunta SQL e o processo de transformação da mesma numa query tree.
Na primeira parte, parse da pergunta SQL, o parser recebe a pergunta em plain text e verifica se a mesma apresenta uma sintaxe correta. Se isto não se verificar, é devolvido um erro ao cliente, caso a sintaxe esteja correta irá ser construída uma parse tree, uma estrutura de dados que representa formalmente e de forma não ambígua a pergunta SQL que foi feita. O parser é construído usando as ferramentas Unix bison e flex, sendo definida a gramática a ser usada no ficheiro gram.y.
Na segunda parte, transformação numa query tree da pergunta SQL, o processo de transformação recebe a parse tree, criada no passo anterior, como input e faz a interpretação necessária para se saber quais as tabelas, funções e operadores que estão a ser utilizados na pergunta SQL, sendo a estrutura de dados criada, para representar esta informação, chamada de query tree. A query tree criada é estruturalmente semelhante à parse tree obtida anteriormente, mas inclui informações sobre o tipo de dados das colunas e o resultado de expressões.
Esta fase está dividida em duas partes devido ao facto que consultas ao catálogo de sistema apenas podem ser feitos durante uma transação. Visto que não se pretende iniciar uma transação assim que uma pergunta é recebida, esta fase é separada em duas partes para se poder fazer a análise da pergunta e verificar que esta está construída corretamente, e só depois correr uma transação.

4.3 Reescrita da pergunta SQL

Nesta fase a query tree vai ser modificada através de um conjunto de regras definidas pelo utilizador, sendo o resultado também uma query tree. Um dos usos desta fase é na execução de perguntas em views, onde a pergunta é modificada de maneira a utilizar as views no acesso às tabelas, visto que as views são apenas tabelas com regras aplicadas, sendo que quando uma view é definido o conjunto de regras a ser aplicado a uma pergunta feita sobre essa view.

4.4 Criação do plano de execução

O objectivo desta fase é criar o plano de execução optimo para a query tree dada. Isto involve criar e examinar todos os planos de execução possíveis, caso seja computacionalmente fazível, e escolher o melhor de entre os planos criados. Caso tal não seja possível,por exemplo quando os planos ocupam demasiado espaço de memória ou demoram demasiado tempo, a examinar todas as possibilidades, por exemplo quando a pergunta inclui um grande número de joins, é utilizado o Generic Query Optimizer(GEQO). O número de joins, por omissão, a partir do qual se usa o GEQO é 12, sendo possível ao utilizador alterar este valor.
Nesta fase são utilizadas estruturas de dados chamadas paths, que são representações de planos de execução, contendo apenas a informação necessária para ser possível fazer uma decisão de qual usar. Depois de ser escolhido o path com menor custo, é criado uma plan tree que representa o plano de execução que irá ser corrido.
No início são gerados planos para percorrer as várias tabelas presentes na pergunta, sendo que são gerados planos para os índices de cada relação. Visto que existe sempre a possibilidade de se fazer um scan sequencial na relação, também são criados planos que usem este mecanismo. Caso a relação tenha um índice e este for utilizado numa operação, é criado um plano contendo esse índice. Caso haja alguma restrição que utilize o índice, este seja utilizado na claúsula ORDER BY, ou exista alguma ordenação, pelo índice, que seja útil para o merge join, também é criado um plano que utilize esse índice.
Se a pergunta necessita de join de duas ou mais relaçõe, os planos para a junção de tabelas são considerados depois de todos os planos que involvem percorrer uma única tabela tenham sido descobertos. Para os planos que usam join, existem três estratégias, de join, possíveis: nested loop join, merge join e hash join.
Caso o número de relações usadas seja menor que o valor de threshold, a partir do qual se utiliza o algoritmo GEQO, são consideradas, preferencialmente, os joins entre relações que apresentam uma cláusula de junção no WHERE, da pergunta, sendo que joins que não apresentem tal cláusula apenas são considerados quando não houver outra escolha, ou seja, não apresentem nenhuma cláusula join possível com outra relação.
Caso seja ultrapassado o valor de threshold, é utilizado o Genetic Query Optimization(GEQO), que usa um algoritmo genético, em conjunto com uma heurística, para determinar a combinação de joins entre as relações, que apresenta um menor custo.
O resultado final desta fase consiste numa plan tree com scans de índice ou sequencias nas relações, mais os algoritmos de join escolhidos, e também funções de ordenação e

agregação.
Comparativamente ao SGBD Oracle, o PostgreSQL, não tem implementado o uso de hints nesta fase, sendo que a equipa de desenvolvimento defende que a implementação de hints tornaria a manutenção do código mais difícil, não escalam bem com o tamanho da relação, fazem com que os utilizadores não reportem os problemas com o optimizador, utilizando invés disso, uma hint, e que o uso de hints não melhora a performance da pergunta, visto que na maioria das vezes o optimizador está correcto.

4.5 Execução

O executor é o responsável por executar a pergunta SQL, feita pelo cliente, usando para isso a plan tree criada na fase anterior.
O executor percorre recursivamente o plano recebido, para poder extrair os nós necessários para responder à pergunta. Para isso usa um mecanismo demand-driven pipeline, ou seja, quando um nó é chamado este deve devolver mais um tuplo ou avisar que tal já não é possível pois este não tem mais tuplos para devolver.
Perguntas mais complexas podem envolver vários nós da plan tree mas a sua execução geral é a mesma, sempre que um nó é chamado, este calcula e devolve o seu próximo tuplo, sendo que cada nó é responsável por aplicar qualquer seleção ou projecção que lhe foram atribuídas.
O executor é usado para avaliar os tipos básicos de perguntas em SQL:

- SELECT - Apenas precisa de retornar cada tuplo obtido pela plan tree da pergunta e devolver o resultado ao cliente;
- INSERT - Cada tuplo devolvido é inserido na tabela especificada pela pergunta, sendo isto feito num nó especial chamado ModifyTable;
- UPDATE - É garantido que cada tuplo incluí os valores actualizados e também o id do tuplo original, sendo que esta informação é passada para um nó ModifyTable, que irá criar um novo tuplo com os dados actualizados e assinalar o antigo como apagado.
- DELETE - Para esta operação são retornados os ids dos tuplos a serem apagados, sendo estes valores passados para o nó ModifyTable que irá marcar cada tuplo devolvido, como apagado.

4.6 Ver planos de perguntas

O PostgreSQL permite ver os planos de execução das perguntas feitas através do comando EXPLAIN. Estes comando mostra o plano de execução gerado pelo planeador para a pergunta especificada, mostrando como as tabelas, presentes na pergunta, vão ser percorridas e que algoritmos de join vão ser utilizados.Para além disso, também mostra o custo estimado de execução, sendo retornado dois valores de custo, o custo de inicialização e o custo total. O custo de inicialização é o custo até ser retornado o primeiro tuplo do resultado da pergunta, sendo o custo total o custo de retornar todos os tuplos

resultantes da pergunta.

A estrutura do comando é a seguinte:
EXPLAIN [ANALYZE] [VERBOSE] statement

A opção ANALYZE, faz com que a pergunta seja executada e não seja apenas calculado o custo da execução da pergunta, e é mostrado o custo real da execução e não uma estimativa.
A opção VERBOSE permite mostrar mais informação sobre o plano, como o output de cada nó durante a execução, as funções usadas e mais algumas estatísticas.
Este comando pode ser usado para qualquer tipo de pergunta, sendo que é indicado que caso se deseje usar a opção ANALYZE, em qualquer pergunta que possa alterar os dados é recomendado que seja feito numa transação com rollback no fim de maneira que os dados não sejam alterados.
Outra funcionalidade deste comando é permitir que tipo de output pretendemos, podendo ser texto, XML, JSON ou YAML, sendo o valor por defeito texto.

5 Gestão de transacções e controlo de concorrência

O PostgreSQL trata cada pergunta como uma transacção, sendo que de maneira a executar um bloco de perguntas como uma transacção é preciso iniciar o bloco com a keyword BEGIN e terminar com END.
O PostgreSQL também define um mecanismo de savepoints, onde é possível seleccionar partes de uma transacção para onde se pode fazer roll back, de maneira a tratar de erros da transacção, sem ser necessário ter que a refazer completamente. O PostgreSQL não permite o uso de nested transactions, sendo usado o mecanismo de savepoints, para se obter este comportamento.
De maneira a gerir a concorrência entre várias transacções, o PostgreSQL utiliza o modelo multiversão, ou Multiversion Concurrency Control(MVCC), isto significa que cada transacção vê uma snapshot da base de dados, de maneira a ver dados consistentes da base de dados, ou seja, não vê dados que tenham sido alterados por outras transacções concorrentes. A vantagem de usar este modelo é que locks obtidos para leitura não entram em conflito com locks obtidos para escrita, assim estas operações nunca se bloqueiam uma à outra. O PostgreSQL garante este isolamento através do uso do sistema Serializable Snapshot Isolation.
Como o modelo MVCC obriga o sistema a guardar várias versões da base de dados, isto leva a um consumo excessivo do espaço em disco. O PostgreSQL liberta espaço em disco sempre que encontra versões de tuplos ao qual nenhuma transacção consegue aceder, sendo que esta acção pode ser chamada explicitamente através do comando VACUUM, pelo utilizador. É possível também correr o comando VACUUM FULL, que reescreve o conteúdo das tabelas num novo ficheiro, permitindo que o espaço inutilizado possa ser utilizado posteriormente. Este processo é mais lento e obriga ao uso de locks exclusivos em cada tabela enquanto executa.

5.1 Níveis de isolamento

O SQL define quatro níveis de isolamento, read uncommited,read commited, repeatable read e serializabel, sendo possível escolher qualquer um destes níveis de isolamento no PostgreSQL, através do comando SET TRANSACTION. No entanto apenas existem três níveis de isolamento distintos, no PostgreSQL, visto que quando é seleccionado o modo read uncommited, na realidade é usado o read commited.

5.2 Read commited

O nível de isolamento read commited é o nível usado por defeito no PostgreSQL. Quando é utilizado este nível de isolamento uma pergunta do tipo SELECT vai ver apenas as alterações que foram feitas commit antes de iniciar a sua execução. Ou seja, a pergunta vai ver um snapshot da base de dados do instante em que começou a sua execução, no entanto vê as alterações feitas anteriormente na sua transacção.
No caso da transacção tentar actualizar um tuplo que entretanto já foi alterado por outra transacção, vai ficar à espera que a transacção concorrente faça commit, do que

foi alterado, ou roll back, sendo que caso seja feito commit, a transacção vai utilizar os valores alterados, ou caso o tuplo tenha sido eliminado, ignora-o, no caso de ter sido feito roll back, então as alterações feitas são eleminadas e a transacção pode actualizar o valor encontrado originalmente.

5.3 Repeatable Read

Neste nível de isolamento, a transacção apenas vê alterações, feitas por outras transacções que fizeram commit, antes de ter começado, sendo que as únicas alterações, à base de dados, que vê são aquelas que ela própria fez. Para obter este isolamento, os snapshots são obtidos no inicio da transacção, daí se forem feitos dois comandos SELECT seguidos irão obter resultados iguais.
No caso da transacção tentar actualizar um tuplo que foi actualizado por uma transacção concorrente, esta vai ter que esperar que a transacção concorrente faça commit ou roll-back. No caso da transacção concorrente fizer commit, a transacção original tem de fazer roll back, de maneira a poder ver as alterações feitas pela transacção concorrente. No caso de ter sido feito roll back, a transacção original pode continuar, visto que significa que as alterações feitas pela transacção concorrente foram negadas.

5.4 Serializable

Este nível de isolamento simula a execução de transacções serializadas como se tivessem executa uma à seguir à outra e não concorrentemente. Este modo comporta-se da mesma maneira que o repeatable read, com a diferença de que é necessário verificar condições extra para se impedir que as transacções serializáveis se comportem de maneira inconsistente.
De maneira a garantir a serialização, o PostgreSQL, utiliza predicate locking, significa que guarda locks que permitem determinar quando uma escrita tem impacto nos resultados de uma leitura feita anteriormente por uma transacção concorrente. Estes locks não causam nenhum bloqueio e por isso não podem provocar deadlocks.

5.5 Locks

O PostgreSQL oferece vários modos de locks de maneira a controlar o acesso concorrente à base de dados, sendo que estes são usados quando o MVCC não apresenta o comportamento esperado. Para além disso, alguns comandos adquirem locks automaticamente de maneira a garantir que as tabelas não são apagadas ou modificadas de maneira incompatível.

5.6 Locks ao nível da tabela

Existem vários tipos de lock que afectam a tabela inteira, apesar de alguns conterem a palavra row, sendo que isto apenas acontece por razões históricas. Alguns dos modos de lock são incompatíveis entre si, ou seja, não é possível duas transacções diferentes terem locks incompatíveis na mesma tabela. No entanto uma transacção pode obter vários

locks incompatíveis visto que nunca entra em conflito com si mesma.

Os modos de lock, possíveis ao nível de tabela, são os seguintes:

- ACCESS SHARE – Perguntas que apenas leiam a tabela obtém locks deste tipo, como por exemplo uma pergunta do tipo SELECT.
- ROW SHARE – Os comandos SELECT FOR UPDATE e SELECT FOR SHARE obtêm este tipo de locks nas tabelas referenciadas.
- ROW EXCLUSIVE – É obtido pelos comandos UPDATE, DELETE e INSERT, nas tabelas que vão ser alteradas.
- SHARE UPDATE EXCLUSIVE – Este modo protege contra alterações de esquema e de execuções do comando VACUUM.
- SHARE – Protege a tabela contra alterações concorrentes aos dados, sendo obtido pelo comando CREATE INDEX.
- SHARE ROW EXCLUSIVE – Protege contra alterações concorrentes aos dados da tabela, sendo que apenas uma sessão pode obter o lock. Este tipo de lock não é obtido automaticamente por nenhum comando PostgreSQL.
- EXCLUSIVE – Este tipo de lock apenas permite leituras à tabela em paralelo, com a transacção que obteve o lock. Este tipo de lock também não é obtido automaticamente por nenhum comando PostgreSQL.
- ACESS EXCLUSIVE – Este tipo de lock garante que a transacção que o obteve é a única que tem acesso à tabela. Obtido através dos comando ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER e VACUUM FULL. Este é o modo por defeito do comando LOCK TABLE quando não é especificado nenhum tipo de lock.

Tipos de locks	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE
ACCESS SHARE				
ROW SHARE				
ROW EXCLUSIVE				
SHARE UPDATE EXCLUSIVE				X
SHARE			X	X
SHARE ROW EXCLUSIVE			X	X
EXCLUSIVE		X	X	X
ACCESS EXCLUSIVE	X	X	X	X

Tipos de locks	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
SHARE		X	X	X
SHARE ROW EXCLUSIVE	X	X	X	X
EXCLUSIVE	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X

5.7 Locks ao nível de tuplo

Os locks ao nível de tuplo podem ser do tipo shared ou exclusive. Os lock do tipo exclusive podem ser obtidos quando um tuplo é actualizado ou apagado, sendo que, tal como com locks ao nível da tabela, este lock só é libertado quando a transacção faz commit ou rollback, sendo que apenas ficam bloqueadas as escritas no mesmo tuplo.
Um exclusive lock pode ser obtido através do comando SELECT FOR UPDATE sem ser realmente alterado, sendo que assim que este for obtido, a transacção pode actualizar o tuplo várias vezes sem problemas de conflito.
De maneira a se obter um shared lock num tuplo, selecciona-se o tuplo com o comando SELECT FOR SHARE. Este tipo de lock permite a outras transacções lerem o tuplo, no entanto não lhes é permitido fazer qualquer tipo de alteração, ou obter um exclusive lcok ao mesmo.
Como o PostgreSQL não guarda nenhuma informação, sobre os tuplos alterados, em memória, não existe limite para o número de tuplo sobre os quais se pode obter lock, no entanto podem causar escritas em disco, por exemplo o comando SELECT FOR UPDATE modifica os tuplos de maneira a marcá-los como bloqueados, o que resulta em escritas em disco.

5.8 Deadlocks

De maneira a detectar se ocorreu deadlock, o PostgreSQL utiliza um mecanismo baseado em timeouts, ou seja, caso a espera ,de uma transacção, por um lock demore mais do que um segundo, por defeito, vai ser chamado o algoritmo de detecção de deadlock. Este algoritmo cria um grafo de esperas por lock entre as transacções, sendo que, depois de ser

criado o grafo, vai detectar se existe algum ciclo, é feito rollback de uma das transacções que contribuem para o deadlock.

5.9 Advisory Locks

Para além dos locks normais, o PostgreSQL oferece também advisory locks, que são específicos para cada aplicação, não sendo forçado o seu uso pelo sistema. Existem dois tipos de advisory locks ao nível de sessão e ao nível de transacção.
Quando um advisory lock é obtido ao nível de sessão, este é mantido até ser libertado explicitamente ou quando a sessão terminar. Ao contrário dos locks normais, um lock deste tipo, não é libertado quando é feito o rollback de uma transacção, mas pode o ser se for libertado durante a transacção e esta falha. Em relação aos lock obtidos ao nível de transacção, estes comportam-se como lock normais, sendo libertados no fim da transacção. Pedidos de advisory locks ao nível de sessão e de transacção, ao mesmo lock bloqueiam-se da maneira esperada.

5.10 Consistência

De maneira a se garantir a consistência da base de dados existem duas formas, obrigar a utilização do modo de isolamento serializable, ou o uso de locks explícitos. Ambos estes modos garantem consistência, no entanto limitam seriamente a execução concorrente de transacções.
A verificação das restrições que garantem consistência pode ser alterada através do comando SET CONSTRAINTS, podendo ser do tipo DEFERRED ou IMMEDIATE, sendo que o primeiro faz com que as restrições sejam verificadas após cada pergunta, enquanto que o outro faz com que as restrições apenas sejam verificadas depois do commit da transacção.
Ao ser criada uma restrição esta pode ser definida de uma de três formas, DEFERRABLE INITIALLY DEFERRED,DEFERRABLE INITIALLY IMMEDIATE, ouNOT DEFERRABLE. Nos dois primeiros modos a verificação pode ser diferida para o fim da transacção sendo que caso seja DEFERRED esse é o modo por defeito, e caso seja IMMEDIATE é verificado após cada execução de uma pergunta. No caso de NOT DEFERRABLE, a verificação é feita após cada pergunta, não sendo possível adiar esta verificação para o fim da transacção.

6 Suporte para bases de dados distribuídas

O postgreSQL suporta a implementação de várias soluções distintas para este problema, sendo que as que a própria documentação [1] referencia as seguintes:

- *Shared Disk Failover* – usando geralmente uma *Network-attached storage*, tendo em conta que o postgreSQL considera que os NAS (com funcionamento interno de *Network File System*), age tal e qual uma *Direct Attached Storage*; esta solução utiliza somente uma cópia da base de dados, pelo que não tem um *overhead* de sincronização (falhando o servidor principal da base de dados, é chamado o servidor secundário) mas tem limitações pois se o disco falha ou fica corrupto, ambos os servidores ficam não funcionais;

- *File System (Block-Device) Replication* – principalmente usando um *Distributed Replicated Block Device* para realizar esta replicação;

- *Transaction Log Shipping* – usando *Streaming Replication* [2] ou *file-based log shipping* (ou uma junção dos dois. Este método consiste em fazer servidores *warm* e/ou *hot* permanecerem actualizados lendo um *stream* de registos de *write-ahead log* (WAL), de tal forma que se o servidor principal falhar, estes podem-se tornar no novo servidor principal com bastante rapidez;

- *Trigger-Based Master-Standby Replication* – usando Slony-I e/ou linguagens processuais (referido brevemente na próxima secção), actualiza para os servers em *standby* em *batches*, pelo que pode ocorrer perda de dados durante um *fail over*;

- *Statement-Based Replication Middleware* – utilizando de preferência *two-phase commit*, tanto com a ferramenta Pgpool-II como Continuent Tungsten;

- *Asynchronous Multimaster Replication* – usado geralmente para dispositivos móveis com tempos de conecção limitada, os servidores do dispositivo comunicam com os outros utilizando um critério "quando possível"; as resoluções de conflitos podem ser resolvidas pelos próprios utilizadores (não recomendado a não ser para casos específicos) ou com ferramentas próprias para estes conflitos. O Bucardo é um exemplo de ferramenta deste tipo de replicação;

- *Synchronous Multimaster Replication* – embora o postgreSQL não ofereça este tipo de replicação, o *two-phase commit* pode ser usado para implementar tal em código da aplicação ou *middleware*.

  Havendo também outras soluções comerciais visto o postgreSQL ser *open source*.

Referido anteriormente, a todos os momentos o postgreSQL mantém um WAL para garantir a integridade dos dados, este *log* regista todas as mudanças feitas aos ficheiros de dados da base de dados, sendo que existe primáriamente para segurança contra *crashes*; quando o sistema falha a base de dados restora a consistência ao repetir as

entradas do log feitas desde o último *checkpoint*. Contúdo, este *log* faz com que seja possível também usar estratégias de *backup*, suportar *backup online, point-in-time recovery.* [3][4]
É referido também *Data Partitioning*, em que os dados podem estar divididos entre vários servidores, sendo que quando é pedido dados de ambos os servidores, podem ser ambos *querried* ou então cada um deles pode ter um uma replicação *master/standby* do(s) seu(s) homólogos em que guardam uma cópia *read-only*; e *Multiple-Server Parallel Query Execution* (executar a mesma *query* em servidores distintos, concorrentemente) em que a ferramenta pgpool-II ou PL/Proxy pode ser usada.
De forma a criar uma *high availability* o postgreSQL faz um armazenamento contínuo em diversos servidores em *standby*, caso o primário falhe. A esta capacidade denomina-se de *log shipping* ou *warm standby* [5] (como referido anteriormente), usado como substituição ao método antigo (e se necessário ainda utilizável), restore_command [6].
O postgreSQL contém também parâmetros para a realização de *hot standby*, utilizando um servidor remoto que recebe somente pedidos de leitura e sendo actualizado (eventualmente) a partir do servidor principal. Esta técnica é interessante pois serve tanto para replicação como para *backup* [7].
Referido em alguns pontos anteriormente, se o servidor primário falha, entao um dos servidores em *standby* pode retomar o controlo, denominando-se tal de *failover*. Embora o postgreSQL não forneça maneiras de identificar uma falha deste género, tanto os sistemas operativos, como muitos outros componentes em que assenta têm ferramentas para tal, como os estudos e teses referentes a sistemas distribuidos podem comprovar. A mudança de servidor perante *failover* é realizada mediante *triggers* (discutidos brevemente na próxima secção).
Segue-se uma lista das várias ferramentas utilizadas para a distribuição do postgreSQL e os métodos de replicação e sincronização [8]:

- PgCluster – *master to master; synchronous*;
- pgpool-I – *statement-based middleware; synchronous*;
- Pgpool-II - *statement-based middleware; synchronous*;
- slony - *master-slave; asnchronous*;
- Bucardo - *master-master, master-slave; asynchronous*;
- Londiste - *master-slave; asynchronous*;
- Mammoth - *master-slave; asynchronous*;
- rubyrep - *master-master, master-slave; asynchronous.*

De notar que embora não referido na documentação do opengreSQL, postgresXC (entre outros) suportam a replicação *multi-master* [9] [10].

7 Outras características

7.1 Procedural Languages

Referido anteriormente, o postgreSQL permite a definição de funções por utilizadores que não sejam somente realizadas em SQL e C, estas linguagens são genericamente designadas por procedural languages. Há quatro PLs disponíveis na versão *standard* do postgreSQL: PL/pgSQL, PL/Tcl, PL/Perl e PL/Python (a qual pode ser implementada em Python 2 ou Python3), havendo também outras que estão disponíveis para utilização ou mesmo a possibilidade de se definir novas PLs pelo utilizador [11].

Um exemplo de uma função em PL/Python poderá ser o seguinte, que retorna o maior de dois inteiros [12]:

```
CREATE FUNCTION pymax (a integer, b integer)
RETURNS integer
AS $$
if a > b:
return a
return b
$$ LANGUAGE plpythonu;
```

E o mesmo exemplo mas em PL/Perl seria o seguinte [13]:

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
if ($_[0] > $_[1])  return $_[0];
return $_[1];
$$ LANGUAGE plperl;
```

7.2 Triggers

Estas linguagens e a linguagem C (mas não o *plain* SQL) podem também ser usadas para a criação de *triggers* e *event triggers* que são *triggers* globais aplicados a uma determinada base de dados e são capazes de capturar eventos de *Data Definition Language* [14].

Um exemplo de um *event trigger* poderá ser o seguinte, que ergue um NOTICE sempre que um comando suportado é executado [15]:

```
CREATE OR REPLACE FUNCTION snitch() RETURNS event_trigger AS $$
BEGIN
RAISE NOTICE 'snitch:
END;
$$ LANGUAGE plpgsql;
```

CREATE EVENT TRIGGER snitch ON ddl_command_start EXECUTE PROCEDURE snitch();

E um exemplo da utilização de um *trigger* usando PL/Tcl que obriga um valor inteiro a incrementar sempre que se faz uma actualização na linha que lhe corresponde, sendo o mesmo inicializado a zero [16]:

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
switch $TG_op
INSERT
set NEW($1) 0

UPDATE
set NEW($1) $OLD($1)
incr NEW($1)

default
return OK


return [array get NEW]
$$ LANGUAGE pltcl;

CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

7.3 Serviços de Acesso, Consulta e Transmissão

O postgreSQL implementa ODBC (*Open Database Connectivity*) e também JDBC (*Java Database Connectivity*) [17] [18]; implementa funções de XML, podendo-se configurar a base de dados com libxml para se usar as expressões xmlparse e xmlserialize, para converter para e de tipo XML [19]; implementa também funções e operadores JSON [20].

7.4 Segurança

Segurança em postgreSQL é realizada a partir de *roles*. Um *role* é, em termos genéricos, um *user* ou um conjunto de *users*, dependendo de como se pretendes gerenciar as permissões. *Roles* podem ter distintos objectos e estas permissões podem ser concedidas ou revogadas a outros *roles*.

Em concordância com os *roles*, somente *superusers* podem ter privilegios de criar *triggers* e funções noutras linguagens, pois esses elementos têm acesso completo à base de dados e podem contornar estes privilégios [21].

7.5 Autenticação

O postgreSQL usa também métodos de autenticação externos tais como *passwords*, certificados, GSSAPI, etc. para conecções de clientes externos com a base de dados [22].

7.6 Aplicações Administrativas

Contidos em alguns dos instaladores vêm algumas ferramentas administrativas, duas importantes são o pgAdmin III e o StackBuilder. O pgAdmin III [23], tal como muitas outras [24], por exemplo o phppgadmin [25], é uma ferramenta de interface gráfica de administração que permite escrever simples *queries* de SQL para desenvolver bases de dados complexas. O StackBuilder é um gestor de pacotes que pode ser usado para fazer o *download* e instalar aplicações de *drivers* para o postgreSQL [26].

8 Comparação com o Oracle 11g

8.1 Armazenamento e file structure

No que toca a este tópico, existem semelhanças e diferenças entre as bases de dados PostgreSQL e as Oracle 11g.
Ambos os sistemas utilizam para organização dos tuplos um heap. Mas por exemplo, no que toca ao sistema de ficheiros, os dois SGBD distinguem-se, sendo que o sistema Oracle tem o seu próprio sistema de ficheiros implementado, enquanto que o PostgreSQL não. O facto de não ter diminui a complexidade de implementação do sistema PostgreSQL, mas deixa-o vulnerável aos prós e contras do sistema de ficheiros do sistema operativo. Enquanto q ue em Oracle, existe maior controlo, e consequentemente melhor performance.
No que toca a clustering, como já foi vito o PostegreSQL permite clustering por tabela, o memso se passa com o Oracle. No entanto o Oracle implementa uma funcionalidade de extra de permitir fazer clustering em múltiplas tabelas, isto vai permitir alcançar performances muito boas em certos tipos de queries, nomeadamente, aquelas que envolvem junções que sejam frequentes.
Caso se queira fazer o particionamento de tabelas, o PostgreSQL oferece duas formas de o fazer: Range Partitioning e List Partitioning. Já o Oracle oferece mais opções neste ramo. Para além das duas formas de particionamento que o PostgreSQL oferece, o Oracle ainda oferece mais dois tipos: Hash Partitioning e Composite Partitioning. Sendo que este último, é a combinação entre Range e Hash Partitioning.

8.2 Indexação e hashing

Ambos os sistemas criam por defeito indexes B+-tree. No entanto outros tipos de indexes estão disponíveis. No que toca a indexes O PostgreSQL implementa mais tipos de que o Oracle, sendo que já foram falados os principais, mas no total, estes são os tipos que o PostgreSQL implementa: GIN, GiST, Bitmap, Expression, Full-text, Hash, Partial, R-/R+ Tree, Reverse. Já o Oracle, de entre estes, deixa de fora os indexes GIN e GiST. Para executar um index não é necessário sequer indicar qual o index que se pretende utilizar, o próprio PostgreSQL escolhe um apropriado, para a execução rápida das queries.

8.3 Controlo de concorrência

A diferença entre o PostgreSQL e o Oracle 11g, é que o primeiro usa sempre um mecanismo de snapshot para as suas transacções independentemente do nível de isolamento escolhido para correr a transacção.

8.4 Suporte para bases de dados distribuídas

Embora o suporte para bases de dados distribuídas tenha sido implementado a partir da versão 9, esse facto torna até mais importante este ponto, visto em poucas versões

ter-se conseguido desenvolver (graças também ao facto de ser *opensource*) uma grande colecção de ferramentas para distribuição e mesmo que em alguns pontos possa não ser tão robusto como o Oracle, com algum esforço conseguem-se replicar práticamente as mesmas funcionalidades.

9 Referências

[1] - maioritáriamente derivada de http://www.postgresql.org/docs/9.4/static/different-replication-solutions.html
[2] - implementação em http://www.postgresql.org/docs/current/static/warm-standby.html#STREAMING-REPLICATION
[3] - http://www.postgresql.org/docs/9.4/static/wal-intro.html
[4] - http://www.postgresql.org/docs/9.4/static/continuous-archiving.html
[5] - http://www.postgresql.org/docs/9.4/static/warm-standby.html
[6] - http://www.postgresql.org/docs/9.4/static/log-shipping-alternative.html
[7] - http://www.postgresql.org/docs/9.4/static/hot-standby.html
[8] - http://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling
[9] - http://en.wikipedia.org/wiki/Multi-master_replication#PostgreSQL
[10] - http://postgres-xc.github.io/1_0/intro-whatis.html
[11] - http://www.postgresql.org/docs/9.4/static/xplang.html
[12] - http://www.postgresql.org/docs/9.4/static/plpython-funcs.html
[13] - http://www.postgresql.org/docs/9.4/static/plperl-funcs.html
[14] - http://www.postgresql.org/docs/9.4/static/event-triggers.html
[15] - http://www.postgresql.org/docs/9.4/static/plpgsql-trigger.html
[16] - http://www.postgresql.org/docs/9.4/static/pltcl-trigger.html
[17] - http://www.postgresql.org/ftp/odbc/versions/
[18] - http://jdbc.postgresql.org/
[19] - http://www.postgresql.org/docs/9.3/static/functions-xml.html
[20] - http://www.postgresql.org/docs/9.3/static/functions-json.html
[21] - http://www.postgresql.org/docs/9.4/static/user-manag.html
[22] - http://www.postgresql.org/docs/9.4/static/client-authentication.html
[23] - http://www.pgadmin.org/
[24] - http://wiki.postgresql.org/wiki/Community_Guide_to_PostgreSQL_GUI_Tools
[25] - http://phppgadmin.sourceforge.net/doku.php
[26] - http://www.postgresql.org/download/windows/
[27] - http://www.postgresql.org/docs/9.3/static/storage.html
[28] - http://www.postgresql.org/docs/9.3/static/ddl-partitioning.html
[29] - http://www.postgresql.org/docs/9.3/static/sql-cluster.html
[30] - http://www.postgresql.org/docs/9.3/static/indexes.html
[31] - http://www.postgresql.org/docs/9.3/static/overview.html
[32] - http://www.postgresql.org/docs/9.3/static/querytree.html
[33] - http://www.postgresql.org/docs/9.3/static/geqo-pg-intro.html
[34] - http://www.postgresql.org/docs/9.3/static/sql-explain.html
[35] - http://www.postgresql.org/docs/9.3/static/mvcc.html