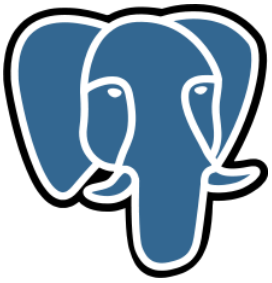
Relatório

Sistemas de Bases de Dados

Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa



***PostgreSQL* 9.3**

1 de Junho de 2014

David Fernandes Semedo Nº 41788
Dinis Bernardo Crisóstomo Belo Nº 42108
Nuno Ricardo Cordeiro Alves Nº 42224

Conteúdo

Capítulo 1

Introdução

Este relatório, desenvolvido no âmbito da disciplina Sistemas de Bases de Dados, tem por objetivo um estudo aprofundado do sistema *PostgreSQL* 9.3. A escolha deste sistema, deveu-se ao facto de ser um sistema bastante completo em termos de funcionalidades oferecidas, da documentação ser bastante detalhada no que toca a certos aspetos de implementação e por ser *open-source*. Pretende-se neste relatório abordar os vários temas estudados na disciplina, nomeadamente, os tipos de Armazenamento e estruturas de ficheiros, Indexação e Hashing, Processamento e Otimização de perguntas, Gestão de transações e controlo de concorrência, tipos de Suporte a Base de Dados Distribuídas e outras características do sistema. Para cada um dos temas, será efetuada uma comparação crítica com o sistema *Oracle*® , estudado na disciplina.

O relatório está organizado com um capítulo por tema. Inicialmente, é apresentada uma breve introdução histórica do sistema *PostgreSQL* bem como a sua aplicabilidade, na próxima secção. No capítulo 2 são estudados os tipos de armazenamento de dados e estrutura dos ficheiros da base de dados, no capítulo 3 são estudados os tipos de indexação e hashing suportados, no capítulo 4 é analisada a forma como são processadas as perguntas à base de dados bem como as estratégias de otimização das mesmas, no capítulo 5 é estudado o processamento de transações e mecanismos de suporte à concorrência, no capítulo 6 são estudados os mecanismos de suporte a bases de dados distribuídas, por fim, no capítulo 7 são abordadas outras características interessantes que o sistema *PostgreSQL* oferece.

1.1 Introdução Histórica do Sistema

O *PostgreSQL* teve origem no Ingres[77][78], um projeto académico desenvolvido por Michael Stonebraker, na universidade da Califórnia, Berkeley. O Ingres apresentava alguns problemas relacionados com o modelo de dados relacional, nomeadamente, problemas na interpretação de *tipos* de dados. Com o objetivo de resolver estes problemas, foi criado o Postgres (pós-Ingres), cuja primeira

versão foi disponibilizada no ano de 1989. O Postgres continuou em desenvolvimento até à quarta versão, na qual o projeto foi abandonado pela Universidade de Berkeley. Durante esta fase de desenvolvimento foi introduzido um sistema de regras, uma nova abordagem ao modelo clássico relacional (*Object Relational Database*), foram adicionados novos mecanismos de armazenamento de dados e foram introduzidas várias melhorias no processador de perguntas.

Em 1994, Andrew Yu e Jolly Chen, adicionaram um interpretador *SQL*, que veio substituir a anterior linguagem *QUEL*. O projeto foi renomeado para *Postgres95*.

Em 1996 foi divulgada a primeira versão do *PostgreSQL* , a versão 6.0. Dado ao facto do *PostgreSQL* ser *open-source*, formou-se um grupo de desenvolvedores que juntamente com outros voluntários têm assegurado a manutenção e desenvolvimento do sistema.

Na transição de *Postgres95* para *PostgreSQL* , versão 6.0, foram introduzidas novas funcionalidades. Introduziu-se o *MVCC* (Multiversion Concurrency Control) e acrescentaram-se novos tipos de dados (como datas, horas, etc. . .).

Na versão 7.0 foi adicionado o *Write-Ahead Log (WAL)*, esquemas *SQL* , *outer-joins*, suporte a *IPv6*, indexação por texto, estatísticas da base de dados e melhorado o suporte SSL.

Na versão 8.0 foi adicionado suporte nativo para *Microsoft Windows*, suporte a *tablespaces*, *savepoints*, *roles* e o mecanismo *Two-Phase Commit (2PC)*.

A versão estável atual é a versão 9.3 (já é possível obter a versão 9.4 *beta*). A principal linguagem de desenvolvimento é o C. O *PostgreSQL* é um sistema bastante usado em diversas empresas[1](Fujitsu, Skype, Cisco, entre outras . . .) dado ser um sistema bastante robusto, completo e livre.

[1] *PostgreSQL* Featured Users: http://www.postgresql.org/about/users/

Capítulo 2

Armazenamento e File Structure

2.1 Buffer Management

Apesar não termos encontrado um local em que dissese explicitamente como é que o *PostgreSQL* implementa o seu buffer partilhado, acreditamos que implementa, mas que este é usado em simultâneo com o buffer do sistema operativo. [54]

Em *PostgreSQL* o modulo *pg_buffercache* permite examinar o que está a acontecer no buffer partilhado. Existe ainda a função *pg_buffercache_pages* que devolve uma série de registos que estão armazenados no buffer partilhado. Por defeito o acesso público a esta função é proibido, quer por uma questão de segurança, quer de privacidade.

2.2 Armazenamento da Base de Dados

Em *PostgreSQL* os ficheiros de dados(tabelas) e de configuração são armazenados numa pasta chamada de *PGDATA* (este nome pode ser alterado), uma localização comum é */var/lib/pgsql/data*. Na mesma máquina podem existir múltiplos clusters, geridos por diferentes instâncias do servidor, cada cluster tem a sua pasta dentro da *PGDATA*

A pasta contém várias pastas e ficheiros de controlo (como mostrado na tabela seguinte). Além desses, os ficheiros *postgresql.cong*, *pg_hba.conf* e *pg_ident.conf* são igualmente gravados na pasta *PGDATA* (apesar de a partir do PostgreSQL 8.0 ser permitido grava-los noutra localização).

Item	Descrição
pg_version	O ficheiro que contém o número da versão do *PostgreSQL*
base	Pasta que contém as pastas que representam as BD's (uma por cada base de dados)
global	Pasta que contém tabelas "cluster-wide", tal como "pg_database"
pg_clog	Pasta que contém dados sobre o o estado dos commits
pg_multixact	Pasta que contém dados sobre multi-transacções (usado para locks partilhados [shared locks])
pg_notify	Pasta que contém dados sobre o estado do *LISTEN/NOTIFY*
pg_serial	Pasta que contém dados sobre as transacções serializáveis que já fizeram *commit*
pg_snapshots	Pasta que contém dados sobre *snapshots* exportáveis
pg_stat_tmp	Pasta que contém ficheiros temporários para serem usados no sistema de estatística
pg_subtrans	Pasta que contém dados sobre o estado de *sub-transactions*
pg_tblspc	Pasta que contém ligações simbólicas para as tabelas
pg_twophase	Pasta que contém ficheiros de estado para preparar transacções
pg_xlog	Pasta que contém dados sobre os ficheiros *WAL* (Write Ahead Log)
postmaster.opts	Um ficheiro onde são armazenados os argumentos da linha de comandos com que o servidor foi iniciado
postmaster.pid	Um ficheiro de lock que armazena o PID de um processo, a pasta para os dados *cluster*,o *timestamp* de início do *postmaster*, número de porta, o domínio Unix do socket (vazio no windows), o primeiro endereço de escuta válido (vazio se não está à escuta em *TCP/IP*) e ainda o ID do segmento de memória partilhada (este ficheiro não está disponível depois do *shutdown* do server.

Já o armazenamento de informações sobre tabelas, é feito utilizado o catálogo *pg_database*. Ao contrário da maioria dos catálogos de outros sistemas, o *pg_database* é partilhado por todas as bases de dados de um cluster, isto é, só existe uma cópia do *pg_database* por cluster ao invés de uma por base de dados[47].

2.3 Partições

Quando se fala em partições, no âmbito de um SGBD, refere-se à capacidade de dividir uma tabela em pequenos pedaços. Ao criar partições numa base de dados consegue-se um aumento na performance. Em *PostgreSQL* estão definidas duas formas de particionar:

range partitioning - a tabela é dividida em "intervalos" definidos pelo valor do atributo ou conjuntos de atributos, não havendo sobreposição do mesmo valor em partições diferentes;

list partitioning - a tabela é particionada tendo que ser indicado explicitamente quais as chaves que ficam em cada partição.

2.3.1 Exemplo:

Para fazer a criação de uma partição é preciso, em primeiro lugar, ter a tabela "master" ou principal, criada. Depois de a tabela principal existir é necessário criar as partições. Com o *script* seguinte:

```
CREATE TABLE film_category_eq3 (CHECK (category_id=3) )INHERITS (film_category);

CREATE TABLE film_category_df3 (CHECK (category_id!=3) )INHERITS (film_category);
```

é possível ver um exemplo da criação de duas tabelas particionadas. É importante garantir que as restrições *(CHECK (category_id=3)) e (CHECK (category_id!=3))* garantem a não sobreposição de dados entre partições.

Depois de criadas as partições é aconselhado definir um índice na chave primária e, se desejável, outros índices sobre outros atributos. (Ver em detalhe no capítulo 3).

```
CREATE INDEX film_category_eq3I ON film_category_eq3(film_id);

CREATE INDEX film_category_df3I ON film_category_df3(film_id);
```

É igualmente importante assegurar que o parâmetro *"constraint_exclusion"*, presente no ficheiro de configuração "postgresql.conf", não esteja desativado.

Por fim, é preciso criar um *trigger* e associá-lo a tabela principal. Tal pode ser feito usando o código:

```
CREATE OR REPLACE FUNCTION insert_film_category()
RETURNS TRIGGER AS $$
BEGIN
IF(NEW.category_id=3) THEN
INSERT INTO film_category_eq3 VALUES(NEW.film_id, NEW.category_id);
ELSE
INSERT INTO film_category_df3 VALUES(NEW.film_id, NEW.category_id);
END IF;
RETURN NULL;
END;
$$ LANGUAGE PLPGSQL;

CREATE TRIGGER insert_film_category
BEFORE INSERT ON film_category FOR EACH
ROW EXECUTE PROCEDURE insert_film_category();
```

2.3.2 Apagar uma Partição

Caso seja necessário remover os dados, basta apagar a partição. Para remover a ligação entre a partição e a tabela principal mas mantendo o acesso aos dados, executa-se:

```
ALTER TABLE film_category_eq3 NO INHERIT film_category;
```

2.3.3 Comparação com o standard *SQL*

Uma última nota, para lembrar que todas as funcionalidades previstas no *SQL* sobre tabelas estão também disponíveis para as tabelas particionadas. Ou seja, para fazer uma pesquisa sobre as vendas de um determinado mês, pode-se fazer a consultar diretamente sobre a partição correta, conseguindo assim melhor performance.

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

2.4 Layout de uma Página

Todas as tabelas e índices são armazenadas num *array* de páginas de tamanho fixo (normalmente de 8kb, mas pode ser definido outro tamanho enquanto se compila o servidor). Numa tabela, todas as páginas são logicamente equivalentes, então um item particular (linha) pode ser armazenado em qualquer página. [55]

Todas as linhas da tabela são definidas da mesma maneira, existe um *header* fixo (que ocupa 27 bytes na maioria das máquinas), seguido de *bitmap* (opcional), e de um ID de objeto (opcional) e, por fim, os dados do utilizador.

Os novos tuplos são armazenados no espaço livre existente numa página (o número da página pode ser encontrado num identificador (*pd_lower*).

2.5 TOAST

Como o *PostgreSQL* usa um tamanho de página fixo de 8Kb, quando é necessário armazenar valores maiores do que 8kb utiliza-se o mecanismo *TOAST (The Oversized-Attribute Storage Technique).* [48] Este mecanismo numa primeira fase, faz a compressão dos atributos como valores muito grandes e, depois, divide-os em vários ficheiros lógicos.

Apenas alguns tipos de dados suportam *TOAST*, os outros são tipos de dados em que não são produzidos valores de grandes dimensões, por exemplo, booleanos.

Com *TOAST* é possível escolher de entre quatro diferentes estratégias para o armazenamento de tabelas:

- Plain
- Extendend
- External
- Main

Estas estratégias variam em relação ao suporte de armazenamento "out of line" e compressão. A estratégia utilizada para uma dada coluna pode ser alterada através do comando:

```
Alter table set storage
```

2.6 Vacuum

Durante uma operação os tuplos podem ser apagados ou atualizados (através de um *update*) mas na realidade eles não são logo fisicamente apagados, ficando presentes até que uma operação de *VACUUM* seja realizada com sucesso.

O *VACUUM* é então o responsável por libertar o espaço ocupado por tuplos "apagados". Assim, é necessário realizar operações de *VACUUM* regularmente, especialmente nas tabelas que são frequentemente atualizadas. É ainda recomendado que em bases de dados que estejam frequentemente a ser atualizadas, o *VACUUM* seja efetuado pelo menos uma vez por dia, por exemplo, à noite. Este comando não permite ser executado dentro de uma transação[50].

2.6.1 Execução com um parâmetro

Quando o *VACUUM* é executado com um parâmetro, é apenas processada aquela tabela.

2.6.2 Analyze

De entre os parâmetros disponíveis para a execução do comando gostaríamos de destacar o *Analyze*, este parâmetro atualiza as estatísticas usadas pelo *query plan* para determinar qual a forma mais eficiente de executar uma pergunta.

2.6.3 Execução sem parâmetros

Ao executar o comando sem parâmetros, o *PostgreSQL* re-executa o comando *VACUUM* em todas as tabelas que o utilizador tem permissão, dentro da base de dados atual.

```
VACUUM (VERBOSE, ANALYZE) onek;
```

2.7 Comparação com o *Oracle®*

- Sistema de ficheiros - à semelhança do *PostgreSQL* o *Oracle®* também implementa o seu próprio *buffer management*;
- Partições - a este nível o *Oracle®* implementa List, Range e Hash Partitioning, enquanto que o *PostgreSQL* implementa Range e List Partitioning [53];
- Clustering - à semelhança do *PostgreSQL* o *Oracle®* também implementa *clustering* por índice, permitindo ainda o *cluster* de tabelas. [49];

Capítulo 3

Indexação e hashing

3.1 Indexação

Em *PostgreSQL* estão disponíveis os seguintes tipos de índices[45]:

- B-tree
- Hash
- GiST
- SP-GiST
- GIN

Cada um destes tipos é mais apropriado para um tipo de consulta. Por defeito, o comando CREATE INDEX cria um indice B-Tree, que é o que melhor se enquadra na maioria das situações.

3.1.1 B-tree

As B-tree são apropriadas para consultas envolvendo igualdades e intervalos em que os dados estão ordenados por algum critério. O query planner do PostgreSQL considera o uso de índices B-Tree sempre que uma coluna que está indexada está envolvida numa comparação utilizando um dos seguintes operadores <, <=, =, >= e >. Este tipo de índice pode ser utilizado também para fazer procuras por operadores do tipo pattern matching (ex. LIKE)

3.1.2 Hash

Os índices de *Hash* só são utilizados quando as consultas envolvem comparações (operador =). O *Planeador / Optimizador* do *PostgreSQL* considera o uso de índices *hash* quando uma coluna para a qual existe um índice, é envolvida numa

consulta usando o operador =. O comando seguinte pode ser utilizado para criar um índice *Hash*:

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

Deve-se ter em atenção que os índices de *Hash* devem ser recalculados com REINDEX depois de uma falha na Base de Dados se houverem atualizações não gravadas. Além de que alterações aos índices não são replicadas através de streaming ou pela replicação de ficheiros, depois de um backup inicial. Então os índices podem dar respostas erradas a consultas que subsequentemente os usem. Por estas razões, o uso de índices de *Hash* é fortemente desaconselhado.

3.1.3 GiST

Os índices GiST(Generalized Search Tree) não são um tipo simples de índice, mas antes uma infraestrutura em que várias estratégias de indexação podem ser implementadas [46]. Os operadores que podem ser usados dependem da estratégia de indexação que foi utilizada. Por exemplo, o PostgreSQL inclui por defeito operadores GiST para dados bidimensionais. Estes suportam perguntas contendo os operadores: $<<, >>, \&<, \&>, >>, <<|, \&<|, |\&>, |>>, @>, <@, =$ $e \&\&$.

3.1.4 SP-GiST

Os índices SP-GiST, tal como os índices GiST, oferecem uma infraestrutura que suporta vários tipos de pesquisas. Estes permitem a implementação de um vasto leque de diferentes estruturas não balanceadas baseadas em disco, tal como *quadtrees*, *k-d trees* e *radix trees*.

3.1.5 GIN

Os índices GIN são índices invertidos que suportam valores que têm mais do que uma chave, por exemplo, *arrays*. A distribuição standard do GIN inclui operadores para *arrays* uni-dimensionais, que suportando perguntas que contêm os operadores: <@, @>, = e &&.

3.2 Cluster

Quando é feito *cluster* de uma tabela, ela é fisicamente armazenada tendo em conta a informação que está no índice[44]. *Clustering* é uma operação de uma só vez, ou seja, quando a tabela é actualizada, as mudanças não são *clustered*, ou seja, não é feita nenhuma tentativa para gravar ou actualizar os novos tuplos de acordo com a sua ordem no índice.

3.2.1 Cluster sem parâmetros

Ao executar o comando *cluster* sem parâmetros, faz com que o *PostgreSQL* recalcule todas as tabelas *clustered* que o utilizador possui. Se o comando for executado pelo super-utilizador, todas as tabelas serão re-calculadas. Este comando (CLUSTER sem parâmetros) não pode ser executado dentro de uma transação.

Exemplo:

```
CLUSTER employees USING employees_ind;
```

O comando CLUSTER não está disponível no SQL standard.[49]

3.3 Limitação ao número de índices

Em *PostgreSQL* não está definida um limitação em relação ao número de índices sobre o mesmo conjunto de atributos. Como se pode ver pelo exemplo seguinte, é possível criar dois index sobre os mesmos atributos.

```
CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
CREATE INDEX test_index ON test_table (col varchar_pattern_ops) USING
hash;
```

3.4 Estruturas temporariamente inconsistentes

SET CONSTRAINTS - definir as restrições que são aplicáveis à transacção corrente[51].

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

IMEDIATE - são verificadas no final de cada instrução;

DEFFERED - não são verificadas até que a transacção faça *commit*.

NOT NULL e CHECK são sempre verificáveis imediatamente quando uma linha é inserida ou atualizada (e não no fim da transacção).

Este comando está de acordo com o comportamento definido no SQL standard, excepto pela limitação que, em *PostgreSQL* , não são aplicáveis as restrições NOT NULL e CHECK. Além de que e como foi anteriormente dito, o *PostgreSQL* em transacção não-DEFFERED verifica a unicidade imediatamente e não no final da transacção, tal como o standard sugere.

3.5 Comparação com o *Oracle®*

- Estruturas de indexação - O *Oracle®* implementa tudo o que o *PostgreSQL* implementa, excepto os índices GIST, SP_GIST e GIN. [52]
- Multi-index para os mesmos atributos - Sim, é permitido vários conjuntos de índices sobre os mesmos atributos.

Capítulo 4

Processamento e Otimização de Perguntas

4.1 Etapas no processamento e otimização de uma Pergunta

O processamento de uma pergunta passa por cinco etapas distintas:

Transmissão da Pergunta - A aplicação cliente transmite a pergunta para o servidor e espera que este devolva o resultado. É necessário que esteja estabelecida uma ligação entre a aplicação e o servidor *PostgreSQL* .

Parser - Valida a sintaxe da pergunta recebida e cria uma *query tree* da respetiva pergunta.

Sistema de Reescrita - recebe a *query tree* criada pelo *parser* e procura regras (que estão no catálogo do sistema) que possam ser aplicadas à *query tree*. Efetua as transformações que se encontram no corpo de cada uma das regras encontradas.

Planeador / Optimizador - cria um *query plan* da pergunta reescrita pelo *Sistema de Reescrita* procurando encontrar um plano que minimize o custo de execução da pergunta.

Executor - Percorre recursivamente a árvore do *query plan* e em cada nó da árvore obtém os tuplos de acordo com o método definido no plano. Ao analisar relações, efetua ordenações e junções usando o Sistema de Armazenamento para eventuais resultados intermédios. No fim, devolve os tuplos obtidos.

4.1.1 Transmissão da Pergunta

O *PostgreSQL* é implementado segundo uma arquitetura cliente/servidor em que um cliente estabelece uma ligação *TCP/IP* com exatamente um *processo servidor*. No servidor um processo principal (*master process*) lança um novo *processo servidor* quando recebe um pedido de um cliente para iniciar uma nova conexão. Um *processo cliente* pode ser qualquer aplicação que implemente o protocolo do *PostgreSQL* [2].

Após uma ligação estar estabelecida, o *processo cliente* pode enviar uma pergunta para o servidor. A pergunta é transmitida usando texto simples, sem que qualquer pré-processamento seja efetuado no cliente, delegando toda a carga computacional desta operação ao servidor.

4.1.2 Parser

A etapa do *Parser* divide-se em duas partes:

- O *parser* que valida a sintaxe da pergunta. Implementado com as ferramentas *bison* e *flex* [1];
- O *processo de transformação* que efetua alterações necessárias nas estruturas de dados criadas pelo *parser*.

partioning

Parser

Dada uma pergunta (em texto simples), o *parser* irá verificar e validar a sintaxe da pergunta. Caso a validação tenha sucesso é criada uma *parse tree* que será passada à fase seguinte. Caso a sintaxe seja inválida é retornado erro.

Processo de Transformação

O *parser* apenas analisa a pergunta de acordo com as regras sintáticas do SQL.

O *Processo de Transformação* usa a *parse tree* devolvida e analisa semanticamente a árvore de forma a inferir que tabelas, funções e operadores são referenciados na pergunta. Desta análise, resulta uma estrutura de dados que representa a informação obtida. Esta estrutura de dados denomina-se *query tree*.

4.1.3 Sistema de Reescrita

O *PostgreSQL* suporta um sistema de regras (*rule system*) para a especificação e atualização de vistas. Atualmente, este sistema é implementado usando a técnica *query rewriting*, que pode ser vista como uma expansão de macros. O

[1] Ferramentas disponíveis no Unix. O *bison* [3] é um gerador de *parser's* que gera um *parser* a partir de uma gramática livre de contexto. O *flex* [4] gera um programa que reconhece padrões lexicográficos em texto (denominado por *Scanner* ou *Tokenizer*).

Sistema de Reescrita é um módulo que existe entre o *parser* e o *Planeador / Optimizador*.

O *Sistema de Reescrita* recebe uma *parse tree* proveniente do *parser* e aplica as regras possíveis. As regras fazem parte do *rule system* e estão especificadas no catálogo do *PostgreSQL* , na tabela **pg_rules**[1].

Quando é efetuada uma pergunta sobre uma view (uma tabela virtual), o *Sistema de Reescrita* transforma a pergunta numa nova pergunta, que acede diretamente às tabelas base na definição da respetiva vista.

4.1.4 Planeador / Optimizador

O objetivo do *Planeador/Optimizador* é criar um plano de execução ótimo Contudo, o plano de execução criado poderá não ser o ótimo

Uma *query tree* de uma pergunta SQL pode ser executada de várias formas. Novas formas de execução de uma determinada pergunta obtêm-se aplicando regras de equivalência de álgebra relacional [5]. O optimizador de perguntas examina todos os possíveis planos de execução, caso isso seja computacionalmente viável e, seleciona o plano de execução com o menor custo estimado. Os custos das várias operações podem ser configurados. Os valores por omissão podem ser consultados no capítulo *18.7.2. Planner Cost Constants*[7] do manual de configuração do *PostgreSQL* .

A partir de uma *query tree* reescrita pelo *Sistema de Reescrita* o planeador explora todos os planos possíveis que produzam o mesmo resultado. Durante a pesquisa o planeador utiliza estruturas de dados denominadas por *paths* em que cada *path* é uma representação mínima de um plano contendo apenas informação que o planeador necessita para tomar decisões.

Após a escolha do melhor plano de execução, é construída uma *plan tree* que é passada ao *Executor*.

Existem perguntas em que não é viável explorar todos os planos possíveis de execução dado esta operação demorar muito tempo e consumir demasiado espaço em memória. Um caso prático desta situação é por exemplo, quando uma pergunta envolve efetuar um número elevado de junções. Nestes casos, o *PostgreSQL* recorre ao *Genetic Query Optimizer*[8] de forma a encontrar um plano de execução razoavelmente bom. Em particular, quando o número de junções excede um determinado valor limite (12, por omissão) definido em *geqo_threshold (integer)* [6], é invocado o *Genetic Query Optimizer* de forma a obter um bom plano de execução. Em perguntas complexas, compensa executar um plano subóptimo dado que pesquisa exaustiva neste tipo de perguntas irá demorar muito tempo.

Apesar de na documentação consultada não ter sido encontrado um limite máximo para o número de junções numa pergunta, o *Oracle*® restringe o número de atributos de uma relação[23] (1000), o que implicitamente, limita o número de junções.

No *PostgreSQL* não existe limites ao número de junções nem ao número de atributos numa relação.

Estratégias de iteração de relações

O *PostgreSQL* suporta três estratégias distintas para iterar sobre relações:

Sequential Scan (*Seq Scan*) - A relação é totalmente percorrida pela ordem que se encontra no disco;

Index Scan - Efetua um percurso pelas folhas de uma *B-tree* de forma a encontrar os tuplos correspondentes e devolve esses tuplos;

Index-only Scans - Podem ser usados nalguns tipos de perguntas e permitem efetuar o processamento da pergunta sem que haja acessos a tabelas, ou seja, utilizando apenas o índice. Este mecanismo foi introduzido na versão 9.2 do *PostgreSQL* e melhorou bastante a eficiência na execução de algumas perguntas, uma vez que o número de operações de *I/O* é bastante menor, dado não existirem acessos às relações.

Bitmap Index Scan - Percorre o índice de forma a obter todos os apontadores para os tuplos, ordena-os usando uma estrutura de dados *bitmap* em memória e visita cada um dos apontadores de tuplos respeitando a ordem com que estes estão armazenados. Este tipo de mecanismo é muito eficiente em perguntas que usam operações como o *COUNT* ou *AVG* uma vez que apenas é necessário percorrer a estrutura de dados *bitmap* e contar o número de bits.

O *Bitmap Index Scan* não está disponível no *Oracle®* . Contudo, no *Oracle®* é implementada adicionalmente a estratégia *Sample Table Scans* que consiste em ler apenas uma percentagem aleatória de uma relação [43]. Esta estratégia é bastante útil em aplicação de *Data-Warehousing* e *Data-Mining* uma vez que, em vez de percorrer uma relação por completo, permite obter apenas uma amostra aleatória (o que em certos contextos, poderá ser suficiente) da relação.

Geração de planos

Em primeiro lugar, o *Planeador/Optimizador* começa por gerar planos para percorrer cada uma das relações referenciadas na pergunta. Estes planos são obtidos tendo em conta os índices disponíveis em cada relação. Como é sempre possível efetuar um *sequential scan* numa relação, é sempre criado um plano com esta estratégia.

De acordo com os índices disponíveis e caso a pergunta contenha uma restrição do tipo $< nomeRelacao.atributo\ OPERADOR\ constante >$ na cláusula *WHERE*, são criados os seguintes planos:

- Um plano em que é utilizado o índice da relação para percorrer a relação, caso *nomeRelacao.atributo* corresponde à chave do índice existente e *OPERADOR* pertence à classe de operadores[9] do índice;
- Um plano para cada índice em que as restrições na pergunta correspondem à chave desse índice;

- Um plano para cada índice cuja ordem de ordenação corresponda à cláusula *ORDER BY* da pergunta ou que seja útil caso seja utilizado o *merge join* numa junção.

Após encontrados todos os planos para iterar sobre uma relação, são considerados planos para junções de relações existentes na pergunta.

Tal como o *Oracle*® [22] o *PostgreSQL* permite efectuar *cache* de planos de execução. Para tal o utilizador deverá utilizar o comando *PREPARE* da seguinte forma:

PREPARE name [(data_type [, ...])] AS statement
em que:

name - é um identificador único na sessão atual;

data_type - tipo de dados da expressão *statement*. Este campo é opcional e, caso não seja especificado, será automaticamente inferido;

statement - Uma expressão com *SELECT*, *INSERT*, *UPDATE*, *DELETE* ou *VALUES*.

Figura 4.1: Comando *PREPARE* para *caching* de planos.

A *cache* de planos é referente a cada sessão e não é mantida de sessão para sessão.

Algoritmos de junção suportados

O *PostgreSQL* implementa os seguintes algoritmos de junção:

- *nested loop join* - Neste algoritmo a relação da direita (*outer relation*) é percorrida uma vez para cada tuplo na relação à esquerda (*inner relation*). Apesar de este algoritmo ser pouco eficiente, se a *outer relation* poder ser iterada com um índice, poderá tornar-se uma boa estratégia;
- *merge join* - Em primeiro lugar cada relação é ordenada pelo(s) atributo(s) de junção. De seguida as duas relações são percorridas em paralelo e os tuplos de ambas as relações que correspondem (segundo os atributos de junção) são combinados e adicionados ao resultado. Neste algoritmo cada relação só necessita de ser iterada uma vez. O *PostgreSQL* percorre o índice de cada relação cuja chave terá que corresponder à chave de junção ou, caso não exista esse índice, efetua uma ordenação explícita da(s) relação(ões).

- *hash join* - Neste algoritmo a *outer relation* é percorrida e é construída a *hash table* correspondente, usando os seus atributos de junção como *hash keys* (chaves da *hash table*. De seguida, a *inner relation* é percorrida e os valores apropriados de cada tuplo encontrados são usados como *hash keys* para localizar os tuplos correspondentes na *hash table*.

Ao gerar planos de execução, o *Optimizador/Planeador* considera a aplicação de todos os algoritmos suportados pelo *PostgreSQL* . Quando é necessário efetuar mais de uma junção (quando é referenciada mais de uma relação na pergunta) o *Optimizador/Planeador* considera todas as permutações possíveis de estratégias de junção. Nesta situação o resultado será uma árvore com vários passos de junção, com um respetivo algoritmo, em cada nó.

O *Oracle*® também implementa estes três algoritmos.

Quando uma pergunta não referência mais de *geqo_threshold*[6] relações o *PostgreSQL* efetua uma *near-exhaustive search* (introduzida no IBM System R [11]). A pesquisa não é exaustiva porque o *Optimizador/Planeador* utiliza uma heurística que consiste em dar preferência a junções entre relações para as quais os atributos de junção ocorrem na cláusula *WHERE* (por exemplo, em *equi-joins* como $< relacaoA.atributoA = relacaoB.atributoB >$). Além disso, pares de relações sem uma cláusula de junção apenas são considerados senão existir nenhum outro plano. Caso exista, este plano é descartado.

Todos os planos possíveis são gerados para cada junção. É escolhido o plano cuja estimativa do custo seja menor.

Quando o número de junções é maior ou igual a *geqo_threshold*[6] o *PostgreSQL* invoca o *Genetic Query Optimizer*.

Por exemplo, na pergunta da figura 4.2 o optimizador opta por aplicar o algoritmo *merge-join* na junção (plano pode ser visualizado na figura 4.3 e a árvore do plano na figura 4.4). A razão deve-se ao facto de existir um índice *B-tree* no atributo *film_id* em cada relação, e dado a existência da cláusula *ORDER BY*. Ao utilizar o *merge-join* nesta situação não é necessário efetuar uma ordenação explícita.

```
SELECT *
FROM film inner join film_ator ON film.film_id = film_actor.film_id
ORDER BY film.film_id
```

Figura 4.2: Exemplo de pergunta em que o *Planeador/Optimizador* opta pelo *merge-join*.

```
"Merge Join (cost=0.56..433.86 rows=5462 width=396)"
"  Merge Cond: (film.film_id = film_atorfilm_id)"
"  ->  Index Scan using film_id_idx on film  (cost=0.28..92.92 rows=1000
width=384)"
"  ->  Index Scan using idx_fk_film_id on film_ator  (cost=0.28..270.17 rows=5462
width=12)"
```

Figura 4.3: Plano de execução gerado para a pergunta na figura 4.2.

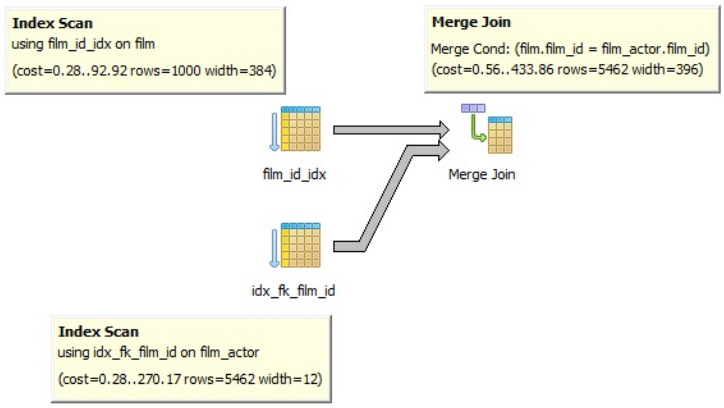


Figura 4.4: Árvore do plano de execução gerado para a pergunta na figura 4.2.

O *Planeador/Optimizador* também coloca eventuais condições de seleções na cláusula *WHERE* e atributos na cláusula *SELECT* nos sítios mais apropriados da árvore do plano de execução. Por exemplo, coloca os operadores de álgebra relacional, de seleção e de projeção, o mais perto possível das folhas da árvore do plano de execução, favorecendo as operações nos nós mais acima.

Sub-perguntas (*subqueries*) criadas com a cláusula *IN* são convertidas numa junção de forma a tirar partido da otimização efetuada às junções, pelo *Planeador/Optimizador*. Quanto a sub-perguntas criadas na cláusula *FROM*, são também convertidas numa junção caso não contenham *GROUP BY, HAVING, ORDER BY* ou funções de agregação. Nos restantes casos, o optimizador cria um novo plano para executar a sub-pergunta[10]. A conversão das sub-perguntas em junções além de permitir ao *Planeador/Optimizador* aplicar as otimizações já descritas das junções, permite otimizar melhor a pergunta (incluindo a sub-pergunta) uma vez que a otimização é efetuada sobre a pergunta como um todo.

Dada a seguinte pergunta que contém uma sub-pergunta na cláusula *WHERE*: Na figura 4.5 encontra-se uma pergunta com uma sub-pergunta.

```
SELECT *
FROM film
WHERE EXISTS (
    SELECT film.film_id
    FROM film_category
    WHERE film.film_id = film_category.film_id and
       film_category.category_id = 1
);
```

Figura 4.5: Exemplo de pergunta com uma sub-pergunta.

O plano gerado pelo *PostgreSQL* para a pergunta 4.5 pode ser visualizado na figura 4.6. Com a ferramenta *pgAdmin* 7.4 é possível visualizar a árvore do plano (figura 4.7). Como se vê no plano 4.6, o *Planeador/Optimizador* converteu a sub-pergunta numa junção, neste caso, num *hash join*. Esta pergunta exemplifica um tipo de junção especial, o *semi-join*. Um *semi-join* é uma junção em que só são considerados tuplos da relação à esquerda na junção que correspondem a tuplos na relação à direita.

```
"Hash Semi Join (cost=19.30..86.64 rows=64 width=384)"
"  Hash Cond: (film.film_id = film_category.film_id)"
"  -> Seq Scan on film (cost=0.00..64.00 rows=1000 width=384)"
"  -> Hash (cost=18.50..18.50 rows=64 width=2)"
"     -> Seq Scan on film_category (cost=0.00..18.50 rows=64 width=2)"
"        Filter: (category_id = 1)"
```

Figura 4.6: Plano de execução gerado para a pergunta na figura 4.5.

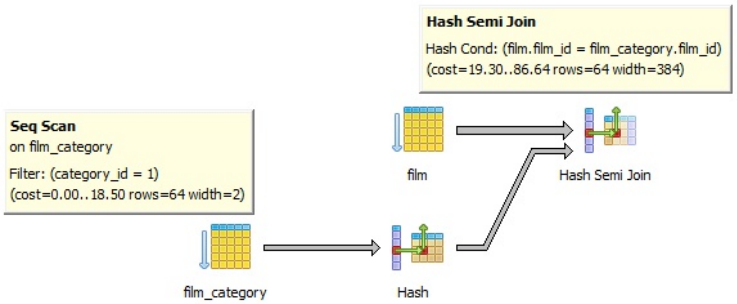


Figura 4.7: Árvore do plano de execução gerado para a pergunta na figura 4.5.

A árvore do plano resultante é composta por iterações sequenciais ou sobre índices às relações base, operações de junção (com *nested-loop*, *merge-join* ou *hash-join* de acordo com as decisões tomadas pelo *Planeador/Optimizador*) e outros passos adicionais como ordenações em certos nós da árvore ou cálculo de funções de agregações.

Visualização de Planos

O *PostgreSQL* permite visualizar o plano escolhido pelo *Planeador/Optimizador* para uma dada pergunta. Para o efeito, é utilizado o comando *EXPLAIN* da seguinte forma:

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement

onde option pode ser:

ANALYZE [ boolean ]
VERBOSE [ boolean ]
COSTS [ boolean ]
BUFFERS [ boolean ]
TIMING [ boolean ]
FORMAT  TEXT | XML | JSON | YAML
```

Figura 4.8: Sintaxe comando *EXPLAIN*.

Este comando mostra o plano de execução da pergunta. O plano de execução contém o método de iteração sobre as relações, quais os algoritmos de junção

utilizados numa junção e quais as relações que participam nessa junção. Adicionalmente, o comando permite saber o custo estimado da execução do plano criado.

Com a opção *ANALYZE*, não só irá ser calculado um plano de execução para a pergunta em *statement* como esta será executada. Com esta opção serão mostradas estatísticas sobre a execução da pergunta tais como o tempo total gasto em cada nó de um plano e o número total de tuplos retornados.

Genetic Query Optimizer

O operador de junção é o operador relacional mais difícil de processar e otimizar. Isto porque o número de planos possíveis cresce exponencialmente com o número de junções na pergunta. Dado esta situação, a técnica usual usada pelo *PostgreSQL* para otimizar a pergunta não é adequada para perguntas com muitas operações de junção, dada a sua ineficiência.

Para resolver este problema, o *PostgreSQL* implementa um algoritmo genético para tornar a otimização deste tipo de perguntas eficiente.

O algoritmo genético (AG) é um método de otimização heurística que opera sobre pesquisa aleatória. O conjunto de soluções possíveis do problema de otimização é considerado como uma população de indivíduos. Apesar de ser um método estocástico, o AG não é totalmente aleatório.

O *Genetic Query Optimizer* vê o problema da otimização de uma pergunta como uma instância do problema *traveling salesman problem (TSP)*. Os planos possíveis de uma pergunta são codificados como strings em que cada string representa a ordem de junção de uma relação da pergunta a outra relação.

Algumas características específicas da implementação do *Genetic Query Optimizer* no *PostgreSQL* são:

- Substituição do individuo com pior valor da função de *fitness*, em vez de substituição de toda a geração (técnica *steady state*) que permite convergir rapidamente melhores planos;
- O operador de mutação não é utilizado de modo a não ser necessário mecanismos de reparação das soluções (por exemplo, para garantir a consistência de um circuito no problema *TSP*),
- Utilização do operador *edge recombination crossover*[12] que é bastante adequado a problemas de *TSP*[13].

Apesar de não garantir o plano ótimo, o *Genetic Query Optimizer* permite obter planos razoavelmente bons.

4.1.5 Executor

O *Executor* recebe o plano criado pelo *Planeador/Optimizador* e processa-o recursivamente de forma a extrair o conjunto de tuplos necessários. Ou seja, o *Executor* utiliza um mecanismo de *demand-driven pipeline*. Cada operação de

um nó solicita o próximo tuplo às operações nos nós filhos, de forma a retornar o seu próximo tuplo.

Os quatro tipos básicos de perguntas *SQL* (*SELECT*, *INSERT*, *UPDATE* e *DELETE*) são todos avaliados pelo *Executor*:

***SELECT* -** O *Executor* envia cada tuplo retornado na execução do plano da pergunta para o cliente;

***INSERT* -** Cada tuplo retornado é inserido na tabela destino especificada na pergunta;

***UPDATE* -** O *Planeador/Optimizador* arranja cada tuplo de forma a incluírem todos os valores atualizados em cada coluna e o *TID* (ID do tuplo) do tuplo original. Um novo tuplo atualizado é criado e o tuplo antigo é marcado como *apagado*;

***DELETE* -** O único atributo que é retornado pela execução do plano é o *TID*. O *TID* é depois usado para visitar cada tuplo e marcá-lo como *apagado*.

Paralelismo

O *PostgreSQL* suporta paralelismo total no lado do cliente. As aplicações podem estabelecer múltiplas ligações ao servidor e geri-las assincronamente ou com *threads*.

No servidor o *PostgreSQL* oferece pouco suporte a paralelismo, no entanto são suportados os seguintes tipos de paralelismo:

- *effective_io_concurrency(integer)*[19] que permite definir o número esperado de operações de *I/O* executadas em simultâneo. Aumentar este valor, irá aumentar o número de operações de *I/O* que cada sessão *PostgreSQL* tenta iniciar em paralelo. Atualmente, esta definição apenas afeta iterações em relações utilizando índices *Bitmap*;
- Linguagens utilizadas no servidor com suporte a paralelismo. Permitem paralelizar operações.

4.2 Estatísticas

O *PostgreSQL* tem um sistema, *PostgreSQL Statistics Collector*, que suporta a recolha e armazenamento de informação sobre a atividade do servidor. Este sistema conta os acessos a tabelas e índices quer em termos de *disk-block* quer em *individual-row*.

As estatísticas recolhidas podem ser consultadas acedendo às vistas definidas pelo *PostgreSQL* , que se encontram especificadas na documentação [20].

4.3 Hints

O *PostgreSQL* não permite fornecer *hints* numa pergunta. Por exemplo, não é possível forçar a utilização de um certo algoritmo de junção ou um certo mecanismo de iteração sobre uma relação. A equipa do *PostgreSQL* justifica esta decisão na seguinte declaração[21]:

> We are not interested in implementing hints in the exact ways they are commonly implemented on other databases. Proposals based on "because they've got them"will not be welcomed. If you have an idea that avoids the problems that have been observed with other hint systems, that could lead to valuable discussion.

Figura 4.9: Declaração equipa *PostgreSQL* sobre *Hints*.

Os principais problemas das *hints* apontados pela equipa do *PostgreSQL* são:

- Poor application code maintainability: hints in queries require massive refactoring;
- Interference with upgrades: today's helpful hints become anti-performance after an upgrade;
- Encouraging bad DBA habits slap a hint on instead of figuring out the real issue;
- Does not scale with data size: the hint that's right when a table is small is likely to be wrong when it gets larger;
- Failure to actually improve query performance: most of the time, the optimizer is actually right;
- Interfering with improving the query planner: people who use hints seldom report the query problem to the project.

Figura 4.10: Problemas da utilização de *Hints*.

Contudo, é possível forçar a utilização do *Index Scan* para testes, embora não seja recomendado. O *PostgreSQL* permite desativar"*Sequential Scans* numa sessão da seguinte forma:

```
SET enable_seqscan = OFF;
```

Figura 4.11: Comando para desativar *Sequential Scan's*.

Na verdade, não é possível desativar por completo o mecanismo de *Sequential Scan* dado que por vezes, é a único mecanismo possível de ser utilizado. Contudo, nos restantes casos, não é considerado pelo *Optimizador/Planeador*. Apesar deste comando estar disponível, a sua utilização não é de todo necessária. O *Optimizador/Planeador* é robusto e suficientemente inteligente para decidir qual a melhor estratégia de iteração de uma relação.

O *Oracle*® permite ao utilizador utilizar *Hints*. Suporta *Hints* de transformação de perguntas, ordens de junções, definição dos algoritmos de junção e mecanismos de paralelização. Ao fornecer uma *hint* o utilizador está a tomar uma decisão pelo optimizador, o que nem sempre é recomendado. O arquiteto da base de dados, pode decidir fornecer uma *Hint* ao SGBD numa determinada altura para o processamento de uma determinada pergunta, no entanto, no futuro as relações referenciadas nessa pergunta poderão mudar por completo (por exemplo, aumento do número de tuplos) de uma forma que não foi prevista, levando a uma execução ineficiente. Assim sendo, o *PostgreSQL* protege uma base de dados dessas situações recorrendo sempre ao *Optimizador/Planeador* para obter o melhor plano de execução (ou um plano subóptimo).

Capítulo 5

Gestão de Transações e Controlo de Concorrência

5.1 Controlo de Concorrência

5.1.1 Propriedades ACID

Uma transação pode ser definida como uma "unidade do programa de execução que acede e possivelmente altera vários dados".

Estas transações são necessárias para, por exemplo, efetuar movimentos de contas nos bancos, ou para marcar voos num avião, ou para efetuar compras *online*, onde temos de ter a garantia que todas as operações são executadas ou nenhuma delas é executada.[73]

Por exemplo, imaginemos um caso onde a Alice quer pagar 100.00 dólares ao Bob através de uma transferência no banco. O conjunto de operações seria mais ou menos este:

```
UPDATE accounts SET balance = balance - 100.00 WHERE name = 'Alice';

UPDATE branches SET balance = balance - 100.00
      WHERE name = (SELECT branchName FROM accounts
      WHERE name = 'Alice');

UPDATE accounts SET balance = balance + 100.00
      WHERE name = 'Bob';

UPDATE branches SET balance = balance + 100.00
      WHERE name = (SELECT branchName FROM accounts
      WHERE name = 'Bob');
```

Nesta transação é importante que todas as operações aconteçam ou se algo falhar, nenhuma aconteça. Seria muito aborrecido se o Bob recebesse 100 dólares que não foram depositados pela Alice, ou se a Alice depositasse 100 dólares e o Bob não os recebesse.

Em *PostgreSQL* , para iniciar uma transação, temos de colocar o comando **BEGIN** no inicio da transação e **COMMIT** no fim.

A transação do banco ficaria então escrito desta forma:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
        WHERE name = 'Alice';
...

COMMIT;
```

A transação pode também ser cancelada a meio, se por exemplo se verificar que o saldo da Alice fica negativo, utilizando o comando **ROLLBACK**.

Além disso, para isto acontecer sem falhas, temos que prever erros de vários tipos (sejam por software ou por hardware) e também permitir múltiplas execuções de transações concorrentes. Afinal, no Natal existem milhares, se não milhões de transações a cada minuto nos bancos para se efetuar as compras. Imaginemos o caos que seria se de repente, todos os multi-bancos parassem de funcionar por causa de um *crash* na base de dados.

Também é possível definir pontos de controlo numa transação através do comando **savepoint**. Estes pontos de controlo permitem descartar partes de uma transação, mas fazer *commit* do resto. Podemos então utilizar o comando **ROLLBACK TO** para regressar a um determinado ponto de controlo.

Por exemplo, imaginemos que durante uma transação se tinha retirado 100 dólares à Alice e depositado esse dinheiro na conta do Bob, mas afinal devíamos ter depositado na conta do Wally. Poderíamos corrigir parte da transação desta forma:

```
BEGIN;

UPDATE accounts SET balance = balance - 100.00
        WHERE name = 'Alice';

SAVEPOINT oMeuPontoDeControlo;

UPDATE accounts SET balance = balance + 100.00
        WHERE name = 'Bob';

– oops ... Era para ter sido a conta do Wally

ROLLBACK TO oMeuPontoDeControlo;

UPDATE accounts SET balance = balance + 100.00
        WHERE name = 'Wally';

COMMIT;
```

Pontos de controlo oferecem um grande controlo sobre as transações. Além disso, *ROLLBACK TO* é a única forma de ter controlo sobre uma transação que teve de abortar por causa de um erro, e não queremos começar a transação toda de novo.

Quando várias transações tentam aceder ao mesmo tempo a uma base de dados, é necessário garantir algumas propriedades para que não haja um completo caos com transações a "atropelarem-se"umas às outras, ou a deixarem estados inconsistentes nos dados.

Foram então definidas algumas propriedades de transações chamadas **ACID**, *Atomicity, Concistency, Isolation, Durability.*

- ***A****tomicity*, atomicidade é a propriedade que garante que, ou todas as operações da transação são propriamente reflectidas na base de dados ou nenhuma delas é executada. Isto evita que transações apenas parcialmente executadas alterem a base de dados.

- ***C****onsistency*, consistência garante que, uma transação que ao iniciar encontre uma versão consistente da base de dados, então ao terminar, deixa uma versão consistente da mesma. Os requisitos mais utilizados são as restrições de integridade explicitamente impostas e as relações com chaves primárias e estrangeiras. Durante uma transação podem existir dados inconsistentes, mas no fim as restrições têm de ser verificadas.

- ***I****solation*, isolamento é a propriedade que garante que quando múltiplas transações concorrentes estão a ser executadas, cada transação tem de

estar "escondida"das outras, ou seja, nenhuma transação pode saber que outras transações também estão a aceder aos mesmos dados. Isto poderia ser facilmente resolvido se todas as transações fossem executadas sequencialmente, mas poder executá-las concorrentemente trás muitas vantagens.

- ***D**urability*, durabilidade é a propriedade que garante que uma vez que o utilizador esteja notificado do fim de uma transação, todas as alterações executadas na base de dados têm de persistir mesmo no caso de falhas de hardware ou software. Pode parecer uma propriedade óbvia, mas quando estamos na presença de ações que não podemos reverter, com mensagens enviadas para o monitor, é necessário garantir que não há informações enganosas. ...

Para a parte da consistência podemos definir restrições através de:

```
SET CONSTRAINTS  ALL | nome [, ...]    DEFERRED | IMMEDIATE
```

Se a restrição for criada com modo *IMMEDIATE*, então esta é verificada depois de cada operação. Se for *DEFERRED* apenas é verificada quando a transação fizer *commit*.[72]

5.1.2 Controlo de concorrência em PostgreSQL

Em PostgreSQL, o controlo de concorrência é feito através de um mecanismo multi-versão chamado **MVCC** (*Multi-version Concurrency Control*). Este modelo utiliza um sistema de *locks*[74] quando são feitas perguntas à base de dados ao mesmo tempo que mantém algumas cópias de versões mais antigas dos dados aumentando a concorrência, pois cada transação vê como que uma "fotografia"da base de dados, não interessando o estado corrente dos mesmos.[71]

Isto permite que uma pergunta de leitura nunca bloqueia uma escrita e vice-versa, garantindo que uma transação não veja um estado inconsistente da base de dados causada por uma outra transação que por acaso está a alterar os mesmos dados, atingindo assim uma das propriedades ACID, o **isolamento**, mantendo uma ótima performance mesmo quando utilizando o nível mais restrito de isolamento.

- MVCC - Como funciona -

Cada transação quando iniciada recebe um *ID* chamado *XID*, que é incrementado de cada vez que o *PostgreSQL* tem de o atribuir a uma transação. Também é guardada informação sobre cada linha na base de dados para se poder saber se a linha é visível a outras transações.[70]

Quando uma transação insere uma linha na BD, esta recebe um *xmin* que é igual ao *XID* da transação que a inseriu. Todas as linhas que estejam *commited*,

estão visíveis a todas as transações que tenham um *XID* maior que o *xmin* dessa linha. Um mecanismo semelhante é utilizado para apagar ou alterar linhas, mas em vez de um *xmin*, é dado um *xmax* para determinar visibilidade, como se pode ver pela figura 5.1.

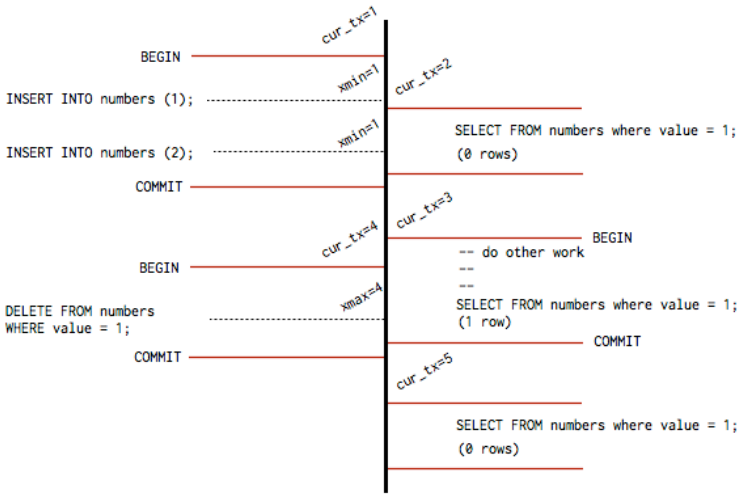


Figura 5.1: Visibilidade dos dados durante a execução de transações concorrentes

Se duas transações tentarem alterar a mesma linha ao mesmo tempo, uma de duas situações podem ocorrer. Se o *PostgreSQL* estiver em modo de isolamento READ COMMITTED, volta a tentar executar as operações onde deu conflito.

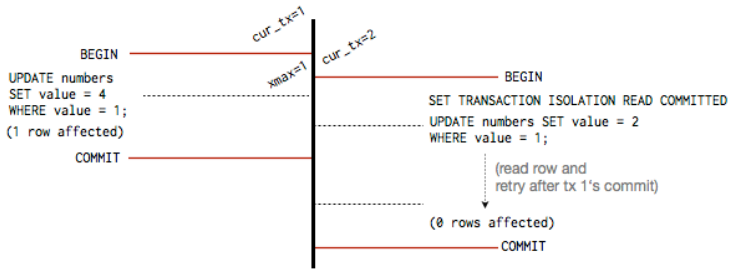


Figura 5.2: Re-execução de uma transação em caso de conflito

Caso o nível de isolamento seja *SERIALIZABLE*, o cenário descrito acima é impossível de resolver, e uma das transações vai falhar, sendo que o *PostgreSQL* devolve uma mensagem de erro.

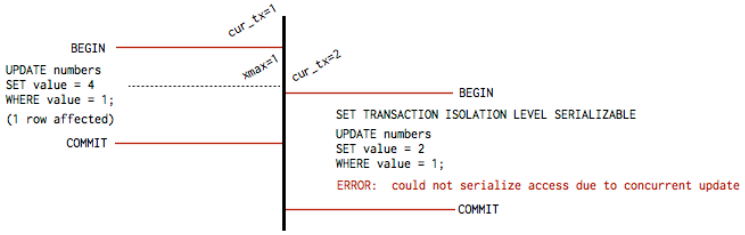


Figura 5.3: Mensagem de erro em caso de conflito

- Desvantagens do MVCC -

Para poder manter várias versões dos dados disponíveis para leitura, *PostgreSQL* também vai manter vários registos obsoletos. Isto acontece pois perguntas que alterem a BD na verdade criam uma linha nova com um *xmax*, e perguntas que apagam linhas na verdade não as apagam, ficam marcadas como apagadas e recebem um *xmax*. Mesmo quando as transações acabam e estas linhas não poderem ser visíveis para nenhuma futura transação, estas ficam na tabela na mesma ocupando imenso espaço desnecessariamente.

PostgreSQL resolve este problema utilizando o VACUUM para fazer a manutenção das tabelas.

5.1.3 Níveis de Isolamento

O *standard* do SQL define quatro níveis de isolamento. O nível mais restrito é o SERIALIZABLE que garante que quando duas transações estão a executar concorrentemente, a sua serialização faz com que o efeito de as executar concorrentemente seja o mesmo de as executar uma após a outra. Os outros três níveis podem ser definidos por fenómenos que não queremos que aconteçam a cada nível.[69]

Os fenómenos que não queremos que aconteçam são os seguintes:

- **Dirty Read**: Uma transação é capaz de ler informação escrita por outra transação que ainda não fez *commit*.
- **Nonrepeateble Read**: Uma transação re-lê informação que foi alterada por outra transação e percebe que essa informação foi alterada e *committed* desde a leitura inicial.
- **Phantom Read**: Uma transação re-executa uma pergunta que devolve um conjunto de tuplos que satisfazem uma determinada condição e percebe que esse conjunto de tuplos foram alterados por uma outra transação que fez *commit* entretanto.

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Figura 5.4: Fenómenos de transações versus níveis de isolamento do SQL

O *PostgreSQL* permite executar os quatro níveis de isolamento, mas internamente apenas três deles funcionam a que corresponde ao *READ COMMITTED*, o *REPEATEBLE READ* e o *SERIALIZABLE*. Quando executamos uma pergunta com nível *READ UNCOMMITTED*, na verdade executa *READ COMMITTED*. A razão pela qual *PostgreSQL* oferece apenas três níveis é devido ao MVCC.

- Níveis de isolamento do *PostgreSQL* -

Read Committed

É o nível de isolamento por defeito no *PostgreSQL* . Quando uma transação executa uma pergunta de leitura utilizando este nível de isolamento, este só pode ler informação que tenha sido *committed* antes da pergunta ter sido efetuada. É importante lembrar que o MVCC faz com que a transação veja uma fotografia da base de dados no momento em que a pergunta começou a ser executada.

Se for uma pergunta de escrita (UPDATE, DELETE, SELECT FOR UPDATE, ou SELECT FOR SHARE), então a transação poderá ver informação que tenha sido alterada. Quando se pesquisa informação com determinadas características, essa informação pode ter sido alterada por outra transação mesmo antes de ter sido encontrada.

Por exemplo:

```
BEGIN;
UPDATE website SET hits = hits + 1;
– run from another session: DELETE FROM website WHERE hits = 10;
COMMIT;
```

Aqui pode-se verificar que o comando que apagaria todas as linhas com *hits* = 10 poderia não apagar nenhum tuplo apesar de existirem tuplos com *hits*=10 antes e depois do UPDATE. A transação DELETE iria saltar todos os tuplos com *hits*=9, então o UPDATE de seguida iria altera-los para 10 e colocar todos

os 10 para 11. Quando o DELETE chegasse aos *hits*=10, estes tinham sido alterados para 11, a transação iria aperceber-se que tinha havido uma alteração e reavaliaria essa pergunta para essas linhas que foram alteradas.

Sendo que agora o número de *hits* é 11, estas já não têm os critérios de pesquisa definidos. A função DELETE chegou ao fim sem apagar nenhum tuplo.

O nível de isolamento Read Committed é adequado para aplicações em que as perguntas executadas e os critérios de pesquisa são muito simples. Outras precisam de níveis um pouco mais restritos.

Repeatable Read

Este nível é um pouco mais restrito que o *Read Committed*, pois uma transação com este isolamento vê a fotografia da base de dados tirada no início dessa transação e não no início de uma operação dessa transação. Assim, SELECT's sucessivos dentro de uma transação vêm sempre os mesmos tuplos.

Se enquanto a transação está a decorrer (usando *Repeatable Read*), outra transação aceder aos dados, a primeira vai esperar que a segunda faça *commit* ou *rollback*. Se fizer *rollback*, então as suas operações serão desfeitas e a primeira transação continua. Se fizer *commit*, então a primeira irá devolver um erro de serialização:

```
ERROR: could not serialize access due to concurrent update.
```

É de salientar que apenas operações de escrita podem precisar de ser re-executadas. Operações apenas de leitura nunca devolverão erro de serialização.

Serializable

O nível de isolamento serializável executa as transações concorrentes como se tivessem sido executadas uma depois da outra, serialmente. Este nível trabalha de forma semelhante à do *Repeateble Read*, mas os monitores procuram condições de transações concorrentes que poderiam levar a dados inconsistentes com todas as execuções possíveis serializadas dessas transações.

Esta monitorização não introduz mais nenhum nível de bloqueio do que o *Repeateble Read*, mas exige mais processamento por parte do monitor que tenta detetar condições que possam causar uma anomalia de serialização, o que leva a um erro de serialização.

Por exemplo:

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

E que a transação *A* executa:

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

e que depois insere o resultado (30) como valor num novo tuplo com class=2. Concorrentemente, a transação *B*, com nível de isolamento serializável executa:

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

e obtém o resultado 300, que insere num novo tuplo com class=1. Então, ambas as transações tentam fazer *commit*. Se alguma das transações estiver a correr com nível de isolamento *Repeateble Read*, então ambas as transações conseguem fazer *commit*.

Se utilizarmos o nível *SERIALIZABLE*, então uma das transações irá fazer *commit* e a outra fará *rollback* devolvendo a seguinte mensagem de erro:

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

Isto acontece pois não existe ordem de execução serializada consistente com os resultados.

Para garantir uma verdadeira serialização, *PostgreSQL* utiliza ***predicative locking***. São locks que permitem identificar se existem dependências entre transações serializadas e não produzem qualquer tipo de bloqueamento nem causam deadlocks.

5.1.4 Locks explícitos

O *PostgreSQL* também permite a utilização explícita de locks para os casos em que o MVCC não oferece o comportamento adequado. Também tem locks automáticos em operações que não são seguras de executar concorrentemente, por

exemplo, o comando *TRUNCATE* adquire automaticamente um lock exclusivo na tabela.[65]

Para utilizar explicitamente locks, executa-se o comando LOCK:

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

Onde *lockmode* é um dos seguintes:
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE

Locks a nível da tabela

Todos os modos são a nível de tabela, mesmo que no comando tenha a palavra ROW (por razões históricas). A única grande diferença entre um modo e outro é o conjunto de modos com que faz conflito. Duas transações não podem ter em sua posse modos que entrem em conflito entre si, ao mesmo tempo, sobre a mesma tabela. Na tabela seguinte mostramos que modos entram em conflito com que outros modos:

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 5.5: Fenómenos de transações versus níveis de isolamento do SQL

É de notar que o modo *ACCESS SHARE* não entra em conflito consigo mesmo, ou seja pode ser utilizado ao mesmo tempo, na mesma tabela, por várias transações, enquanto que o *ACCESS EXCLUSIVE* só pode ser usado por uma única transacção numa tabela num dado momento.

Locks a nível de linha

O *PostgreSQL* também permite locks a nível do tuplo de uma tabela em vez da tabela em si, que podem ser exclusivos ou partilhados.

Quando uma transação altera ou apaga tuplos da tabela, um lock exclusivo é automaticamente adquirido e pertence à transação até que esta faça *commit* ou aborte.

Para adquirir um lock exclusivo a um tuplo sem o modificar utiliza-se o comando SELECT FOR UPDATE e para o adquirir um de leitura utiliza-se o SELECT FOR SHARE.

Por tuplo só uma transação pode ter um lock de UPDATE. Múltiplas transações podem ter locks SHARE, mas nenhuma outra transação pode alterar ou apagar essa linha enquanto alguma transação tiver o lock SHARE.

Deadlocks

A utilização de locks de forma explícita pode originar *deadlocks*. Um deadlock acontece quando, por exemplo, a transacção A detém o lock X e precisa do lock Y que a transação B tem em sua posse, ficando à espera que este o liberte. Mas a transação B precisa do lock X e fica à espera que a transação A o liberte.

Ficam então ambas as transações à espera uma da outra para sempre. O *PostgreSQL* tem formas para automaticamente detetar *deadlocks* e resolvê-los, abortando uma das transações envolvidas (libertando os locks em sua posse) permitindo que as outras transações possam prosseguir.

Um exemplo de ocorrência de um *deadlock*:

Transação A
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;

- - Transação A adquire o lock correspondente à conta 11111.

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

Transação B
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;

- - Transação B adquire o lock correspondente à conta 22222.
- - Tenta adquirir o lock correspondente à conta 11111, mas está em posse da Transação A, então fica á espera que A o liberte.
- - - - - - - - - - - - - - - -

Transação A
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;

- - Tenta adquirir o lock correspondente à conta 22222, mas esse tem a Transação B, então fica á espera que B o liberte.

A ficou à espera que B liberte o lock da conta 22222 e B ficou à espera que A liberte o lock da conta 11111. Ficariam eternamente à espera um do outro. Para evitar isto, o *PostgreSQL* deteta o *deadlock* e aborta uma das operações. Qual a transação que será abortada, é difícil de prever.

É possível utilizar técnicas para evitar *deadlocks*, mas como os locks são executados de forma explícita, depende totalmente da aplicação que utiliza a

base de dados gerir este problema.

5.1.5 Sistema de recuperação - Write Ahead Log

Para garantir atomicidade e durabilidade apesar de falhas como *crashes* do sistema, *PostgreSQL* utiliza uma técnicas baseada em Logs chamada *Write Ahead Log*.

As modificações físicas aos ficheiros onde estão os dados e tabelas deve ser feita apenas depois das descrições das mudanças que vão ser efetuadas estarem escritas no log.[66] Seguindo este procedimento não é necessário fazer vários acessos aos ficheiros no disco onde estão as tabelas de cada vez que uma operação é executada (apenas se modifica o Log), e caso exista uma falha no sistema, é possível recuperar a base de dados utilizando as descrições feitas no Log. Esta é uma técnica chamada *roll-forward recovery - REDO*.

O Log é escrito sequencialmente, o que facilita a sincronização do mesmo custando muito menos do que escrever cada operação. Assim, oferece uma grande vantagem quando o servidor tem muitas transações pequenas mas que afetam vários ficheiros físicos, sendo que uma única leitura do Log, é suficiente para executar totalmente várias transações.

Em caso de falha do sistema, *PostgreSQL* procura o último ponto de controlo (definido quando se tem a certeza que o Log está sincronizado com as escritas na base de dados, e esta está num estado consistente) e refaz as operações que não tenham sido aplicadas fisicamente (do ponto de controlo para a frente) até chegar a um estado consistente.

Exemplo de um Log[67]

total 131076
- rw - - - - 1 pgdba pgdba 16777216 2011-07-14 01:39 000000010000000800000058

- rw - - - - 1 pgdba pgdba 16777216 2011-07-14 01:03 000000010000000800000059

- rw - - - - 1 pgdba pgdba 16777216 2011-07-14 01:03 00000001000000080000005A

- rw - - - - 1 pgdba pgdba 16777216 2011-07-14 01:04 00000001000000080000005B

- rw - - - - 1 pgdba pgdba 16777216 2011-07-14 01:04 00000001000000080000005C

- rw - - - - 1 pgdba pgdba 16777216 2011-07-14 01:04 00000001000000080000005D

- rw - - - - 1 pgdba pgdba 16777216 2011-07-14 01:06 00000001000000080000005E

- rw - - - - 1 pgdba pgdba 16777216 2011-07-14 01:06 00000001000000080000005F

drwx - - - 2 pgdba pgdba 4096 2011-07-14 01:14 archive_status/

Assíncronismo

É possível utilizar o modo *Asynchronous commit* para aumentar a velocidade com que as transações são executadas. Neste modo, em vez de esperarmos que as transações estejam escritas no Log antes de as escrever na base de dados (*synchronous commit*, é o que vem por defeito), assim que uma transação esteja logicamente completada, é devolvida uma mensagem de sucesso.[68]

Não existe perigo de corrupção de dados (pois a reescrita através do Log evita isso), mas em caso de falha, transações que não tiveram tempo de ser colocadas em disco serão descartadas quando o sistema recupera.

Algumas operações como DROP TABLE são obrigatoriamente executadas em *synchronous commit* de forma a garantir um estado de consistência entre a base de dados em disco e a base de dados lógica.

5.1.6 Comparação com a Oracle

O modelo de transações da Oracle utiliza *Rollback Segments*. Estes guardam a informação modificada de cada transação nas próprias tabelas. No entanto, informação necessária para fazer *rollback* é guardada num segmento *rollback*.[76]

Quando a transação precisa de ver um estado consistente da informação, pode necessitar de retirar dados do segmento *rollback* em vez da tabela em si, o que pode levar a erros de "Snapshot too old".

A informação dos segmentos é acumulada e apagada quando a transação está completa.

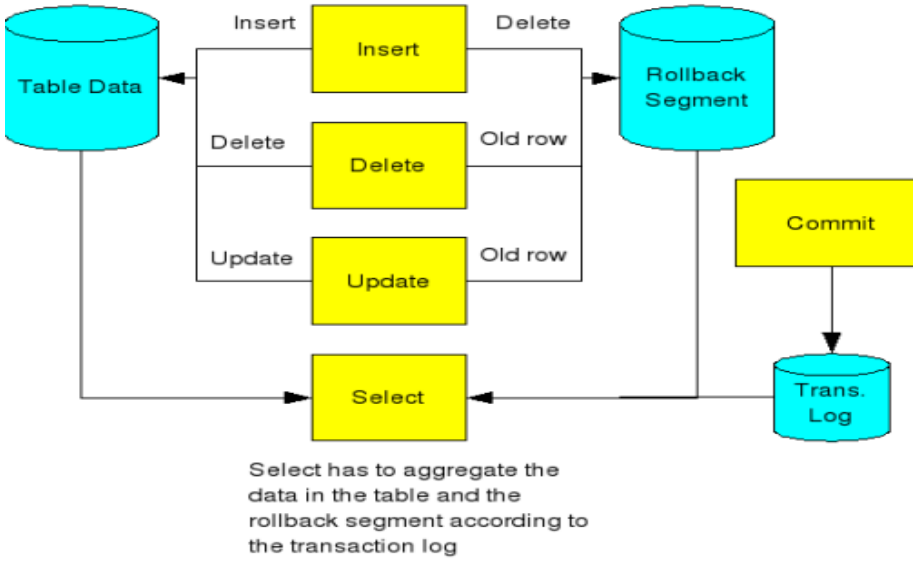


Figura 5.6: Desenho do Rollback Segment da Oracle

O método do MVCC é menos intuitivo mas ainda assim com grande performance. Tem a vantagem de não manter ficheiros de segmentos em disco durante transações de longa duração, em vez disso consome apenas espaço da tabela fazendo uso das identificações dos tuplos e das transações para determinar a sua visibilidade.

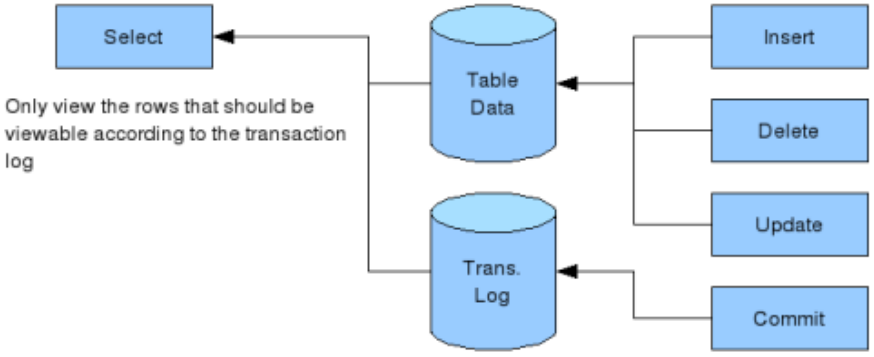


Figura 5.7: Desenho do MVCC

Sobre os níveis de isolamento, ambos suportam os mesmo níveis, *Read Com-*

mitted e *SERIALIZABLE*. O *PostgreSQL* ainda tem o *Repeateble Read*, a *Oracle*® tem um nível semelhante se utilizarmos o comando SET TRANSACTION READ ONLY.

Sobre consistência, ambos permitem fazer a verificação das restrições depois de cada operação ou apenas depois da transação fazer *commit*.

Sobre a granularidade, ambos permitem mudar a granularidade do lock, mas por defeito no *Oracle*® é feito nas linhas enquanto que no *PostgreSQL* é feito nas tabelas. Os modos de lock são semelhantes e têm o mesmo objetivo

Capítulo 6

Suporte para Bases de Dados Distribuídas

Até meados de 2008, a equipa de desenvolvimento do *PostgreSQL* evitava que a replicação fizesse parte do core do SGBD. [63], mas esta foco mudou e, atualmente, existe no core do *PostgreSQL* : [64]

- Streaming Replication está disponível desde a versão 9.0 e oferece sincronização assíncrona para um ou mais servidores de standby.
- Log Shipping é uma solução de alta performance que "replica"um cluster da base de dados para um arquivo ou conjunto de servidores de standby. É fácil de configurar e tem um *overhead* muito baixo.

6.1 Replicação

Arquivamento contínuo pode ser usado para criar uma configuração de *cluster* de alta disponibilidade. [62]

O servidor primário opera continuamente em modo de arquivamento (*continous recovery mode*), enquanto que cada servidor de *standby* opera continuamente em modo de recuperação (*continous recovery mode*) lendo os ficheiros WAL do servidor primário. Esta configuração tem um baixo impacto na performance do servidor primário.

6.1.1 Replicação por Stream

Em replicação por *stream* o servidor de standby liga-se ao servidor primário, que lhe transmite registos *WAL* assim que são gerados, sem esperar que estes sejam filtrados. A replicação *stream* por *default* é assíncrona, como consequência existe um pequeno *delay* entre o *commit* no servidor primário e o tempo em que se tornam visíveis as alterações no servidor de standby. Com replicação por *stream*, não é necessário especificar o *archive_timeout*.

Para usar replicação deste tipo é necessário definir um servidor standby *file-based log-shipping*, depois, o passo que transforma o servidor em *streaming replication standby* é, no ficheiro *recovery.conf*, definir um apontador para o servidor primário.

6.1.2 Replicação em Cascata

A replicação em cascata permite ao servidor standby aceitar conexões replicadas e registos *stream WAL* para que outros servidores *standby* atuem como repetidores. Isto pode ser usado para reduzir o número de ligações directas para o servidor principal e ainda minimiza os *overheads* de largura entre as ligações dos servidores para com o servidor primário.

Quando um servidor *standby* atua como recetor e remetente, é conhecido como *cascading standby*. Um *cascading standby* envia não apenas os registos *WAL* que recebe do primário mas também aqueles que são restaurados do arquivo.

A replicação em cascata é de momento assíncrona, mesmo que se defina replicação síncrona, estas definições não terão efeito.

6.1.3 Replicação Síncrona

Em *PostgreSQL* , por defeito, se um servidor primário falhar quando uma transação que já fez commit mas ainda não tinha sido replicada para o servidor de *standby*, existe perda de informação. A quantidade de dados perdida é proporcional ao tempo de replicação no momento da falha.

Replicação síncrona oferece a capacidade de confirmar que todas as alterações feitas por uma transição são transferidas para um servidor de síncrono. Desta forma é estendido o nível \emph{standard} de durabilidade oferecido quando uma transação faz *commit*. Este nível de proteção é também chamado de *2-save replication*.

Transações "*read-only*"e transações de "*roolbacks*"não precisam de esperar pela resposta dos servidores de standby.

6.2 Ligações Remotas

6.2.1 Bases de dados Homogéneas

Até à versão 9.2 do *PostgreSQL* o estabelecimento de ligações remotas a outros servidores *PostgreSQL* era efetuado recorrendo ao módulo *dblink*[59] que faz parte da instalação do *PostgreSQL* .

Para estabelecer uma ligação é usado o comando *dblink_connect* cuja sintaxe pode ser visualizada na figura 6.1. Para executar uma query numa base de dados remota é usada a expressão *dblink* cuja sintaxe pode ser visualizada na figura 6.2. Apesar de funcionais, estes mecanismos não oferecem de todo transparência ao utilizador, sendo até a sintaxe bastante pesada.

dblink_connect(text connname, text connstr) returns text
em que:
connname - nome da ligação;
connstr - string com informações da ligação (no formato definido em *libpq-style*[60]).

Figura 6.1: Sintaxe comando *dblink_connect*.

SELECT *
FROM dblink(text connname, text sql [, bool fail_on_error]) returns setof record

em que:
connname - nome da ligação;
sql - pergunta *SQL* ;
fail_on_error - flag para indicar erro localmente caso ocorra um erro na base de dados remota.

Figura 6.2: Exemplo pergunta com *dblink*.

Na versão 9.3 do *PostgreSQL* foi introduzido um novo módulo para substituir o *dblink*, que pretende resolver este problema. O módulo *postgres_fdw*[61] oferece um *Foreign-Data Wrapper*[1]que pode ser usado para aceder a dados em servidores *PostgreSQL* externos. Este módulo veio assim trazer um maior grau de transparência ao utilizador.

Para efetuar um acesso remoto utilizando o *postgres_fdw* é necessário efetuar os seguintes passos:

1. Instalar a extensão *postgres_dfw* usando o comando *CREATE EXTENSION* (ver figura 6.3);
2. Criar um objeto servidor externo usando o comando *CREATE SERVER* (ver figura 6.4), para representar cada base de dados remota, especificando dados sobre a ligação;

[1]Abstração que permite aceder a dados externos ao *PostgreSQL* utilizando perguntas *SQL* [57].

3. Criar um mapeamento de utilizador usando o comando *CREATE USER MAPPING* (ver figura 6.5), para cada utilizador que pretenda aceder a um servidor externo;

4. Criar uma tabela externa usando o comando usando o comando *CREATE FOREIGN TABLE* (ver figura 6.6), para cada tabela que se deseja aceder. É possível redefinir os nomes de atributos da tabela remota que serão apresentados na base de dados local.

Após estes passos, é possível executar qualquer pergunta colocando a tabela externa na cláusula *FROM*.

```
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
    [ WITH ] [ SCHEMA schema_name ]
        [ VERSION version ]
        [ FROM old_version ]
```

Figura 6.3: Comando *CREATE EXTENSION* .

```
CREATE SERVER server_name [ TYPE 'server_type' ] [ VERSION 'server_version' ]
    FOREIGN DATA WRAPPER fdw_name
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

Figura 6.4: Comando *CREATE SERVER*.

```
CREATE USER MAPPING FOR  user_name | USER | CURRENT_USER | PUBLIC
    SERVER server_name
    [ OPTIONS ( option 'value' [ , ... ] ) ]
```

Figura 6.5: Comando *CREATE USER MAPPING*.

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name ( [
    column_name data_type [ OPTIONS ( option 'value' [, ... ] ) ] [ COLLATE
collation ] [ column_constraint [ ... ] ]
    [, ... ]
] )
SERVER server_name
[ OPTIONS ( option 'value' [, ... ] ) ]

em que column_constraint é:

[ CONSTRAINT constraint_name ]
NOT NULL |
NULL |
DEFAULT default_expr
```

Figura 6.6: Comando *CREATE FOREIGN TABLE*.

6.2.2 Bases de dados Heterogéneas

Nativamente o *PostgreSQL* não oferece nenhum tipo de suporte a bases de dados heterogéneas. No entanto, existem algumas soluções disponíveis:

PostgreSQL Module ODBC Link[56] - Desenvolvido pela empresa *Cybertec*[2]. Este módulo permite estabelecer ligações com qualquer sistema de base de dados que suporte o padrão de acesso aos dados *ODBC* (Open Database Connectivity);

PostgreSQL Foreign Data Wrapper for Oracle[58] - Extensão para o *PostgreSQL* que implementa um *Foreign Data Wrapper* para acesso simples e eficiente a uma base de dados *Oracle*® .

Apesar destas soluções oferecerem este tipo de suporte e já oferecerem um certo grau de transparência para o utilizador, não torna por completo transparente o acesso a outras bases de dados. Isto porque não é suportada a definição de *Aliases* como no *Oracle*® .

6.3 Atomicidade - Two-phase Commit

O *PostgreSQL* suporta *two-phase commit* utilizando o comando PREPARE TRANSACTION.

[2] `http://www.cybertec.at/en/`

```
PREPARE TRANSACTION transaction_id
```

PREPARE TRANSACTION é uma extensão do *PostgreSQL* , criado com a intenção de ser utilizado por sistemas de gestão de transações externos ao *PostgreSQL* , para se poderem executar transações globais atómicas através de múltiplas base de dados ou outros recursos transacionais, não devendo portanto ser utilizado por sessões interativas ou aplicações.[75]

Quando executado, a transação deixa de estar associada à sessão corrente, é totalmente colocada em disco e a probabilidade de ser completada sem erros sobe bastante, mesmo que existam *crashes* antes dela completar.

Para completar a transação utiliza-se o comando COMMIT PREPARED ou ROLLBACK PREPARED para abortar. Estes comandos podem ser executados em qualquer sessão, não apenas na sessão de onde a transação é originária.

Capítulo 7

Outras Características

7.1 TeamPostgreSQL

TeamPostgreSQL[24][25] é uma aplicação web que permite aceder à base de dados de uma organização através da Internet (ou intranet). Também permite ser utilizada localmente.

Figura 7.1: Interface *TeamPostgreSQL*.

Esta aplicação é bastante útil uma vez que permite navegar pelos dados de uma base de dados de uma forma simples. Permite uma boa visualização dos dados e várias operações sobre os mesmos. Algumas das funcionalidades chave

são:

- Acesso direto a tuplos referenciados por chaves externas (e vice-versa), clicando nas mesmas;
- Edição de tuplos;
- Suporte a operações diretamente sobre os dados (por exemplo, ordenações);
- Execução de *snippets* SQL e partilha dos mesmos com a equipa;
- Interface gráfica para o *pgdump*. Permite criar scripts *SQL*.

7.2 Suporte *XML*

O *PostgreSQL* oferece as seguintes funcionalidades, em que é utilizada a tecnologia *XML*:

- Tipo de dados *XML*;
- Funções *XML*

Cada uma destas funcionalidades serão apresentadas detalhadamente de seguida.

7.2.1 Tipos de dados *XML*

O *PostgreSQL* implementa o tipo de dados *XML*[27] para armazenar dados *XML*. A utilização deste tipo é vantajosa uma vez que o *PostgreSQL* verifica a sintaxe do documento e oferece várias funções para manipulação deste tipo, que serão abordadas na secção 7.2.2.

Por omissão, o instalador do *PostgreSQL* não inclui suporte para o tipo de dados *XML*. É necessário fornecer o comando ***configure –with-libxml***.

É possível guardar documentos *XML* bem formados de acordo com o standard *XML* ou apenas fragmentos. Para saber se um documento armazenado é um documento completo ou apenas um fragmento pode-se utilizar a seguinte expressão:

xmlvalue IS DOCUMENT

em que *xmlvalue* é do tipo *XML*.

Figura 7.2: Determinar se um valor do tipo *XML* é um documento completo.

Para criar um valor do tipo *XML* usa-se o comando:

```
XMLPARSE (  DOCUMENT | CONTENT  value)

Ex:
XMLPARSE ( DOCUMENT
   '<?xml version="1.0"?><book><title>Manual</title>
   <chapter>...</chapter></book>'
)
```

Figura 7.3: Criação de um valor do tipo *XML*.

No *PostgreSQL* a única forma de comparar valores do tipo *XML* é convertendo-o os para texto. Não é implementado nenhum mecanismo de comparação para este tipo de dados o que impossibilita a existência de índices sobre atributos deste tipo. O *Oracle*® suporta a criação de índices sobre atributos do tipo *XML* (denominados por *XMLType*)[26].

7.2.2 Funções *XML*

O *PostgreSQL* implementa várias funções para manipulação do tipo de dados *XML*[28].

Vamos salientar as funções mais interessantes.

Para verificar se existe um determinado elemento, através de uma *query XPATH* num documento *XML* pode-se usar a seguinte função:

```
XMLEXISTS(text PASSING [BY REF] xml [BY REF])
Ex:
SELECT xmlexists('//town[text() = "Toronto"]' PASSING BY REF
   '<towns><town>Toronto</town><town>Ottawa</town></towns>'
);
```

Figura 7.4: Função *XMLEXISTS* .

Para processar um documento *XML* o *PostgreSQL* implementa a função *xpath* que, dada uma *query XPATH*, avalia a *query* e devolve o conjunto de nós resultantes do documento. A sintaxe é a seguinte:

```
xpath(xpath, xml [, nsarray])
Ex:
SELECT xpath('/my:a/text()',
   '<my:a xmlns:my="http://example.com">test</my:a>',
        ARRAY[ARRAY['my', 'http://example.com']]);
Resultado:
xpath
------
test
(1 tuplo)
```

Figura 7.5: Função *xpath* .

O *PostgreSQL* também permite converter os dados de uma relação para um documento *XML* com a função *table_to_xml*. Este comando tem a seguinte sintaxe:

table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)
Em que:

tbl - Nome da relação;

nulls - Considerar valores *nulls*, ou não;

tableforest - Permite especificar o formato do documento *XML*;

targetns - Permite especificar o *namespace* em que será inserido o resultado.

Figura 7.6: Função *table_to_xml* .

Além desta última função, *PostgreSQL* fornece uma função idêntica para exportar o *schema* de uma relação (***table_to_xmlschema***) em que os parâmetros são os mesmos.

7.3 Web Semântica

Atualmente o *PostgreSQL* não oferece suporte no contexto de Web Semântica. Contudo, foi publicado um artigo [14] que propõe algoritmos para integrar su-

porte à Web semântica no *PostgreSQL* . De entre vários aspetos, a proposta inclui:

- Adicionar suporte a perguntas sobre dados no formato *RDF*[16] (Resource Description Framework);
- Permitir inferência lógica de proposições com base em regras existentes e ontologias recorrendo a *triggers* e ao sistema de regras (*rule system*) do *PostgreSQL* ;

Sendo que nesta proposta, um dos objetivos é não introduzir alterações ao *SQL* suportado pelo *PostgreSQL* .

Existem algumas ferramentas para a *Web Semântica* não oficiais, para o *PostgreSQL* , tais como o *Sparqlify*[15]. O *Sparqlify* é uma aplicação que permite definir vistas *RDF* numa base de dados relacionais e permite interrogar a base de dados usando *SPARQL*[1][17];

7.4 Ferramentas

Nativamente, o *PostgreSQL* oferece uma ferramenta para administração do sistema de base de dados, o *pgAdmin*[29]. Esta ferramenta é bastante intuitiva e permite gerir todo o sistema.

7.5 Segurança

7.5.1 Permissões de Acesso

O *PostgreSQL* gere as permissões de acesso através de um conceito denominado por *roles*[32]. No *PostgreSQL* uma *role* denota um utilizador ou a um grupo de utilizadores e, detém objetos da base de dados (por exemplo, relações) e pode atribuir privilégios de objetos que detém a outras *roles*.

A criação de uma *role* é efetuada com a seguinte expressão:

[1]Linguagem para efetuar interrogações sobre *RDF*

```
CREATE ROLE name [ [ WITH ] option [ ... ] ]

em que option pode ser:

    SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
```

Figura 7.7: Criação de uma *role* .

Funções e *triggers* possibilitam a inserção de código malicioso por parte de um utilizador. A única forma de prevenir estes ataques é não permitir a definição de novas funções ou *triggers*, mantendo esses privilégios para um grupo de utilizadores muito restrito e de confiança.

Contudo, no caso das funções são necessários mecanismos de segurança ainda mais fortes. Isto porque as funções são executadas do lado do servidor e as permissões de execução correspondem às permissões dadas pelo sistema operativo ao *daemon* do servidor da base de dados. Se a linguagem de programação não possuir mecanismos de controlo de acessos à memória, torna-se possível alterar as estruturas de dados usadas pelo servidor. Dado a gravidade do problema, quando são utilizadas este tipo de linguagens, o *PostgreSQL* apenas permite *superusers*[2]criarem funções.

[2]Um *superuser* é um utilizador especial que tem controlo total sobre o servidor da base de dados.

7.5.2 Métodos de Autenticação

No que toca a autenticação, o *PostgreSQL* é um sistema bastante rico e oferece suporte para vários tipos de autenticação:

Trust Authentication - Todos os utilizadores que estabelecem uma ligação com o servidor são autenticados;

Password Authentication - Cada utilizador autentica-se usando a sua *password*. É possível especificar se a *password* passa em claro no canal o respetivo *hash* (é utilizado o *md5*).

GSSAPI Authentication - Autenticação utilizando o *standard GSSAPI*[34] (Generic Security Services Application Program Interface) para autenticação segura definido no *RFC 2743*[33];

SSPI Authentication - SSPI[35] (Security Support Provider Interface) é uma tecnologia *Windows* para autenticação segura com *single sign-on*[3][36];

Ident Authentication - O nome de utilizador do sistema operativo do cliente é obtido de um servidor *ident* e é utilizado para efetuar a autenticação. Apenas é suportado em ligações *TCP/IP*;

Peer Authentication - O nome de utilizador do sistema operativo do cliente é obtido diretamente a partir do *kernel* e é utilizado na autenticação. Apenas é suportado em ligações locais;

The peer authentication method works by obtaining the client's operating system user name from the kernel and using it as the allowed database user name

LDAP Authentication - Semelhante à autenticação com *passwords* em que é utilizado o protocolo LDAP[37] (Lightweight Directory Access Protocol) para validar a *password* do cliente;

RADIUS Authentication - Semelhante à autenticação com *passwords* em que é utilizado o protocolo RADIUS[38] (Remote Authentication Dial In User Service) para validar a *password* do cliente;

Certificate Authentication - Utilização de certificados SSL[39] (Secure Sockets Layer) para autenticar o cliente. Apenas está disponível em ligações *SSL*;

PAM Authentication - Semelhante à autenticação com *passwords* em que é utilizado o mecanismo PAM[40] (Pluggable Authentication Modules) para autenticar o cliente.

[3]Propriedade de controlo de acessos em que um utilizador se autentica uma vez só para todos os sistemas.

7.5.3 SSL - Secure Sockets Layer

O *PostgreSQL* oferece suporte nativo a ligações utilizando SSL[31][39]. Para tal, é necessário que esteja instalado o *OpenSSL*[4][42] quer no cliente quer no servidor e o *PostgreSQL* tem de ser compilado definindo o parâmetro *ssl*[41] como *true* no ficheiro de configuração *postgresql.conf*.

[4]Suíte *open-source* de ferramentas para *SSL/TLS*

Bibliografia

[1] Manual PostgreSQL 9.3 - Catálogos do sistema: http://www.postgresql.org/docs/9.3/static/catalogs.html

[2] Manual PostgreSQL 9.3 - Protocolo Frontend/Backend: http://www.postgresql.org/docs/9.3/static/protocol.html

[3] Bison - GNU parser generator: http://www.gnu.org/software/bison/

[4] flex - The Fast Lexical Analyzer: http://flex.sourceforge.net/

[5] Regras de equivalência álgebra Relacional - PostgreSQL: http://www.postgresql.org/message-id/attachment/32495/equivalenceRules.pdf

[6] PostgreSQL 9.3 Join limit - geqo threshold: http://www.postgresql.org/docs/9.3/static/runtime-config-query.html#GUC-GEQO-THRESHOLD

[7] PostgreSQL 9.3 Server Configuration - Query Planning: http://www.postgresql.org/docs/9.3/static/runtime-config-query.html

[8] PostgreSQL 9.3 Genetic Query Optimizer (GEQO): http://www.postgresql.org/docs/9.3/static/geqo.html

[9] PostgreSQL 9.3 Operator Classes and Operator Families: http://www.postgresql.org/docs/9.3/static/indexes-opclass.html

[10] Inside the PostgreSQL Query Optimizer: http://www.neilconway.org/talks/optimizer/optimizer.pdf

[11] IBM System R: http://en.wikipedia.org/wiki/IBM_System_R

[12] Edge Recombination Crossover: http://en.wikipedia.org/wiki/Edge_recombination_operator

[13] Whitley, Darrell; Timothy Starkweather, D'Ann Fuquay (1989). "Scheduling problems and traveling salesman: The genetic edge recombination operator". International Conference on Genetic Algorithms. pp. 133–140. ISBN 1-55860-066-3

[14] Levshin, D. V. & Markov, A. S. (2009). "Algorithms for integrating PostgreSQL with the semantic web". Programming and Computer Software 35 (3). pp. 136-144

[15] Sparqlify: http://aksw.org/Projects/Sparqlify.html

[16] Resource Description Framework (RDF): http://www.w3.org/RDF/

[17] SPARQL Protocol and RDF Query Language: http://www.w3.org/TR/rdf-sparql-query/

[18] PostgreSQL Wiki - Parallel Query Execution: https://wiki.postgresql.org/wiki/Parallel_Query_Execution

[19] PostgreSQL 9.3 Server Configuration - Asynchronous Behavior: http://www.postgresql.org/docs/9.2/static/runtime-config-resource.html#RUNTIME-CONFIG-RESOURCE-ASYNC-BEHAVIOR

[20] PostgreSQL Manual - Monitoring Database Activity: http://www.postgresql.org/docs/9.3/static/monitoring-stats.html

[21] PostgreSQL Wiki - Optimizer Hints Discussion: http://wiki.postgresql.org/wiki/OptimizerHintsDiscussion

[22] *Oracle*® Database Performance Tuning Guide - Using SQL Plan Management: http://docs.oracle.com/cd/B28359_01/server.111/b28274/optplanmgmt.htm#PFGRF00701

[23] *Oracle*® Database Reference B14237-04 - Logical Database Limits: http://docs.oracle.com/cd/B19306_01/server.102/b14237/limits003.htm#i288032

[24] TeamPostgreSQL Announce: http://www.postgresql.org/about/news/1396/

[25] TeamPostgreSQL: http://www.teampostgresql.com/

[26] *Oracle*® - Indexing XMLType Columns: http://docs.oracle.com/cd/B10501_01/appdev.920/a96620/xdb04cre.htm#1031630

[27] *PostgreSQL* - XML Type: http://www.postgresql.org/docs/9.3/static/datatype-xml.html

[28] *PostgreSQL* - XML Functions: http://www.postgresql.org/docs/current/static/functions-xml.html

[29] pgAdmin: http://www.pgadmin.org/

[30] *PostgreSQL* - Authentication Methods http://www.postgresql.org/docs/9.3/static/client-authentication.html

[31] *PostgreSQL* - SSL-TCP http://www.postgresql.org/docs/9.3/static/ssl-tcp.html

[32] *PostgreSQL* - Database Roles http://www.postgresql.org/docs/9.3/static/user-manag.html

[33] Generic Security Service Application Program Interface Version 2, Update 1 RFC 2743 https://datatracker.ietf.org/doc/rfc2743/?include_text=1

[34] Generic Security Services Application Program Interface http://en.wikipedia.org/wiki/Generic_Security_Services_Application_Program_Interface

[35] Security Support Provider Interface http://msdn.microsoft.com/en-us/library/windows/desktop/aa380493(v=vs.85).aspx

[36] Single sign-on http://en.wikipedia.org/wiki/Single_sign-on

[37] Lightweight Directory Access Protocol http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol

[38] Remote Authentication Dial In User Service http://en.wikipedia.org/wiki/RADIUS

[39] Secure Sockets Layer http://en.wikipedia.org/wiki/Secure_Sockets_Layer

[40] Pluggable Authentication Module http://en.wikipedia.org/wiki/Pluggable_authentication_module

[41] Secure Sockets Layer http://www.postgresql.org/docs/9.3/static/runtime-config-connection.html#GUC-SSL

[42] Secure Sockets Layer http://www.openssl.org/

[43] *Oracle*® Acess Paths http://docs.oracle.com/cd/B19306_01/server.102/b14211/optimops.htm#i82080

[44] Managing Clusters http://docs.oracle.com/cd/B19306_01/server.102/b14231/clustrs.htm#sthref2926

[45] *PostgreSQL* - Index Types: http://www.postgresql.org/docs/9.3/static/indexes-types.html

[46] Wikipedia - GiST: http://www.postgresql.org/docs/9.3/static/indexes-types.html

[47] *PostgreSQL* - pg_database: http://www.postgresql.org/docs/9.3/static/catalog-pg-database.html

[48] *PostgreSQL* - TOAST: http://www.postgresql.org/docs/9.3/static/catalog-pg-database.html

[49] *PostgreSQL* - CLUSTER: http://www.postgresql.org/docs/9.3/static/sql-cluster.html

[50] *PostgreSQL* - VACUUM: http://www.postgresql.org/docs/9.3/static/sql-vacuum.html

[51] *PostgreSQL* - SET CONSTRAINTS: http://www.postgresql.org/docs/9.3/static/sql-set-constraints.html

[52] *PostgreSQL* - Partitioned Tables and Indexes: http://docs.oracle.com/cd/B19306_01/server.102/b14220/partconc.htm

[53] Oracle vs PostgreSQL: http://database-management-systems.findthebest.com/compare/36-43/Oracle-vs-PostgreSQL

[54] *PostgreSQL* - pg_buffercache : http://www.postgresql.org/docs/9.3/static/pgbuffercache.html

[55] *PostgreSQL* - Database Page Layout: http://www.postgresql.org/docs/8.0/static/storage-page-layout.html

[56] *PostgreSQL* Module ODBC Link: http://www.cybertec.at/download/odbc_link/2010_03_16_odbc_link.pdf

[57] *PostgreSQL* Foreign Data Wrapper: http://www.postgresql.org/docs/current/static/ddl-foreign-data.html

[58] PostgreSQL Foreign Data Wrapper for Oracle: http://laurenz.github.io/oracle_fdw/

[59] PostgreSQL DB-link Module: http://www.postgresql.org/docs/9.3/static/dblink.html

[60] Connection Strings: http://www.postgresql.org/docs/current/static/libpq-connect.html#LIBPQ-CONNSTRING

[61] Módulo postgres-fdw: http://www.postgresql.org/docs/current/static/postgres-fdw.html

[62] Log-Shipping Standby Servers: http://www.postgresql.org/docs/9.3/static/warm-standby.html

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

[64] https://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling#Features_

[65] PostgreSQL - Explicit Locking: http://www.postgresql.org/docs/current/static/explicit-locking.html#LOCKING-TABLES

[66] PostgreSQL - Write Ahead Log: http://www.postgresql.org/docs/current/static/wal-intro.html

[67] DSHL - Write Ahead Log + understanding Postgresql.conf: checkpoint_segments, checkpoint_Timeout, checkpoint_warnings: http://www.depesz.com/2011/07/14/write-ahead-log-understanding-postgresql-conf-checkpoint_segments-checkpoint_timeout-checkpoint_warning/

[68] PostgreSQL - Asynchronous Commit: http://www.postgresql.org/docs/current/static/wal-async-commit.html

[69] PostgreSQL - Transaction Isolation: http://www.postgresql.org/docs/current/static/transaction-iso.html

[70] Heroku - PostgreSQL Concurrency With MVCC: https://devcenter.heroku.com/articles/postgresql-concurrency

[71] PostgreSQL - MVCC: http://www.postgresql.org/docs/9.3/static/mvcc-intro.html

[72] PostgreSQL - SET CONSTRAINTS: http://www.postgresql.org/docs/9.3/static/sql-set-constraints.html

[73] PostgreSQL - Transactions: http://www.postgresql.org/docs/9.3/static/tutorial-transactions.html

[74] PostgreSQL - Lock: http://www.postgresql.org/docs/current/static/sql-lock.html

[75] PostgreSQL - Prepare Transaction: http://www.postgresql.org/docs/9.3/static/sql-prepare-transaction.html

[76] PostgreSQL's Transaction Model: http://www.packtpub.com/article/transaction-model-of-postgresql

[77] *PostgreSQL* History: http://www.postgresql.org/about/history/

[78] PostgreSQL's Wikipedia: http://pt.wikipedia.org/wiki/PostgreSQL