Faculdade de Ciências e Tecnologia

Sistemas de Bases de Dados

Análise do Sistema PostgreSQL

Grupo 21

Carlos Lopes
Nº 42408

Nuno Correia
Nº 41841

Nuno Fernandes
Nº 41728

Docentes
José Júlio Alferes
Ricardo Silva

1 de Junho de 2014

Conteúdo

1 Introdução

Este relatório visa estudar o sistema de gestão de bases de dados PostgreSQL e como se relaciona com a matéria estudada na unidade curricular.

O primeiro capítulo serve como introdução ao sistema. Os capítulos 2-6 relacionam-se diretamente com a matéria estudada, incluindo exemplos e comparações com o Oracle. O capítulo 7 é um resumo de curiosidades sobre o sistema descobertas ao longo do desenvolvimento deste relatório.

Salvo indicação de contrário, os exemplos encontrados ao longo do documento foram executados sobre a base de dados Mondial disponível *online* à data em `www.dbis.informatik.uni-goettingen.de/Mondial/`

Para a elaboração do relatório foi estudada a versão 9.3 do PostgreSQL.

1.1 Breve história do PostgreSQL

O PostgreSQL é um sistema de gestão de bases de dados *open source* baseado no sistema Postgres 4.2, desenvolvido no Berkeley Computer Science Department da Universidade da California.

O PostgreSQL é descendente do código fonte original de Berkeley e suporta uma grande parte do *standard* SQL.

1.1.1 O Projeto Berkeley Postgres

A implementação do projeto Postgres teve o seu inicio em 1986 sob a liderança do Professor Michael Stonebraker. O projeto era patrocinado pela Defense Advanced Research Projects Agency (DARPA), pelo Army Research Office (ARO), pela National Science Foundation (NSF) e pela ESL, Inc. Várias versões do sistema foram lançadas até à data do término do seu desenvolvimento em 1994. A última versão do sistema é a 4.2.

1.1.2 Postgres95

Em 1994 Andrew Yu e Jolly Chen adicionaram um interpretador SQL ao Postgres. Sob um novo nome, o Postgres95 foi lançado na *web* como o descendente *open-source* do código fonte original de Berkeley. O projeto inclui várias características importantes que permitiram um desempenho 30-50% mais rápido que o Postgres.

O foco de desenvolvimento centrou-se em identificar e compreender os problemas existentes no código do servidor.

1.1.3 PostgresSQL

Em 1996 tornou-se claro que o nome Postgres95 não iria envelhecer graciosamente. Como tal tomou-se a decisão de renomear o projeto para PostgreSQL refletindo a sua nova relação com a capacidade de interpretar SQL. O versionamento mudou, também, para seguir a linha iniciada pelo projeto Postgres. A nova versão era assim a 6.0.

O foco de desenvolvimento centrou-se em acrescentar funcionalidade e capacidade.

2 Armazenamento e *File Structure*

2.1 Organização de ficheiros

Existem vários mecanismos para armazenamento de ficheiros em bases de dados. No PostgreSQL, este mecanismo é o sistema de ficheiros do sistema operativo. Na terminologia associada ao PostgreSQL, é usual designar uma tabela por *heap relation*, e um índice por *index relation*. Nesta secção detalha-se a forma como estas relações são armazenadas e o seu lugar no sistema de ficheiros.

Como consequência de não implementar um sistema de ficheiros próprio, o PostgreSQL faz uso de ficheiros separados para efeitos de armazenamento de cada relação, por forma a aproveitar ao máximo as funcionalidades do sistema de ficheiros do sistema operativo.

Em particular isto significa que, ao contrário do que se passa no Oracle, não existe a opção de armazenar várias tabelas num mesmo ficheiro (*multitable clustering*), funcionalidade acessível no Oracle através do comando **CREATE CLUSTER**.

Internamente, o PostgreSQL guarda toda a informação necessária para aceder fisicamente a uma relação na estrutura *RelFileNodeBackend*, disponível em: `src/include/storage/relfilenode.h`.

```
typedef struct RelFileNodeBackend {
    RelFileNode node;
    BackendId   backend;
} RelFileNodeBackend;
```

com a estrutura *RelFileNode* definida como:

```
typedef struct RelFileNode {
     Oid         spcNode;        /* tablespace */
     Oid         dbNode;         /* database */
     Oid         relNode;        /* relation */
 } RelFileNode;
```

onde *Oid* é um *unsigned int*. Estas estruturas são relevantes para o contexto de armazenamento de ficheiros detalhado em seguida. Na coluna *relfilenode* (*relation filenode*) do catálogo *pg_class* está presente para cada relação, o que em PostgreSQL se designa o *filenode number*. O *filenode number*, é simplesmente o valor do campo *relNode* da estrutura *RelFileNode*. Os campos *relNode* e *backend* dão o nome aos ficheiros das relações da seguinte forma:

- relações não temporárias têm como nome *filenode*
- relações temporárias têm como nome o formato *t <backend>_<filenode>*

A observação das estruturas permite ainda aferir que o *filenode* de uma relação numa base de dados não é único, uma vez que o *tablespace* da relação pode ser distinto. Isto justifica o fato de uma relação ficar associada a um *tablespace* assim que é

definida (o caso das tabelas particionadas requer uma reformulação/clarificação desta afirmação, mas o principio é o mesmo).

Devido à utilização do sistema de ficheiros do sistema operativo, existem limites associados à dimensão dos ficheiros, que são dependentes da plataforma. Por este motivo, o PostgreSQL, por *default*, particiona relações que excedam 1GB, em segmentos com esta dimensão máxima. O ficheiro com o primeiro segmento recebe o nome do filenode da relação e os ficheiros com os segmentos seguintes são renomeados sucessivamente como filenode1, filenode2,... Esta dimensão é segura em todas as plataformas em que o sistema PostgreSQL é suportado.

Há aqui mais um pequeno *tradeoff* que vale a pena observar: quanto maior a dimensão dos segmentos, menor o número potencial de *file descriptors* consumidos ao trabalhar com uma tabela grande. Em sistemas com suporte a ficheiros de grande dimensão, pode compensar aumentar este valor. O PostgreSQL permite a configuração deste parâmetro usando:

```
--with-segsize=SEGSIZE
```

aquando da instalação e configuração do servidor.

Ao contrário do PostgreSQL, o Oracle disponibiliza um mecanismo designado por *Oracle Automatic Storage Management* (Oracle ASM) que, entre outras funcionalidades, implementa um sistema de ficheiros, desenhado especificamente para a base de dados. Internamente, o sistema operativo aloca ficheiros de sistema a pedido do ASM, que os gere de forma transparente. Todos os objetos da base de dados são assim armazenados fisicamente no que se designam por *datafiles*.

A solução implementada pelo Oracle beneficia de mais valias como segurança reforçada, maior consistência e menor redundância de dados, rapidez de acesso, entre outras. Por outro lado, a implementação de um sistema de ficheiros dedicado torna o sistema mais complexo, tendo nomeadamente, um impacto negativo no *start-up* do mesmo. Não pode ainda deixar de se considerar que exige conhecimento sobre uma componente adicional, no mínimo, por parte do programador.

Por último, existem custos monetários de R&D associados ao desenvolvimento, manutenção e melhoramento de um sistema de ficheiros personalizado que não podem ser ignorados. Ao usar o sistema de ficheiros do sistema operativo, beneficia-se automaticamente do constante desenvolvimento deste que, ainda que não otimizado para bases de dados, garante uma *performance* muito aceitável neste contexto.

2.2 Tablespaces

Relembre-se que o Oracle guarda os dados, logicamente, em *tablespaces*, e fisicamente, em *datafiles* associados aos *tablespaces*.

Em PostgreSQL os *tablespaces* definem localizações físicas (diretorias), no sistema de ficheiros, onde os ficheiros que representam os objetos da base de dados são guardados (catálogos, tabelas, índices, ...).

Quando o *cluster* é inicializado, são criados dois *tablespaces*:

pg_global para objetos de sistema partilhados (i.e. comuns a todas as bases de dados contidas no cluster)

pg_default para uso das bases de dados *template0* e *template1* (e consequentemente para todas as bases de dados criadas pelo utilizador)

Por omissão da cláusula **TABLESPACE** no comando **CREATE DATABASE**, é usado o *pg_default*.

O catálogo de sistema *pg_tblspc* (por *default* encontrado em *PGDATA*), indica quais os *tablespaces* definidos na base de dados. A implementação de *tablespaces* em PostgreSQL é feito com base em *symbolic links*, pelo que estes só podem ser usados em sistemas que tenham suporte para tal. O parâmetro default_tablespace, quando incializado, indica qual o *tablespace* a usar por omissão em comados como **CREATE TABLE** e **CREATE INDEX**, e pode ser parametrizado com:

```
set default_tablespace = <tablespace_name>;
```

2.3 Partições

O PostgreSQL suporta a divisão física de uma tabela lógica via herança de tabelas. Uma partição é apenas uma tabela, criada como descendente de uma única tabela (*master table*). A tabela da qual as tabelas partição herdam, permanece, regra geral, vazia. São fornecidas duas formas básicas de particionamento:

partição por intervalos (range partitioning) a partição é gerada de acordo com um conjunto de intervalos, mutuamente disjuntos, de valores de uma coluna chave ou um conjunto de colunas.

partição por enumeração (list partitioning) a partição é gerada de forma explícita, definindo quais os valores chave que aparecem em cada partição (o que pode ser feito via *constraints* sobre as partições).

O particionamento também é oferecido pelo Oracle, uma vez que é uma funcionalidade que potencia ganhos significativos em performance e flexibilidade de gestão. Numa nota aparte, o Oracle oferece ainda particionamento de índices, de vistas materializadas e de índices sobre vistas materializadas.

Considerações sobre as vantagens de cada tipo de particionamento têm relevo no campo da otimização de queries. No contexto de armazenamento, o que importa perceber é que o particionamento de tabelas como é implementado, leva à criação de tanto ficheiros quanto partições.

Organização de Registos

O PostgreSQL aloca dados num ficheiro em unidades designadas por páginas. Este termo é usado, neste contexto, com o mesmo significado de bloco. Por motivos de eficiência, é esperado que a dimensão de uma página/bloco do PostgreSQL seja um múltiplo, não muito elevado, da dimensão do bloco do sistema operativo (e portanto também um múltiplo da dimensão dos setores do disco).

Por *default* este valor é de 8K, e é adequado à generalidade das situações. Este parâmetro é no entanto parametrizável aquando do setup do servidor, recorrendo ao comando:

`--with-blocksize=BLOCKSIZE`

Nesta secção detalha-se a estratégia usada para organizar registos de uma relação numa página.

À semelhança do Oracle, o PostgreSQL usa a técnica da **slotted-page** para este efeito. Esta escolha facilita o uso de registos de dimensão variável. Existe uma secção da *slotted-page* designada por "*special space*". Esta área pode ou não ser aproveitada para guardar dados consoante o tipo de relação a que os registos na página dizem respeito (detalhado adiante).

É também usual designar as entradas das páginas por **items**. Se a página armazena *items* de uma tabela (tuplos) é designada por *heap page*, e se armazena *items* de um índice (entradas do índice) designa-se por *index page*. Note-se que os dois tipos de *item* não se misturam numa mesma página.

Relativamente à organização de tuplos no ficheiro, numa tabela, as páginas são equivalentes do ponto de vista lógico e portanto um tuplo pode encontrar-se guardado em qualquer página, i.e., o PostgreSQL implementa, também à semelhança do sistema Oracle, uma **heap file organization**.

A secção final, o já referido "*special space*", pode conter o que quer que seja que os métodos de acesso necessitem de armazenar. Como exemplo, no caso de índices *B-Tree*, são guardados *links* para as páginas irmãs (*left/right siblings*) e outra informação relevante para a estrutura. Por outro lado, tabelas comuns (*heap relations*) não fazem uso da secção especial.

Tabelas TOAST

Tal como no Oracle, um registo tem de estar contido exclusivamente numa única *slotted-page*. No sistema Oracle, estes são conhecidos como **Large Objects (LOB)**.

O PostgreSQL disponibiliza um sistema de armazenamento alternativo que guarda, automaticamente, registos cuja dimensão excede aquela suportada pela da página, numa área de armazenamento à parte.

A técnica de armazenamento *TOAST* (**The Oversized-Attribute Storage Technique**) consiste em particionar e/ou comprimir o registo original em vários registos físicos, armazenando-os numa tabela (a tabela *TOAST*) associada à tabela a cujo registo original pertence (a técnica não é suportada por qualquer tipo de dados: é necessário ter uma representação de dimensão variável).

A tabela *pg_stat_all_tables* mantém uma linha para cada tabela na base de dados com as estatísticas de acesso a um dada tabela, incluindo as *TOAST*. Isto é a resolução de um problema anterior, onde a não contabilização do número de páginas de uma tabela *TOAST* poderia levar à não utilização de um índice.

Free Space Map (FSM)

Cada relação, *heap* ou *index* (com exceção para índices *hash*), tem associado um **FSM** que "mapeia"o espaço livre nas páginas da mesma.

Este mapa é armazenado na diretoria da relação a que está associado, ficando o seu nome definido pelo valor do *filenode* da relação concatenado com o sufixo "*_fsm*". Internamente, o *FSM* está organizado como uma árvore binária de páginas, permitindo

localizar rapidamente páginas de uma relação com espaço livre suficiente para efeitos de inserção de dados numa relação.

O intervalo de todos os valores possíveis para a quantidade de espaço livre numa qualquer página é particionado em subintervalos de igual dimensão (categorias), e o *FSM* usa apenas um *byte* por página para aproximar o valor real de espaço livre na mesma.

Note-se que usando 1 *byte*, a partição será formada por 256 intervalos disjuntos enquanto que a dimensão dos intervalos (diâmetro da partição) dependerá do tamanho de página, que internamente depende da arquitetura e do valor *BLCKSZ* (ver `src/include/pg_config.h`).

Visibility Map (VM)

Cada relação *heap*, tem associado um *VM* para mapear as páginas com tuplos que são visíveis para todas as transações. O *VM* é essencialmente um *bitmap*, e mantém um *bit* por cada página da *heap relation*. O mapa é conservativo no sentido em que:

bit a 1 todos os tuplos na página são visíveis por todas as relações.

bit a 0 pode ou não ser verdade que todos os tuplos são visíveis por todas as relações.

À semelhança do *FSM*, o *VM* é armazenado na diretoria da relação a que está associado, ficando o seu nome definido pelo valor do *filenode* da relação concatenado com o sufixo "*_vm*". O set dos *bits* do *VM* é sempre feito pela operação **VACUUM** (detalhada adiante), mas o *clear* pode ser feito por qualquer operação que modifique os dados na página (ligado às transações). O *VM* é usado, entre outros, no contexto dos designados *index-only scans*, uma funcionalidade disponibilizado recentemente, cuja implementação requereu tornar o *VM crash-safe*.

Manutenção - VACUUM e ANALYZE

Esta secção refere resumidamente duas estratégias de manutenção importantes para o bom funcionamento do sistema.

Observe-se que quando um tuplo é modificado na base de dados, o PostgreSQL limita-se a efetuar uma inserção do tuplo na sua versão modificada e a marcar o tuplo original como *dead tuple*. Processo idêntico é aplicado à remoção de tuplos. Em nenhum dos casos, o tuplo (versão antiga ou eliminado) é eliminado fisicamente da base de dados de imediato. Este comportamento está relacionado com o controlo de concorrência do PostgreSQL em MVCC e será detalhado no capítulo próprio.

A rotina **VACUUM** reclama espaço de armazenamento ocupado pelos chamados *dead tuples*, que permacencem na base de dados até esta rotina ser executada. Por este motivo, o **VACUUM** de tabelas frequentemente modificadas/atualizadas deve ser um procedimento habitual. Se não for fornecido nenhum parâmetro, o processo de **VACUUM** é aplicado a todas as tabelas a que o utilizador corrente tenha permissão para aplicar tal rotina. Parametrizando adequadamente é possivel aplicar **VACUUM** a apenas uma tabela.

O processo **ANALYZE** reúne estatísticas sobre os conteúdos das tabelas na base de dados e armazena os resultados no catálogo de sistema *pg_statistics*. Estas estatatísticas são importantes para o **query planner** (também detalhado no capítulo próprio) para determinar planos de processamento de *queries* eficientes. Dada a importância destes mecanismos, existe um comando `VACUUM ANALYZE` que combina os dois comandos pela ordem indicada. Para efeitos de estatísticas, é importante referir que para partições, os comandos `VACUUM` ou `ANALYZE` têm de ser aplicados a cada uma das partições individuais, e não à tabela da qual herdam (que é geralmente mantida vazia).

Buffer management

Tal como o Oracle, o PostgreSQL implementa o seu próprio sistema de **buffer management**.

Existe no entanto uma diferença relevante entre os dois sistemas, e esta deriva do facto de o PostgreSQL se apoiar no sistema de ficheiros do sistema operativo, o que se traduz em alguma redundância de funcionalidade e de dados.

Esta redundância é consequência da necessidade de ter uma *cache* própria para melhorar a politica de substituição *LRU* do SO. Esta estratégia nem sempre é a melhor para a base de dados.

Os *buffers* têm duas *flags* importantes:

pinned quando bloqueado por um processo e não pode ser usado enquanto o processo que o está a usar não o libertar.

dirty quando foi modificado desde a leitura de disco.

A estratégia usada como política de substituição designa-se por **clock sweep** e é uma aproximação (melhorada para o objetivo) da estratégia *LRU*.

3 Indexação e *Hashing*

Um índice, ou **index relation**, é uma estrutura com potencial de melhorar a *performance* do sistema no que concerne à pesquisa. No entanto, tal como acontece com uma tabela (*heap relation*), qualquer índice requer espaço próprio em disco, que neste caso particular se traduz em redundância (ao guardar cópias dos dados indexados).

Existem tipos de índice distintos, o que se traduz em estruturas de dados diferentes com algoritmos desenhados especificamente para auxiliar, com eficiência, a resposta a *queries* num dado formato.

A decisão de construir, ou não, um índice sobre um dado atributo (ou conjunto de atributos como se verá adiante) não tem, no caso geral, uma resposta imediata e *standard*. Em geral, é necessária uma análise cuidada de vários fatores diretamente ligados à base de dados em questão, bem como da frequência do tipo de perguntas que possam eventualmente fazer uso desse índice.

Observa-se que, de forma semelhante ao Oracle, é possível, em PostgreSQL, construír índices sobre tabelas e vistas materializadas.

Em PostgreSQL a sintaxe simplificada para a criação de um índice é:

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ]
    ON table_name [ USING method ]
```

Existem naturalmente mais opções e parametrizações para a criação de um índice. A descrição completa é remetida para a documentação do PostgreSQL [2].

3.1 Tipos de Índice

O PostgreSQL permite a construção de cinco tipos de índice distintos. As escolhas são **b-tree**, **hash**, **gist**, **spgist** e **gin**. Em caso de omissão, o PostgreSQL constrói um índice *b-tree*.

Em seguida descreve-se sumariamente os tipos de índice disponibilizados pelo PostgreSQL, sem entrar em demasiado detalhe sobre o funcionamento interno de cada um. A intenção é dar uma perspetiva do que está disponível, quais o fortes de cada um, e quando será melhor aplicar um em vez do outro.

3.1.1 B-tree

```
CREATE INDEX index_name ON table_name USING btree;
```

Suporte:

- *queries* de igualdade e *range queries* sobre atributos para os quais esteja definida uma relação de ordem total (inclui operadores com a mesma semântica **BETWEEN** e **IN**)
- **IS NULL** e **IS NOT NULL**
- *pattern-matching* para padrões constantes e fixados no início

Por *default* as entradas do índice estão ordenadas por ordem crescente, mas esta opção pode ser alterada para ordem decrescente. Note-se que embora o índice *B-tree* possa ser percorrido em qualquer direção, uma vez que as folhas do índice têm ligações para as folhas irmãs, para índices multi-atributo, percorrer o índice na ordem inversa pode não resultar na correspondente ordem inversa das entradas.

A forma correta de implementar este comportamento passa pela parametrização usando **ASC** ou **DESC** aquando da criação do índice:

```
CREATE INDEX index_name ON table_name USING btree [ASC | DESC]
```

Internamente, a implementação da *B-tree* usada pelo PostgreSQL, é baseada na árvore de alta concurrência proposta por Lehman e Yao.

3.1.2 Hash

```
CREATE INDEX index_name ON table_name USING hash;
```

Suporte:

- Condições de igualdade sobre o atributo indexado.

O PostgreSQL implementa *hash* dinâmico para a construção deste tipo de índice.

3.1.3 Índices Generalizados

Generalized Search Tree (GIST)

As motivações para este tipo de índice prendem-se com:

- a eficiência observada nos índices baseados em árvores equilibradas (*B-Tree*, *R-Tree*)
- a falta de suporte para operadores de comparação para além dos usuais (<,=,>,...)
- tipos de dados não *built-in*

Mais concretamente, o índice *GiST* armazena entradas do tipo (*key*, *value*). No entanto a chave de um índice *GiST* pertence a uma classe definida pelo utilizador (ao contrário dos índices *B-tree*, cuja chave é do tipo inteiro). Ao implementar um conjunto de métodos da interface deste índice, o utilizador consegue estender efetivamente o tipo de queries para além daqueles envolvendo os operadores usuais, de forma a suportar qualquer tipo de predicado.

Space-Paritioned GiST (SP-GiST)

Baseado nas mesmas premissas que os índices *GiST*, mas colmatando a falha de os índices GiST se basearem apenas em árvores equilibradas, deixando de fora vários tipos de estruturas interessantes em aplicações, como *K-D-Trees*, *Quadtrees* (CAD, GIS, multimédia), *tries* ou *suffix trees*. A característica comum entre todas estas aplicações é que decompõem o espaço (domínio) em partições disjuntas, como árvores não equilibradas.

Generalized Inverted Index (GIN)

O índice GIN é mais um tipo de índice extensível a tipos de dados definidos por utilizador bem como a tipos de *queries* suportados. Muito à semelhança do índice *GiST*, implementa parcialmente uma estrutura com base na *B-Tree* (fazendo tambem uso do algoritmo de gestão de alta-concurrência de Lehman e Yao).

Este tipo de índice é no entanto desenhado para casos em que os *items* a serem indexados são compostos, e as *queries* a serem tratadas necessitam de procurar por valores (chaves) contidas nos items.

Observa-se que existe uma terminologia própria neste tipo de índice, em que um *item* designa sempre um valor composto a ser indexado e a chave refere-se ao valor a ser procurado no item.

Um exemplo (e aplicação) comum para este índice, é quando os *items* são documentos e as *queries* indicam pesquisas por documentos que contenham determinadas palavras (que neste caso são as chaves).

3.1.4 Comparação com o Oracle

De seguida segue-se uma tabela de comparação entre os índices presentes no sistema PostgreSQL e o sistema Oracle:

O Oracle implementa os índices generalizados do PostgreSQL através de índices *Application Domain*.

	B-tree	Hash	Bitmap	Índices Generalizados
Oracle	X	X	X	X
PostgreSQL	X	X		X

Tabela 1: Comparação entre os diferentes sistemas de gestão de base de dados estudados.

3.2 Cluster e Index-file Organization

No PostgreSQL, ao contrário do Oracle, não é possível usar uma *index-file organization*, i.e., guardar fisicamente uma relação num índice. Existe no entanto uma técnica que permite aproximar esta situação.

O comando `CLUSTER` pode ser usado para reordenar fisicamente uma relação, de acordo com um índice (por exemplo de acordo com a ordem mantida numa *B-tree* que indexa a relação). Esta operação não mantém a tabela organizada desta forma, mas dependendo do *rate* de atualização dos dados da tabela, poderá ser possível mantê-la organizada, ou muito próxima disso, de acordo com o índice, durante períodos de tempo mais, ou menos, prolongados.

Em comparação com o Oracle, esta estratégia requer correr o comando `CLUSTER` de tempo a tempo para reorganizar a tabela, mas por outro lado evita o *overhead* de manter constantemente a tabela naquela forma, o que pode não trazer vantagens significativas, quando comparada com o re-arranjo de tempo a tempo da tabela. Note-se que na prática não o deverá mesmo fazer na generalidade dos casos, uma vez que o Oracle usa por *default* uma **heap-file** organization.

Observe-se que no cenário de *update* de tuplos, o sistema tenta sempre colocar a cópia atualizada do tuplo na mesma página da versão antiga. Se cada página for forçada a manter mais espaço livre para este efeito, consequentemente as páginas deverão manter-se organizadas durante mais tempo. O parâmetro de tabela `FILLFACTOR` que determina a percentagem mínima de espaço livre reservado para inserções numa página, pode ser afinado para atenuar o efeito de updates na ordenação física do ficheiro.

É de referir que o comando `CLUSTER` em PostgreSQL não deve ser confundido com o comando `CREATE CLUSTER` em Oracle que serve efetivamente para criar um *multitable cluster*.

3.3 Subtipos de Índices

É importante mencionar também o suporte do PostgreSQL para subtipos disponíveis para as categorias de índices anteriormente descritas:

Single Column Criado com base numa única coluna da tabela e suportado por todos os tipos de índice.

Multiple Column Definido sobre mais de uma coluna da relação, suportado pelos índices *b-tree*, *gist* e *gin*. Máximo de 32 atributos suportados.

Unique Para efeitos de *performance* e integridade, não permite valores duplicados, forma recomendada de implementar uma *constraint* do tipo chave. Apenas o índice *B-tree* suporta este tipo.

Partial Construído sobre um subconjunto de uma tabela, definido através de um predicado declarado na cláusula `WHERE`.

Implicit Criados automaticamente pelo sistema, e.g. índices criados com a declaração de chaves primárias ou *constraints* do tipo *unique*.

3.4 Uso combinado de Índices – Bitmaps Temporários

O PostgreSQL tem a capacidade de combinar mais de um índice (incluindo uso repetido do mesmo índice) formando condições de **AND** e **OR** sobre os resultados dos mesmos. Internamente, o sistema forma *bitmaps* temporários em memória, de acordo com o resultado da avaliação das entradas do índices, que depois são conjugados usando as condições *AND* e *OR* originais. De acordo com as características de um *bitmap*, as entradas da tabela são percorridas pela ordem física, pelo que qualquer ordem que os índices implementassem é perdida no processo. Se necessário, o resultado é ordenado antes de ser devolvido.

3.5 Covering Indexes - Index-Only Scans

Esta *feature*, também existente no Oracle e foi implementada a partir do PostgreSQL 9.2. Permite que certos tipos de *queries* sejam respondidas apenas acedendo ao índice sem necessidade de consultar a informação na tabela.

Naturalmente, é necessário que os atributos indexados permitam responder à *query*. Uma das dificuldade observadas na implementação desta funcionalidade, e razão pela qual demorou tanto tempo a implementar uma funcionalidade assente num principio tão simples, prende-se com o facto dos índices não armazenarem qualquer informação referente à visibilidade dos tuplos.

Os designados índices de cobertura, são índices criados precisamente para o propósito de **index-only scans**. Tipicamente incluem uma ou mais colunas (que em cirscuntâncias alheias não trariam vantagem incluir no índice), em particular colunas que se sabe fazererm parte de queries frequentes e de custo elevado.

3.6 Estruturas Temporariamente Inconsistentes

O PostgreSQL gere restrições de integridade com o uso de índices do tipo *unique*. Devido ao uso do MVCC implementado pelo PostgreSQL (ver secção 5), existem situações de conflito onde um índice pode necessitar de guardar fisicamente múltiplas versões de uma mesma entrada.

O conceito está associado ao problema das transações, mas em termos simples, o sistema pode permitir que estas estruturas verifiquem temporariamente estados inconsistentes, até que as transações envolvidas/responsáveis por este efeito terminem (sejam *commited*).

4 Processamento e Otimização de Perguntas

O processo de processamento e otimização de uma pergunta por parte do PostgreSQL encontra-se dividido em 4 etapas:

Parser A pergunta é verificada sintaticamente e é criada uma *query tree*;

Sistema de Reescrita Utilizando a *query tree* criada na etapa anterior, é verificada a existência de regras no sitema de regras que se possam aplicar. O sistema de regras efetua as transformações na *query tree*;

Planeador O planeador cria um plano para ser executado baseado na *query tree* reescrita;

Executor O executor é responsável por percorrer recursivamente o plano encontrado na etapa anterior e obter as entradas nele representadas.

De seguida, vamos descrever estas etapas em maior detalhe.

4.1 Parser

Neste primeiro passo o PostgreSQL valida a pergunta a efetuar, retornando um erro caso não seja sintaticamente correta. Se for válida, é construída uma *parse tree* com base num conjunto de regras fixas sobre a estrutura sintática do SQL.

Depois de construída a *parse tree* inicia-se um processo de transformação, fazendo uma intrepretação semântica da pergunta. Durante esta transformação o sistema fica a saber que tabelas, funções e operadores são referenciados pela pergunta e dela resulta a *query tree* que irá ser passada ao sistema de reescrita.

4.2 Sistema de Reescrita

O sistema de regras de reescrita é um sistema que transforma uma *query tree* utilizando um conjunto de regras definido nos catálogos do PostgreSQL. É um sistema poderoso que permite a otimização de uma pergunta antes de esta ser processada pelo planeador. Uma das utilizações deste sistema é a implementação do sistema de vistas e vistas materializadas.

Na maioria dos casos uma regra de reescrita pode ser implementada com um *trigger*.

4.2.1 Vistas

Quando uma vista é criada, o sistema cria uma relação e uma regra sobre uma instrução de seleção em relação ao nome da vista. Esta regra faz com que uma pergunta sobre a vista seja transformada numa instrução de seleção sobre as relações referenciadas.

Podemos ainda criar regras para a atualização, criação e destruição de forma a podermos representar as operações de modificação de entradas sobre uma vista.

4.2.2 Vistas Materializadas

Uma vista materializada é similar a uma vista normal. No entanto, uma vista materializada guarda os resultados da seleção numa relação intermédia. As operações de modificação de entradas não são utilizadas, utilizando antes uma instrução de refrescamento que executa a operação de seleção e atualiza a relação.

4.3 Planeador

O planeador é responsável por processar a *query tree* reescrita e encontrar um plano de execução adequado à pergunta. O planeador tem um processo de otimização com a pretensão de encontrar o plano de custo menor para uma dada *query tree*. A escolha de um plano é feita com base numa análise feita sobre todos os caminhos possíveis para uma dada *query tree*. Um caminho é uma representação de uma *query tree* otimizada para análise pelo planeador.

O planeador começa por examinar as relações base de uma pergunta e analisa o custo dos vários métodos de acesso. Esses métodos podem ser:

***Scan* sequencial** Uma relação pode ser sempre percorrida por um *scan* sequencial, logo é sempre gerado um plano para este método;

***Scan* por índice** Se existe um índice definido sobre um atributo relevante na relação - restringido ou ordenado por um operador apropriado ao tipo de índice - é gerado um plano para um *scan* por índice;

Scan* por *bitmap heap No caso de existir um índice nas condições anteriores também é gerado um plano para o *scan* por *bitmap heap* sobre esse índice. Este tipo de *scan* é uma otimização do *scan* por índice que permite uma maior rapidez nos acessos ao disco ao reordenar os acessos aos blocos de forma a minimizar o número de acessos.

De seguida inicia o planeamento da junção de relações, caso seja necessário.

Normalmente, o planeador otimiza a *query tree* a processar, puxando sub-perguntas simples para a *query tree* principal e simplificando as cláusulas de junção de forma a ter só uma lista de relações a juntar.

No entanto, alguns tipos de junção não são simplificados. Desta forma o nosso planeamento vai incidir sobre uma lista de relações a ser juntadas em qualquer ordem, sendo que uma entrada individual pode ser uma sub-lista que tem que ser juntada internamente antes de poder ser juntada ao resto da relação.

O processamento destes problemas são efectuados recursivamente numa estratégia *bottom up*. Para procurar um tipo de junção a aplicar a estas entradas é identificada uma ordem no tipo de junção a explorar, mas não é excluída nenhuma, nem é indicada qual a ordem das relações. Os tipos de junção que o sistema implementa são:

Nested Loop A relação da direita é percorrida por cada entrada encontrada na relação da esquerda. É possível obter bons resultados se a relação da direita for percorrida através de um índice;

Hash Join A relação da direita é percorrida e carregada para uma hash table usando os atributos de junção como chaves. De seguida percorre-se a relação esquerda e utiliza-se os atributos correspondentes à junção como chave para encontrar as entradas correspondentes da relação da direita;

Merge Join Cada relação é ordenada se necessário antes da junção. As duas relações são percorridas em paralelo e as entradas que satisfaçam a junção são combinadas. A ordenação pode ser feita explicitamente ou percorrendo a relação ordenadamente utilizando um índice sobre a chave de junção.

As estratégias de junção são binárias, pelo que se a pergunta envolver mais do que duas relações é preciso desdobrar em várias operações de junção, cada uma envolvendo apenas duas relações.

A pesquisa pelos tipos de junção é quase exaustiva. Na prática, resultados que são declaradamente inadequados não são envolvidos na geração do plano. No caso de o número de junções ser elevado (valor controlado pela variável `geqo_threshold`) a pesquisa não é exaustiva. Ao invés é aplicado um algoritmo genético a um conjunto base

de junções e percorrem-se algumas gerações, mantendo sempre os melhores indivíduos de geração para geração. Desta forma pretende-se evitar pesquisas demasiado longas na análise do custo, com perda de garantia de que se encontra uma junção ótima.

O plano final consiste de uma árvore em que os nós representam métodos de acesso, métodos de junção e nós auxiliares como de ordenação ou agregação. A maioria dos nós permite a execução de operações de seleção e projeção.

4.4 Execução

O plano gerado pelo planeador é agora executado recursivamente através de um mecanismo *demmand-pull pipeline* para obter as entradas especificadas. Cada vez que um nó do plano é chamado, ele deve devolver uma entrada ou reportar que já não tem mais entradas a devolver.

O executor é responsável por processar os quatro tipos básicos de queries SQL. Para `SELECT` o executor terá que retornar as entradas resultantes da execução do plano. Para `INSERT` cada entrada é inserida na tabela especificada utilizando um nó ModifyTable. Para `UPDATE` é acrescentado o ID da entrada original e utiliza novamente o ModifyTable para inserir uma nova entrada, marcando como apagada a anterior. Para `DELETE` retorna-se apenas o ID das entradas a apagar e utiliza-se o ModifyTable para percorrer os IDs e marcá-los como apagados.

4.5 Utilização de Estatísticas

O PostgreSQL guarda o número de tuplos em cada tabela e índice e o número de blocos utilizados por cada tabela e índice. Por razões de eficiência não são actualizados em tempo real; necessitam de comandos específicos para actualizar. Uma operação `VACUUM` ou `ANALYZE` que não faça *scan* à tabela inteira vai atualizar a contagem dos tuplos baseada na secção da tabela onde foi efectuado o *scan*. O planeador irá então escalar os valores de forma a ir ao encontro do tamanho físico da tabela, obtendo uma aproximação.

O PostgreSQL mantém também uma tabela, *pg_statistics*, que guarda a informação utilizada aquando da utilização de comandos de seleção em que só é retornado um número limitado de tuplos. A tabela *pg_statistics* mantém informação como os valores mais comuns num campo de uma tabela, a frequência desses valores ou os limites do histograma de valores.

É possível alterar o número máximo de entradas para alguns destes campos, de maior interesse os campos *most_common_values* e *histogram_bounds*, utilizando o comando `ALTER TABLE SET STATISTICS`. Ao aumentarmos esse valor (por omissão 100) vamos obter estimativas mais precisas, em particular para campos com uma distribuição de dados irregular, à custa de uma maior ocupação da tabela *pg_statistics* e de um processamento maior aquando do cálculo das estatísticas. Inversamente podemos reduzir este valor para campos que tenham uma distribuição simples.

4.6 Exemplos Práticos

São agora apresentados alguns exemplos que servem de demonstração de alguns dos conceitos apresentados anteriormente. Alguns destes exemplos procuram clarificar caraterísticas específicas do sistema, enquanto outros demonstram uma generalidade de conceitos teóricos.

4.6.1 Redução de *Outer Joins*

Este exemplo apresenta uma *query* que aplica um *outer join* às relações *country* e *ismember*:

```
SELECT name
FROM country LEFT JOIN ismember ON(Country.code = ismember.country);
```

Mas caso haja uma restrição do lado direito isto impede que existam linhas vazias do lado direito, e assim é possível converter de um *outter join* para um simples *join*:

```
SELECT name
FROM country LEFT JOIN ismember ON(Country.code = ismember.country)
WHERE ismember.country > 'P';
```

Enquanto que no primeiro caso é aplicado um *hash right join*, na segunda *query* é aplicado um simples *hash join*.

4.6.2 Utilização de Índices

É agora apresentada a metodologia das *hints* para índices do Oracle:

```
SELECT population
FROM city
WHERE name='Lisbon';
```

Esta *query* começa por usar um índice da chave primária. Caso se desative o uso de índices *b-tree*: `set enable_indexscan = off;` é ainda usado o *heap bitmap*, sendo que para se forçar a usar o *sequential scan* é necessário desativar o uso de *bitmaps* com: `set enable_bitmapscan = off;`, mais exemplos de variáveis são: `enable_hashjoin, enable_mergejoin ou enable_nestloop`.

4.6.3 Inferência do algoritmo Merge Join

É importante referir a escolha de uma estratégia de *merge join* para situações que envolvam uma grande quantidade de registos com elevada seletividade, sendo vantajoso ordenar as relações para junção.

Um exemplo um pouco forçado, passa pela junção de duas tabelas com as distâncias entre cada cidade presentes na base de dados e aplicando uma restrição de igualdade:

```
SELECT *
FROM distance AS d1 INNER JOIN distance AS d2
ON d1.city1 = d2.city1 and d1.city2 = d2.city2
ORDER BY d1.distance;
```

Em que através do comando explain podemos extrair a seguinte informação e confirmar a aplicação do algoritmo *merge join* sem qualquer desativação:

```
Sort  (cost=7093848.74..7121525.21 rows=11070590 width=44)
  Sort Key: d1.distance
  ->  Merge Join  (cost=4145856.90..4436363.05 rows=11070590 width=44)
      Merge Cond: (((d1.city1)::text = (d2.city1)::text) AND ...
      ->  Sort  (cost=2072928.45..2097121.16 rows=9677082 width=22)
          Sort Key: d1.city1, d1.city2
          ->  Seq Scan on distance d1  (cost=0.00..156261.82...
      ->  Materialize  (cost=2072928.45..2121313.86 rows=9677082...
          ->  Sort  (cost=2072928.45..2097121.16 rows=9677082...
                Sort Key: d2.city1, d2.city2
                ->  Seq Scan on distance d2  (cost=0.00..156261.82...
```

De notar que é necessário um grande volume de registos, caso contrário o PostgreSQL aplica um *hash join* para ordenar posteriormente.

4.7 Notas

- O plano gerado pelo planeador aparenta utilizar uma variante da álgebra relacional representada em árvore. As operações de seleção, projeção, agregação e os vários produtos estão representados nos nós de uma forma otimizada para processamento no executor;
- O PostgreSQL tem o comando `EXPLAIN` que permite a visualização do plano de execução encontrado para um dado query.

5 Gestão de Transações e Controlo de Concorrência

Transações procuram oferecer propriedades ACID [1] (atomicidade, consistência, isolamento e durabilidade) para um conjunto de operações sobre uma base de dados. Nesta secção, vamos estudar em pormenor o mecanismo de transações implementado pelo PostgreSQL e que funcionalidades oferece às aplicações/utilizadores de modo a que estes possam adaptar as suas necessidades sobre transações concorrentes.

O PostgreSQL define qualquer operação de leitura ou escrita isolada como uma transação (mecanismo *auto-commit*) mas não tem suporte para transações *nested* nem para transações de longa duração.

5.1 Isolamento em Transações

O PostgreSQL apresenta um mecanismo de **multiversion concurrency control** (MVCC) baseado em **snapshot isolation** para garantir consistência dos dados entre transações concorrentes. Isto garante que cada transação veja uma *snapshot* consistente da base de dados obtida no seu início. Apesar de também serem usados *locks* de leitura e de escrita, o principal objetivo do uso de multiversão passa por operações de leitura e escrita não se bloquearem entre si, nem mesmo em níveis de isolamento restritos devido ao uso de multiversão.

O PostgreSQL oferece aos utilizadores a possibilidade de escolherem um dos 4 níveis de isolamento do SQL *standard* para as suas transações (apesar de na prática não oferecer o nível *Read Uncommitted*):

Serializable

Nível mais restrito de isolamento. Garante que o estado final da base de dados ao fim da execução de transações concorrentes é equivalente ao estado final deixado por uma das serializações possíveis de execução entre as transações, ou seja simula serialismo. Cria uma *snapshot* antes da execução da transação. Este é o nível que oferece menos concorrência, portanto e à semelhança do Oracle, o PostgreSQL afasta-se da definição formal e em vez de bloquear transações, ao fim destas terminarem, verifica através de monitores se o escalonamento real de transações concorrentes equivale a um escalonamento serializado, se não for o caso é efetuado *rollback* da transação pelo sistema e é apresentado o erro: `ERROR: could not serialize access due to read/write dependencies among transactions`. Ou seja não deixa que a base de dados acabe num estado inconsistente, fica a cargo das aplicações/utilizadores refazerem a transação caso não seja executado um escalonamento serializado.

Repeatable Read

Este nível de isolamento, à semelhança do nível *Serializable*, vê apenas dados *commited* por outras transações (*snapshot* da base de dados antes da transação começar). Semelhante ao nível *Serializable*, mas sem monitores para verificação de escalonamentos serializados. Apesar disso, não permite alterações concorrentes aos mesmos dados (leituras concorrentes são sempre executadas). Caso uma transação concorrente t2 faça *commit* de uma alteração de dados que uma transação t1 ainda utilize, é efetuado *rollback* da transação t1 pelo sistema e lançado o erro: `ERROR: could not serialize access due to concurrent update` e é a a aplicação/utilizador que terá de lançar novamente a transação.

Read Committed

Nível de isolamento usado pelo PostgreSQL por defeito. Permite que uma transação possa ver dados *committed* por outras transações concorrentes ao longo da sua execução. Transações não *committed* não alteram a visão de transações concorrentes. Ao contrário do nível *Repeatable Read*, cada transação poderá ler valores diferentes do mesmo *item* em diferentes períodos da sua execução, isto porque em vez de trabalhar sobre uma *snapshot* da base de dados antes de a transação começar, é obtida uma *snapshot* antes de uma operação da transação ser executada (granularidade mais fina).

Read Uncommitted

Não garante qualquer tipo de isolamento entre transações concorrentes. Internamente o PostgreSQL representa este nível como o nível *Read Committed*, para que que seja admissível importar os 4 níveis de isolamento *standard* para uma arquitetura multiversão. Como o SQL *standard* diz sobre os níveis de isolamento apenas o que não pode acontecer e não o que vai acontecer, isto vai de acordo com a definição. O utilizador apenas deverá ser avisado de que com este nível poderá ter mais restrição e portanto menos concorrência do que estaria à espera.

5.1.1 Implementação MVCC

Vamos agora detalhar mais um pouco sobre a implementação do modelo MVCC [3] do PostgreSQL. A cada transação antes da sua execução, é atribuído um identificador único gerado de forma crescente. Também antes da execução, são obtidos todos

os identificadores das transações que se encontram a ser processadas naquela altura. O modelo MVCC associa a cada transação uma *snapshot* da base de dados. Uma *snapshot* não passa de um conjunto de registos visíveis à transação. Cada registo de uma relação no sistema PostgreSQL tem dois identificadores relativos às transações de criação (*xmin*) e de expiração (*xmax*) do registo.

Quando a transação cria um novo registo, o *xmin* deste é o identificador da transação e o *xmax* fica vazio.

Quando a transação elimina um registo, *xmax* passa ao identificador da transação mas o *xmin* mantém-se inalterado e o registo mantém-se no sistema.

Caso a transação altera um registo, a versão antiga é mantida mas o seu *xmax* passa para o identificador da transação e é criado um novo registo com o mesmo conteúdo.

Para que a transação veja o conjunto de registos correto, um registo é visível à transação se e só se:

- O identificador da transação de criação do registo for de uma transação já *commited* e menor que o identificador da transação corrente.
- O registo não possuir um identificador da transação de expiração, ou se este existir, for menor que o identificador da transação corrente.

Apesar das vantagens de não bloqueio entre leitores e escritores, esta metodologia revela problemas para a remoção de versões de registos não necessários. Isto porque diferentes transações têm diferentes visões sobre as relações, não sendo linear descobrir o momento em que os registos deixam de ser utilizados e podem assim ser removidos. O PostgreSQL utiliza um mecanismo de **VACUUM** para *garbage collection* como explicado na secção 2.

5.2 Mecanismos de *Locking*

No PostgreSQL existem também mecanismos de *locks* que são usados internamente pelo sistema para diversas situações, por exemplo garantir que uma relação não é eliminada num determinado período de tempo (inserção de um registo). Este mecanismo pode também ser usado explicitamente pelas aplicações/utilizadores para definir uma granularidade personalizada para as tarefas que pretendem executar.

Quando um *lock* é adquirido implicitamente pela transação, este só é libertado quando a transação termina, ou seja é implementado o protocolo *2 phase locking strict*. O número de *locks* máximo por transação pode ser especificado pelo utilizador.

5.2.1 Níveis de Granularidade

Podem ser especificados vários níveis de *locking* e por consequência vários tipos de *locks*, tanto sobre relações como registos:

Relação

São vários os tipos de *locks* que estão associados a relações completas, apesar de alguns dos tipos de *lock* terem a palavra *row*, isso só acontece por razões históricas:

`ACCESS SHARE; ROW SHARE; ROW EXCLUSIVE; SHARE UPDATE EXCLUSIVE; SHARE; SHARE ROW EXCLUSIVE; EXCLUSIVE; ACCESS EXCLUSIVE`

Os nomes são bastante indicativos do objetivo de cada um dos modos, para mais informação consultar [2].

Como se pode ver pela figura 1, certos *locks* entram em conflito entre si, bloqueando as operações que necessitam de adquirir o acesso completo a uma relação, mas oferece uma grande flexibilidade às aplicações com este vasto conjunto de modos para evitar bloqueios desnecessários.

Requested Lock Mode	Current Lock Mode							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

Figura 1: Modos de *Table-Locking* conflituosos (retirado de [2])

Registo

Ao nível de registos, estão disponíveis os tradicionais *share locks* e *exclusive locks*.

Enquanto que quando uma transação adquire um *share lock* sobre um registo outras transações poderão adquirir o mesmo *share lock* em simultâneo, quando é adquirido um *exclusive lock*, por exemplo implicitamente pelo sistema aquando de uma remoção ou alteração de um registo, mais nenhuma transação consegue aceder ao registo através de um *share lock* ou *exclusive lock*.

5.2.2 Deteção de *Deadlocks*

Com um mecanismo de *locks* nasce naturalmente a possibilidade de *deadlocks* dentro do sistema, mesmo sem o uso de *locking* explícito. O PostgreSQL deteta automaticamente situações de *deadlock* e é garantido progresso: uma das transações que estão em situação de *deadlock* é abortada automaticamente (mas não é especificada qual a transação a ser abortada). Caso não seja detetado qualquer tipo de *deadlock*, uma transação poderá ficar indefinidamente bloqueada por exemplo, à procura de uma relação ou de um registo.

5.3 Mecanismos de Recuperação

5.3.1 Savepoints

Apesar do comando *rollback* anular as operações efetuadas por uma transação até ao momento em que esta foi criada, é conveniente oferecer ao utilizador a possibilidade de explicitamente anular parte da transação sem que esta seja abortada. Isto oferece uma maior flexibilidade às aplicações para executarem as suas tarefas. O PostgreSQL oferece mecanismos para criação de *savepoints* dentro de transações e anulação de operações até à declaração de um *savepoint*. Exemplos de utilização serão apresentados mais à frente neste documento.

5.3.2 Write-Ahead Logging

É utilizado pelo PostgreSQL um mecanismo de *logs* para armazenar as alterações sobre relações e índices em disco antes destas serem executadas, isto para garantia de tolerância a falhas. Escritas em disco são reduzidas aumentando assim a eficiência, pois basta que o *log* de alterações seja despejado para disco para garantir que por exemplo, uma transação esteja num estado *commited*. Mas o principal objetivo deste mecanismo é permitir uma recuperação do sistema: mantendo-se um *backup* da base de dados inalterado e aplicando os diferentes *logs* armazenados a esta (base do mecanismo *Point-in-Time Recovery* (PITR)). O PostgreSQL mantém um *write ahead log* (WAL) na diretoria `pg_xlog/` para todas as alterações.

5.4 Diferimento de Verificação de Consistência em Transações

Durante a execução de uma transação no sistema PostgreSQL, a verificação de consistência é realizada ao fim de cada operação constituinte da transação por defeito. Mas em transações ACID tal não é necessário, bastando que uma transação faça esta verificação antes de ser executada e deixar, quando termina, um estado consistente na base de dados, em vez de cada operação individual da transação. Em conjunto com a propriedade de isolamento, isto é suficiente para garantir a propriedade de consistência. Mesmo a verificação de consistência imediata é também um obstáculo para transações que necessitem de deixar a base de dados num estado inconsistente durante a sua execução para que seja possível terminarem com sucesso.

Assim o PostgreSQL apresenta um mecanismo para adiamento da verificação de restrições de consistência para o final da transação. Serão apresentados mais à frente os comandos necessários para utilização desta funcionalidade.

5.5 Utilização Prática do Sistema

São agora apresentados os comandos oferecidos pelo PostgreSQL (v9.3) para a utilização e parametrização dos conceitos acima descritos.

5.5.1 Transações

No sistema PostgreSQL, uma transação é delimitada da seguinte forma:

```
[ BEGIN; ]
  -- Transaction body
[ COMMIT; | ROLLBACK; ]
```

5.5.2 Parametrização das Transações

Parametrização sobre: diferentes níveis de isolamento, transações só de leitura ou de escrita ou verificação de restrições de integridade verificadas apenas no fim da transação é especificada usando a seguinte sintaxe:

```
BEGIN TRANSACTION [ transaction_mode [, ...] ];
  -- Transaction body;

where transaction_mode is one of:
```

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

    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

5.5.3 *Locking* Explícito

Locking explícito de relações completas e registos:

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

where lockmode is one of:

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

-----------------------------------------------------------------------

FOR [ UPDATE | SHARE] [ OF table_name [, ...] ] -- row level locking
```

5.5.4 *Savepoints*

Já as operações sobre *savepoints* são as seguintes:

```
SAVEPOINT               savepoint_name; -- create a sp
ROLLBACK TO SAVEPOINT   savepoint_name; -- rollback to a sp
RELEASE SAVEPOINT       savepoint_name; -- discard a sp
```

5.5.5 Verificação de Consistência dentro de Transações

Para que restrições sobre dados sejam verificadas apenas no fim de uma transação, depois dessas mesmas restrições serem declaradas obrigatoriamente como *deferrable* na sua criação, é necessário proceder ao seguinte comando para que esse adiamento de verificação se torne efetivo na transação (a primeira parametrização descrita tem também uma opção para este ponto):

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

5.6 Exemplos Característicos

São agora apresentados alguns exemplos que procuram demonstrar alguns dos mecanismos estudados nesta secção. Os exemplos trabalham sobre uma relação criada com o seguinte comando:

```
CREATE TABLE t(
    code INTEGER CONSTRAINT code_pkey PRIMARY KEY DEFERRABLE,
    name VARCHAR(5)
);
```

No início de cada transação de exemplo a relação T é constituída pelos seguintes registos:

```
  code | name
-------+-------
     1 |  'one'
     2 |  'two'
     3 |'three'
     4 | 'four'
```

5.6.1 Controlo de Situações de *Deadlock*

É apresentado um exemplo de escalonamento entre duas transações que provoca uma situação de bloqueio mútuo e como o sistema responde a isso:

```
              T1              |              T2
------------------------------+------------------------------
            BEGIN;            |
                              |             BEGIN;
DELETE FROM T WHERE CODE = 3;|
                              | DELETE FROM T WHERE CODE = 4;
DELETE FROM T WHERE CODE = 4;|
                              | DELETE FROM T WHERE CODE = 3;

*T2 aborts in the last command and presents the following error
                while T1 continues execution*
        (who aborts is not always the older transaction)

ERROR:  deadlock detected
DETAIL:  Process 7856 waits for ShareLock on transaction 793;
blocked by process 7615.
Process 7615 waits for ShareLock on transaction 794;
blocked by process 7856.
```

5.6.2 Bloqueios Devido a *Locking* Explícito

Situação em que uma transação fica bloqueada devido a uma alteração de um registo partilhado de forma concorrente:

```
              T1              |              T2
------------------------------+------------------------------
            BEGIN;            |
                              |             BEGIN;
SELECT * FROM T               |
WHERE CODE = 1 FOR UPDATE;    |
                              | SELECT * FROM T
                              |  WHERE CODE = 1 FOR UPDATE;

                *T2 hangs indefinitely*
     Note that this happens even in serealizable mode
                with concurrent updates
```

5.6.3 Isolamento *repeatable read*

Aqui é apresentada a propriedade do isolamento *repeatable read* que diz que todas as leituras sobre um determinado *item* numa transação não são alteradas por outras transações concorrentes, ou seja é sempre lido o mesmo valor.

```
             T1             |              T2
----------------------------+----------------------------
BEGIN TRANSACTION ISOLATION  |
  LEVEL REPEATABLE READ;     |
                             |BEGIN TRANSACTION ISOLATION
                             |   LEVEL REPEATABLE READ;
SELECT * FROM T              |
    WHERE CODE = 1;          |
                             | UPDATE T SET NAME = 'one1'
                             |      WHERE CODE = 1;
                             | COMMIT;
SELECT * FROM T              |
    WHERE CODE = 1;          |
COMMIT                       |
```

```
     *T1 sees code = 1 and name = 'one' in both selects*
```

5.6.4 Uso de *Savepoints* e Restrições *Deferred*

No seguinte exemplo são apresentados os mecanismos de *savepoints* e adiamento de verificação de restrições para o fim das transações:

```
                T1
--------------------------------------
BEGIN;
SET CONSTRAINTS CODE_PKEY DEFERRED;
SAVEPOINT SP;
UPDATE T SET CODE = 2 WHERE CODE = 1;
ROLLBACK TO SAVEPOINT SP;
RELEASE SAVEPOINT SP;
COMMIT;

*T1 commits sucessfully and no registers are updated*
```

Comparação com o Oracle (11g)

Tanto o mecanismo de implementação de transações como as funcionalidades oferecidas pelo PostgreSQL relativamente a transações, estão de forma muito semelhante presentes no sistema Oracle 11g estudado durante a disciplina. Mesmo a própria sintaxe é muito parecida. As diferença mais notórias passam por:

- Oferta do nível de isolamento *Repeatable Read* pelo PostgreSQL e não pelo Oracle que em oposição oferece um nível de *Read Only*.

- Apesar de não ser suportado o nível de isolamento *Read Uncommitted* pelo Oracle, este nível equivale ao *Read Commited* no PostgreSQL. Em ambos os sistemas não é possível a leitura de dados não *commited* em transações concorrentes em nenhum nível de isolamento (*dirty reads*).

- São usados mecanismos de *locking* implicitamente pelos dois sistemas em várias operações da base de dados como alteração de um registo, podem assim ocorrer bloqueios. Mas de salientar que caso 2 transações sejam executadas de forma concorrente, ambas com um nível de isolamento *Serializable*, enquanto que o Oracle assume que dentro deste nível de isolamento caso uma transação altere um registo, mais nenhuma transação concorrente irá alterar esse registo, e caso isso aconteça é lançado um erro de serialização, o sistema PostgreSQL bloqueia a última transação que quis alterar os dados (o *lock* de escrita do registo não dá acesso à transação) e só é retomada a transação quando a outra terminar, para garantir serialização.

- Enquanto que o PostgreSQL não diferencia o espaço de armazenamento para diferentes versões dos mesmo dados mas mantém um mecanismo de *garbage collection* para eliminar registos antigos, o Oracle armazena versões antigas num espaço de armazenamento limitado e as versões mais recentes dos registo incluem apontadores para esse espaço de armazenamento. Como este espaço é limitado, versões antigas são eliminadas para dar lugar a novas versões. Caso um transação necessite de uma versão já eliminada esta é abortada.

6 Suporte para Bases de Dados Distribuídas

Apesar do conceito de bases de dados estar associado naturalmente a centralização, vários sistemas de gestão de base de dados implementam funcionalidades para distribuirem os dados de forma a oferecer tolerância a falhas, alta escalabilidade ou um aumento de disponibilidade.

6.1 Replicação

Mecanismos de replicação consistem na replicação dos dados presentes na base de dados por diferentes nós distribuídos numa rede, para efeitos de disponibilidade e distribuição de carga para aumento de desempenho na transferência de dados. Estes mecanismos seguem tradicionalmente uma arquitetura *Master-Slave*. O PostgreSQL utiliza a terminologia *standy* para representar um *slave*. Técnicas de *clustering* e replicação que se mantiveram externas ao PostgreSQL, foram integradas nativamente no sistema em 2008[1].

Encontram-se atualmente implementadas no sistema duas funcionalidades distintas de suporte a replicação:

Hot Standby com Streaming Replication - Replicação binária assíncrona dos dados para um ou mais *standbys*. Estes *standbys* podem-se tornar em *hot standbys* para permitirem *read-only queries* dos utilizadores/aplicações enquanto se dá a recuperação, aumentando assim a disponibilidade do sistema.

Warm Standby com Log Shipping - Replicação da base de dados para um servidor *standby* que serve para posterior recuperação dos dados.

[1] `http://www.postgresql.org/message-id/26529.1212070375@sss.pgh.pa.us`

Os casos acima apenas referem a replicação da instância completa do PostgreSQL (na sua terminologia, o *database cluster*).

Apesar disso existem diferentes sistemas que podem fazer apenas a replicação de uma relação ou de um conjunto de relações e que oferecem uma grande conjunto de funcionalidades que o sistema nativo não dispõe. Estes podem ser consultados em: `https://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling`.

Apesar de ter certas caraterísticas implícitas, existe também uma extensão ao sistema: o **Postgres-R** que procura abranger um completo conjunto de funcionalidades distribuídas e procura ser uma versão do PostgreSQL direcionada para ambientes distribuídos.

6.2 Outras Funcionalidades

O PostgreSQL não oferecee mecanismos de ligação remota como as funcionalidade de *aliases* do Oracle, apesar disso encontra-se disponível um módulo que permite uma ligação a um sistema PostgreSQL remoto, o *dblink*, mas que não implementa o *two-phase commit*.

7 Outras Caraterísticas do PostgreSQL

XML Suporta XML como tipo de dados nativo em relações. Disponibiliza um conjunto de funções para manipular o XML e permite a exportação do conteúdo através de cláusulas de seleção;

JSON Suporta JSON como tipo de dados nativo em relações. Disponibiliza um conjunto de funções para manipular objetos JSON;

Linguagem Procedimental Suporta a linguagem procedimental PL/PGSQL através da qual permite a implementação de *stored procedures* e de *triggers*;

Object-Relational Suporta o paradigma object-relational, de onde destacamos a capacidade de herança entre tabelas;

Conectores Tem conectores oficiais para JDBC e ODBC. A comunidade contribui ainda com conectores não oficiais como, por exemplo, o conector para .NET;

Ferramentas Tem uma comunidade bastante ativa e participativa que contribui com várias ferramentas open-source, gratuitas ou comerciais. No site oficial podemos consultar um catálogo de aplicações disponíveis em `http://www.postgresql.org/download/product-categories/` sendo que existem muitas mais espalhadas por outros repositórios. Destacamos a ferramenta de administração pgAdmin, que utilizámos para efetuar os nossos testes.

Bibliografia

[1] José Júlio Alferes. Aulas teóricas da displina de Sistemas de Bases de Dados. 2014.

[2] The PostgreSQL Global Development Group. *PostgreSQL 9.3.4 Documentation*. `http://www.postgresql.org/docs/9.3/static/index.html`.

[3] Bruce Momjian. *Mvcc Unmasked*. `http://momjian.us/main/writings/pgsql/mvcc.pdf`. 2013.

[4] Abraham Silberschatz, Henry Korth e S. Sudarshan. *Database Systems Concepts*. 6ª ed. New York, NY, USA: McGraw-Hill, Inc., 2006. ISBN: 0072958863, 9780072958867.