Sistemas de Bases de Dados

Estudo sobre o sistema de bases de dados *PostgreSQL*

Docente:
José Júlio Alferes

Alunos (*Grupo 13*):
José Bastidas 41897
Nuno Zuzarte 42020
Nuno Martins 42064

1. Introdução

O presente relatório foi desenvolvido no âmbito da unidade curricular de Sistemas de Bases de Dados e tem como objetivo a aquisição de conhecimentos sobre as especificações e mecanismos implementados pelo sistema de gestão de bases de dados *PostgreSQL*.

Ao longo do relatório serão fornecidas algumas comparações entre o sistema a ser estudado e o sistema *Oracle 11g* - utilizado nas aulas práticas da cadeira.

Este relatório encontra-se dividido em diversos tópicos principais, correspondendo a cada um um *capítulo* de matéria abordada. Iniciaremos o nosso estudo pelos mecanismos de armazenamento e estrutura de ficheiros do *Postgres*, passando de seguida para as estruturas de indexação e *hashing* utilizadas pelo mesmo. De seguida são abordados os tópicos de processamento e otimização de perguntas, gestão de transações e controlo de concorrência e bases de dados distribuídas.

O *PostgreSQL* é um sistema de bases de dados *open source* altamente costumizável e configurável que existe no mercado há mais de quinze anos, tendo já dado provas evidentes de fiabilidade, integridade de dados e correção. Corre sobre todos os principais sistemas operativos e é totalmente compatível com as propriedades *ACID*. Inclui a maior parte dos tipos de dados do *SQL* e possui recursos sofisticados tais como *Multi-Version Concurrency Control (MVCC)*, *point in time recovery*, *tablespaces*, *asynchronous replication*, *nested transactions (savepoints)*, *hot backups*, um sofisticado *query planner/optimizer* e *Write-Ahead Logging* (*WAL*) para tolerância a falhas. O *PostgreSQL* possui também diversas bibliotecas de suporte a várias linguagens compiladas ou interpretadas (*Java* (*JDBC*), *ODBC*, *Perl*, *Python*, *Ruby*, *C*, *C++*, *PHP*, *etc*...).

2. Armazenamento e estrutura de ficheiros

2.1. Sistema de gestão de *buffer*

O *PostgreSQL* possui, à semelhança do sistema *Oracle*, um gestor de *buffer* interno. Significa isto que é o próprio sistema a fazer a gestão da memória para onde conteúdo da base de dados é carregado, não deixando tal gestão - pelo menos não totalmente - para o sistema operativo.

Como algoritmo de *chaching*, o sistema utiliza um método denominado *clock-sweep*, que fornece uma aproximação à politica *LRU*. O mecanismo consiste em representar o *buffer* como um simples *array* de blocos.

Cada vez que existe necessidade de aceder a um bloco, é incrementado um contador que representa o número de acessos ao mesmo. Blocos referenciados por pelo menos um cliente (ou transação) são referidos como *pinned* e não podem ser reciclados caso uma nova alocação seja necessária. Quando novos dados são inseridos num bloco, este passa a estando *dirty* - tal significa que, quando for necessário desalocá-lo, o seu conteúdo necessita de ser escrito em disco primeiro.

Mudanças típicas no estado de blocos

```
(unpinned, not dirty) -> read -> (pinned, not dirty) -> update -> (pinned, dirty) -> transaction complete -> (unpinned, dirty) -> checkpoint -> (unpinned, not dirty)
```

```
(unpinned, not dirty) -> insert -> (pinned, dirty) -> background writer write -> (pinned, not dirty) -> transaction complete -> (unpinned, not dirty)
```

Inicialmente existe uma lista de blocos livres no *buffer*. No entanto, com o crescente uso da base de dados, rapidamente deixam de existir espaços livres. Por esta razão, existe um processo de *vacuum* que vai realocando blocos de modo a que a *cache* não se esgote. A decisão sobre qual o próximo bloco a ser realocado é delegada ao mecanismo de *clock-sweep* previamente referido.

Quando é necessário aceder a um novo bloco em disco, é chamada a operação *BufferAlloc*. Se houver algum bloco na lista de blocos livres, um desses blocos é disponibilizado. No entanto, em muitos dos casos não existem blocos livres, pelo que é feito um *scan* à *cache* do *buffer* começando na posição *LRU clock-sweep*, à procura do primeiro que não esteja no estado *pinned* (a ser usado). Se este estiver no estado *dirty*, é utilizado.

2.2. Sistema de ficheiros

Ao contrário do *Oracle* que implementa o seu próprio sistema de ficheiros - de forma a fornecer maior segurança, disponibilidade e robustez, assim como melhores transações e um maior nível de escalabilidade - o *PostgreSQL* utiliza o sistema de ficheiros do sistema operativo sobre o qual está a ser executado. As configurações e os ficheiros de dados usados por um

database cluster (coleção de bases de dados armazenadas numa área do disco), no *PostgreSQL*, são armazenados na diretoria de dados do *cluster*, normalmente conhecida como *DATA*. A organização dos ficheiros baseia-se em subdiretorias, onde corresponde a cada tabela uma subdiretoria, na qual se se encontram vários ficheiros correspondentes à mesma.

A diretoria *DATA* contem várias subdiretorias e ficheiros de controlo como apresentado na seguinte tabela:

Tabela de conteúdos de *DATA*

Item	Descrição
`PG_VERSION`	Ficheiro que contém o número da maior versão do PostgreSQL
`base`	Subdiretoria que contém as subdiretorias por base de dados
`global`	Subdiretoria que contém tabelas *cluster-wide*, como a `pg_database`
`pg_clog`	Subdiretoria que contém dados de estado de *commit* de transações
`pg_multixact`	Subdiretoria que contém dados de estado de multitransações (utilizados para *shared row locks*)
`pg_notify`	Subdiretoria que contém dados de estado *LISTEN/NOTIFY*
`pg_serial`	Subdiretoria que contém informação sobre transações *committed serializable*
`pg_snapshots`	Subdiretoria que contém *snapshots* exportados
`pg_stat_tmp`	Subdiretoria que contém ficheiros temporários do subsistema de estatísticas
`pg_subtrans`	Subdiretoria que contém dados de estado de subtransações
`pg_tblspc`	Subdiretoria que contém *links* simbólicos para *tablespaces*
`pg_twophase`	Subdiretoria que contém ficheiros de estado para transações preparadas
`pg_xlog`	Subdiretoria que contém ficheiros *WAL*
`postmaster.opts`	Ficheiro que regista as opções *command-line* com que o servidor começou
`postmaster.pid`	Ficheiro de *lock* que regista o atual *postmaster process ID (PID)*, caminho para a diretoria de dados do *cluster*, *timestamp* inicial do *postmaster*, número de porta, etc.

Adicionalmente, os ficheiros de configuração do *cluster* (*postgresql.conf, pg_hba.conf* e *pg_ident.conf*) são também armazenados na mesma diretoria.

Cada base de dados tem, por sua vez, uma subdiretoria dentro de *DATA/base* onde estão localizados, por omissão, os seus ficheiros, incluindo os catálogos do sistema.

Em resumo, as bases de dados no *PostgreSQL* são maioritariamente compostas por várias tabelas, onde cada tabela tem uma subdiretoria associada. Nestas subdiretorias, por sua vez, existem vários ficheiros com um tamanho máximo de 1*GB* tratados como sequências de blocos de 8*KB*.

2.3. Mecanismo de partições

O *PostgreSQL* fornece suporte ao particionamento de tabelas através do mecanismo de herança. Para que o particionamento de tabelas seja possível, terá de existir sempre uma

tabela *pai* (normalmente vazia), que representa o conjunto de dados num todo. Várias partições são criadas como *filhos* da tabela principal.

O sistema suportada duas formas distintas de particionamento: *range partitioning*, que consiste no particionamento da tabela por *ranges* definidos por uma ou mais colunas chave (por exemplo, um intervalo de idades) sem sobreposição de valores em diferentes partições; e *list partitioning,* que consiste no particionamento de uma tabela recorrendo a atributos chave (como por exemplo, a divisão de uma tabela `cadeiras` por curso).

Exemplo de criação de partições

```
-- Criar tabela pai sobre a qual as outras tabelas herdarão
CREATE TABLE cadeiras (
    nome NOT NULL,
    curso NOT NULL
);

-- Criar tabelas filho
CREATE TABLE cadeiras_miei (
    CHECK (curso = 'MIEI')
) INHERITS (cadeiras);
CREATE TABLE cadeiras_lei (
    CHECK (curso = 'LEI')
) INHERITS (cadeiras);
-- (...)

-- Criação de índices para as tabelas filho
CREATE INDEX ON cadeiras_miei (nome);
CREATE INDEX ON cadeiras_lei (nome);
-- (...)
```

Opcionalmente poderão ser criados *triggers*, para que os dados sejam redirecionados para as respetivas partições aquando de uma inserção na tabela *pai*, de modo a providenciar transparência de particionamento - é de notar que este comportamento não é nunca automático, ao contrário das partições no sistema *Oracle*.

Deve-se ter em conta de que não existe uma forma automática de verificar que as restrições *check* são mutuamente exclusivas, pelo que cabe ao gestor da base de dados ter essa preocupação.

Por fim, é de notar que existem também métodos de otimização de *queries* em partições e métodos alternativos para a criação de partições (*split* e *merge partitioning*), no entanto, mais uma vez, os métodos de particionamento suportados pelo *Oracle* são bastante superiores.

2.4. *Multitable clustering*

Ao contrário do *Oracle*, o *PostgreSQL* não fornece suporte para *multitable clustering*, possivelmente devido à forma como armazena os dados utilizando o sistema operativo.

2.5. *Page layout* e registos de tamanho variável

O sistema *PostgreSQL* utiliza páginas de tamanho fixo para armazenamento dos seus registos, que têm normalmente 8*KB*, não permitindo que um tuplo ocupe múltiplas páginas. Deste modo, não é possível armazenar campos de dados muito grandes. Como tal, para ultrapassar esta limitação, o sistema recorre à compressão e particionamento deste tipo de atributos em várias linhas físicas, de forma transparente para o utilizador - esta técnica é conhecida como *TOAST* (*The Oversized-Attribute Storage Technique*).

Cada tabela e índice é armazenado como um *array* de páginas de tamanho fixo. O sistema utiliza uma organização de ficheiros em *heap*. Como tal, no que diz respeito a tabelas, todas as páginas são logicamente equivalentes e cada tuplo, pode ser guardado em qualquer página. Em relação aos índices, a primeira página é geralmente reservada para guardar meta-informação.

Cada página, no sistema *PostgreSQL*, é composta por vários itens, de um modo coerente com a maioria das *slotted-pages* numa organização em *heap*.

Regressando à forma como os registos de tamanho elevado são tratados pelo *PostgreSQL*, tal como anteriormente referido, este recorre à técnica *TOAST*.

Apenas certos tipos de dados suportam *TOAST,* não havendo como tal necessidade de impor o *overhead* em tipos de dados que não produzem campos de elevado tamanho. De modo a suportar esta técnica, o item deve ser do tipo *variable-length* (*varlena,* ou array de tamanho variável), no qual os primeiros 4*B* contêm o tamanho total dos dados armazenados.

O limite máximo de um atributo *TOAST* é 1*GB*. É ainda possível definir um atributo do tipo *OID*, e utilizar operações de importação e exportação de *large objects*, tais objetos podem atingir tamanhos até 2*GB*.

3. Indexação e *hashing*

3.1. Estruturas de dados de indexação suportadas

À semelhança do *Oracle*, o *PostgreSQL* suporta vários tipos de índices. Por omissão, o comando `CREATE INDEX` cria um novo índice do tipo *b-tree*, que corresponde a uma árvore B+. É possível atribuir ordenações complexas, que podem utilizar múltiplas colunas, na criação de um índice.

O sistema suporta ainda índices de *hash* e mecanismos para a implementação de outros tipos de índice. Graças a esses mecanismos, o sistema *Postgres* é extremamente rico no que diz respeito aos tipos de índices suportados. Exemplos de outros índices suportados são as *r-trees* e *bitmaps*.

3.2. Suporte de índices *clustered*

Por omissão, aquando da criação de uma tabela com uma chave primária, em oposição ao *Oracle*, o sistema a ser estudado não impõe uma organização física da mesma pela chave primária, isto é, não é criado nenhum índice *clustered*. Para tal, é necessário criá-lo manualmente.

Criação de um índice *clustered* na tabela t, índice t_pk

```
ALTER TABLE t CLUSTER ON (t_pk);
CLUSTER t;
```

3.3. Suporte de múltiplos ficheiros de índice para um mesmo conjunto de atributos

O sistema *PostgreSQL* oferece suporte a múltiplos índices sobre o mesmo atributo, tal como o *Oracle.* Para tal, basta que cada índice contenha um nome diferente, pois mesmo índices com tipos idênticos podem coexistir.

Criação de vários índices sobre o mesmo atributo

```
CREATE INDEX i1 ON t (x);
CREATE INDEX i2 ON t USING hash (x);
```

3.4. Índices de *hash*

Caso haja necessidade de criar um índice de *hash*, deve ser utilizado o comando `CREATE INDEX i ON t USING hash (c)` - o sistema implementa *hashing* dinâmico.

No entanto, é de salientar que as operações sobre índices de *hash* não são *WAL-logged*, como tal, os índices necessitam de ser reconstruídos através do comando `REINDEX` caso hajam modificações não aplicadas após um *crash* da base de dados. Por esta e outras

razões, o uso deste tipo de índices é desencorajado no *PostgreSQL*. Por razões idênticas, o *Oracle* não suporta índices baseados em tabelas de *hashing*, suportando, contudo, uma forma limitada de *static hashing* para a criação de partições sobre tabelas ou ficheiros de índice. O mecanismo de *hashing* no *Oracle* pode também ser usado para organizar *clusters* e *multitable clusters*.

3.5. Inconsistência temporária de estruturas de dados

À semelhança do *Oracle*, o sistema estudado recorre à implementação de restrições de integridade do tipo *deferrable* para que a consistência das tabelas possa ser verificada apenas no final de cada transação em vez de imediatamente após a execução de cada comando.

Mais informação e exemplos sobre este tópico pode ser encontrada na secção 5.7.

4. Processamento e otimização de perguntas

4.1. Linguagem intermédia

Pesquisas são efetuadas no sistema *Postgres* utilizando a linguagem *SQL*. Como se sabe, a linguagem *SQL* exprime apenas aquilo que se quer, sem especificar como se quer. De modo a poder efetuar uma otimização do plano de execução de uma pergunta, o sistema *Postgres* representa o plano de execução como uma árvore, em que cada nó da árvore possui uma relação direta com um operador da álgebra relacional estendida. É ainda mantida informação sobre o algoritmo a executar em cada operação. Toda esta discussão é também válida para o sistema *Oracle*.

4.2. Implementação de operações básicas

As operações básicas implementadas pelo *Oracle* e definidas pelo *standard SQL* são também suportadas pelo *PostgreSQL.* Estas são posteriormente traduzidas para algoritmos compreensíveis pelo sistema para a execução de perguntas às bases de dados. Estes algoritmos dividem-se em três subconjuntos principais: seleção, junção e ordenação.

4.3. Algoritmos suportados

Começando pelas operações de seleção, o *PostgreSQL* suporta os algoritmos de pesquisa linear, pesquisa por índice e pesquisa por índice de *bitmap*, podendo este último recorrer a múltiplos índices de forma a executar, por exemplo, uma conjunção lógica entre dois *bitmaps* para processamento de operações lógicas complexas.

No que diz respeito às operações de junção, os algoritmos suportados são o *nested loop join*, *merge join* e *hash join*. Neste tipo de operações, o otimizador de perguntas do *PostgreSQL* analisa várias ordens ou possibilidades de junção das diferentes tabelas envolvidas na pergunta e determina a opção que espera ser a mais adequada. É ainda de salientar que neste tipo de operações, quando um dado *threshold* é ultrapassado, o *PostgreSQL* recorre a um módulo chamado *genetic query optimizer*, abordado em maior detalhe a seguir.

Aquando da necessidade de ordenação de uma tabela através da utilização da cláusula `ORDER BY`, ou devido à necessidade de eliminação de repetições graças a um `SELECT DISTINCT`, ou devido a operações de conjuntos ou ainda funções de agregação numa pergunta à base de dados, o otimizador de perguntas do *PostgreSQL* irá considerar uma pesquisa sequencial sobre a tabela realizando uma ordenação explícita ou a criação de um índice para realizar a ordenação dos tuplos. No caso de um pedido de ordenação explícito e caso a tabela caiba em memória na totalidade, é feita uma ordenação com base em algoritmos de ordenação em memória (*quicksort*), caso isso não aconteça, será utilizado o algoritmo *external merge sort*. Algoritmos de *hashing* são ainda suportados para efetuar operações sobre conjuntos ou agregações.

4.4. Mecanismos de suporte para expressões complexas

Dos mecanismos de suporte a expressões complexas estudados, o *PostgreSQL* suporta dois deles, nomeadamente *materialização* e *pipelining*.

No que diz respeito à materialização, no *PostgreSQL* a utilização de materialização no otimizador de perguntas está ativa por omissão. No entanto, é possível *desligá-la* (exceto em casos em que seja necessária para correção) explicitamente, desincentivando assim o *PostgreSQL* a utilizá-la.

Em relação ao *pipelining,* não existe qualquer forma explicita de forçar ou anular a sua utilização. Há apenas que acrescentar que o sistema utiliza uma abordagem *on demand* na transmissão de tuplos de um nó para outro enquanto executa um plano.

No que respeita a otimização de *queries*, os dados estatísticos são de enorme importância para a sua eficiência, como tal, é importante salientar que a forma mais correta de tentar melhorar a performance do *PostgreSQL* na resposta a este tipo de perguntas é através da atualização dos dados estatísticos (correndo o comando `ANALYZE`), ou incrementando o valor do parâmetro de configuração `default_statistics_target` (valores superiores aumentam o tempo necessário para correr a operação `ANALYZE` mas melhoram a qualidade das estimativas utilizadas pelo otimizador de perguntas).

4.5. Escolha de entre os vários algoritmos

O otimizador normal do PostgreSQL faz uma procura quase exaustiva sobre o espaço de estratégias alternativas. O algoritmo (*System R*) foi introduzido pela *IBM* e produz uma ordenação quase ótima dos *joins* (a operação mais problemática na geração de perguntas), embora possa demorar uma grande quantidade de tempo e memória quando o numero de operações de *join* é elevado. Este problema levou à implementação de um novo algoritmo, *Genetic Query Optimizer* (*GEQO*).

O *GEQO* utiliza o algoritmo geral quando se trata de gerar planos para *scans* de relações individuais, sem *joins*. Os planos que contêm *joins* são desenvolvidos usando o algoritmo genético de otimização. O algoritmo genético utiliza primeiro o algoritmo normal para gerar sequências de forma aleatória e para calcular o custo das operações para cada *join*. A partir desse ponto são eliminados os piores planos e, de entre os melhores, é utilizado um algoritmo genético para mutar os melhores genes, escolhidos de forma aleatória. Assim, a solução encontrada a um dado momento durante a execução é utilizada para gerar o plano final.

Este algoritmo diferenciará o seu resultado dependendo do tempo que demora a ser executado e consoante a semente. Deste modo este método é não determinista. Para evitar tal facto é parametrizada uma semente que, desde que seja mantida em conjunto com os restantes parâmetros, permite que seja devolvido o mesmo plano para execuções distintas da mesma pergunta.

Em comparação com o *Oracle*, o *Postgres* não oferece a possibilidade de definir *optimization hints*. Em contrapartida, o *Oracle* não implementa nenhum algoritmo para gerar planos para perguntas que envolvam muitas tabelas.

4.6. Transformações de pergunta

Tendo em conta que o *Postgres* implementa o algoritmo *System R* definido pela *IBM*, podemos esperar as transformações por ele especificadas. Nomeadamente, equivalências entre expressões de álgebra relacional, consideração de apenas *joins* sequenciais, verificação de *ordens interessantes*, certas heurísticas relativas *ao que fazer primeiro* e *ao que deixar para o fim*, entre outras.

4.7. Utilização de estatísticas

As estimativas utilizadas pelo *PostgreSQL* são obtidas pelo subsistema *statistics collector.* Este consegue contar os acessos a tabelas e índices tanto em termos de blocos de disco como em termos de linhas individuais. Além disso, regista também o número total de tuplos por tabela e informações sobre as ações *vacuum* e *analyze* para cada tabela. Por fim, contabiliza também o número de chamadas a funções *user-defined* e o tempo despendido em cada uma delas.

4.9. Parametrização e construção de estimativas

De forma a auxiliar o uso de estimativas no *PostgreSQL,* este disponibiliza o comando `ANALYZE` já referido anteriormente. Este comando reúne estatísticas sobre o conteúdo das tabelas na base de dados e armazena os seus resultados no catálogo do sistema `pg_statistics`. Por conseguinte, o otimizador de perguntas do *PostgreSQL* utiliza estas estatísticas de modo a determinar de forma mais precisa os planos de execução mais eficientes para as perguntas. Se utilizado sem parâmetro, o `ANALYZE` analisa todas as tabelas da base de dados corrente, com a utilização de parâmetro, analisa apenas a tabela especificada.

Utilização do comando `ANALYZE`

```
ANALYZE; -- Analisa todas as tabelas da base de dados corrente
ANALYZE t; -- Analisa a tabela t
ANALYZE t(a); -- Analisa o atributo a da tabela t
```

4.10. Mecanismos de visualização de planos

O *Postgres* fornece o comando `EXPLAIN` para descobrir, sem executar, qual o plano a ser executado para uma determinada pergunta.

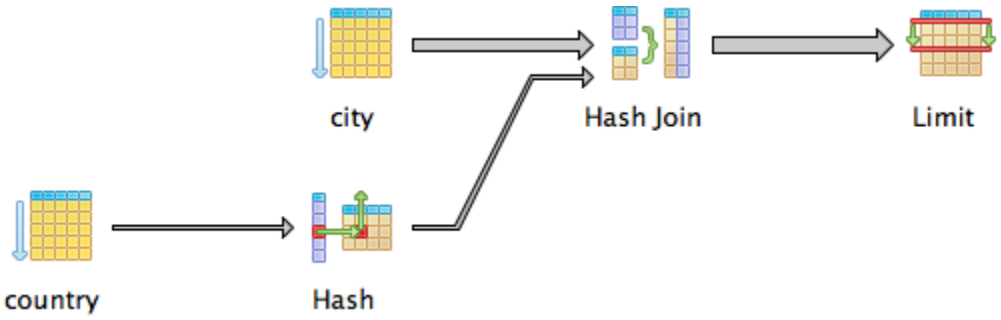
Através da interface gráfica disponibilizada pelo programa *pgAdmin* é possível visualizar o plano gerado de um modo mais *user-friendly*.

Utilização do comando `EXPLAIN` utilizando a base de dados *mondial*

```
EXPLAIN SELECT
    city.name,
    country.name
FROM city
    JOIN country ON (city.country = country.code)
LIMIT 5
OFFSET 100;

-- Resultado do EXPLAIN:
--
-- Limit  (cost=11.53..11.69 rows=5 width=18)
-- ->  Hash Join  (cost=8.36..107.24 rows=3111 width=18)
--     Hash Cond: ((city.country)::text = (country.code)::text)
--     ->  Seq Scan on city  (cost=0.00..56.11 rows=3111 width=12)
--     ->  Hash  (cost=5.38..5.38 rows=238 width=12)
--         ->  Seq Scan on country  (cost=0.00..5.38 rows=238 width=12)
```

`EXPLAIN` visualizado através do *pgAdmin*



5. Gestão de transações e controlo de concorrência

5.1. Definição de transações

Em contraste com o sistema *Oracle*, no qual cada conexão à base de dados é uma transação, sendo uma nova transação automaticamente iniciada após o fim da transação corrente, em *PostgreSQL*, as transações definem-se encapsulando o código SQL respetivo à transação entre os comandos `BEGIN` e `COMMIT`. Cancelar uma transação quando esta está já a executar é também possível através do comando `ROLLBACK`.

Na verdade, para o sistema, qualquer comando executado é visto como uma transação. No caso de o comando `BEGIN` não ser explicitado, cada comando individual possui um `BEGIN` implícito, assim como um `COMMIT` no final do mesmo, caso este execute sem erros, à semelhança de comandos de *DDL* no sistema *Oracle*.

Por omissão, no sistema a ser estudado, quando um erro ocorre numa transação, tal transação fica automaticamente no estado *aborted*, fazendo com que quaisquer comandos executados após o erro falhem com a mensagem: *current transaction is aborted, commands ignored until end of transaction block*. Tal comportamento difere do sistema *Oracle* em que um comando erróneo é simplesmente ignorado, sem forçar um *rollback* da transação.

Exemplo de uma transação em *PostgreSQL*

```
BEGIN;
UPDATE accounts
    SET balance = balance - 100.00
    WHERE name = 'Alice';
UPDATE accounts
    SET balance = balance + 100.00
    WHERE name = 'Bob';
COMMIT;
```

5.2. Suporte para transações de longa duração

De modo a possibilitar transações de longa duração, nas quais a probabilidade de algum erro ocorrer é mais elevada, assim como para contornar o comportamento por omissão das transações, em que um erro faz com que uma transação fique em estado *aborted*, o sistema disponibiliza a utilização de *savepoints*.

Através do comando `SAVEPOINT <name>`, o sistema cria um *savepoint*, que pode ser utilizado para permitir o *rollback* de uma transação para um estado intermédio da mesma - com o comando `ROLLBACK TO <name>`.

Abaixo apresentamos um exemplo (retirado do manual de documentação do *PostgreSQL*) de uma transação - uma transferência de dinheiro simplificada, que utiliza *savepoints*. No exemplo, Alice decide, após entregar a Bob os seus 100€, que afinal não era a Bob que estava a dever dinheiro, mas sim a Wally. Por sorte tinha declarado um *savepoint* após remover o dinheiro da sua conta, pelo que não necessitou de executar um *rollback* completo da transação, mas sim um *rollback* até ao respetivo *savepoint*.

Exemplo de transação utilizando *savepoints*

```
BEGIN;
UPDATE accounts
    SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts
    SET balance = balance + 100.00
    WHERE name = 'Bob';
-- Oops, I meant Wally, not Bob
ROLLBACK TO my_savepoint;
UPDATE accounts
    SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

O sistema disponibiliza ainda o comando `RELEASE SAVEPOINT <name>`, que permite a destruição de um *savepoint*.

De modo a clarificar, deve-se acrescentar que, como mecanismo de suporte a transações de longa duração, o *PostgreSQL* disponibiliza apenas *savepoints*, querendo com isto dizer que não dispõe de mecanismos para a criação de *nested transactions*, onde uma transação pode ser iniciada dentro de outra transação. Por curiosidade, a tentativa de declaração do comando de início de transação dentro de uma transação resulta na mensagem: *there is already a transaction in progress*.

5.3. Protocolo para garantia de isolamento

O isolamento de uma transação é garantido no sistema através de um mecanismo de controlo de concorrência baseado num protocolo de múltiplas versões - *MVCC* (*Multi Version Concurrency Control*) - em semelhança com o sistema *Oracle*. Tal significa que, enquanto os dados estão a ser consultados, cada transação vê um *snapshot* dos dados (uma versão da base de dados) possivelmente não coerente com a versão atual da base de dados. Este comportamento protege uma transação de ver dados inconsistentes causados por transações concorrentes. Tal protocolo garante uma propriedade desejada na maioria dos casos de uso: operações de leitura nunca bloqueiam operações de escrita e vice-versa.

O protocolo funciona do seguinte modo:

- A cada transação está associado um identificador denominado de *XID* - incluindo transações de um só comando. Quando uma transação é iniciada, é-lhe associada o *XID* resultante do incremento do *XID* da última transação iniciada.
- O *Postgres* mantém informação para controlo das transações em cada tuplo da base de dados, tal informação é utilizada para saber se um dado tuplo é visível para uma dada transação. Tal é feito do seguinte modo: cada tuplo possui um campo denominado *xmin*, que indica qual a transação que tornou o dito tuplo corrente (através de um `INSERT` ou `UPDATE`) e um campo *xmax* que indica qual a transação que expirou o tuplo

(através de um `UPDATE` ou `DELETE`). Ao aceder aos dados, o sistema verifica cada tuplo e determina se este é ou não visível à transação.

Embora os atributos *xmin* e *xmax* estejam geralmente escondidos, é possível consultá-los, pedindo-os explicitamente numa query. É também possível saber qual o *XID* de uma transação enquanto esta executa.

Consultar *xmin* e *xmax*

```
SELECT *, xmin, xmax
    FROM table;
```

Consultar *XID* de uma transação

```
SELECT txid_current();
```

Como diferentes transações têm acesso a um conjunto diferente de tuplos, o sistema necessita de manter tuplos potencialmente obsoletos: razão pela qual um `UPDATE` na realidade cria um novo tuplo, assim como um `DELETE` apenas marca um tuplo como removido, em vez de o apagar verdadeiramente. Tal comportamento leva à presença de tuplos que já não podem ser lidos por qualquer transação: *dead rows*.

Tais tuplos são eliminados de tempo a tempo atravém de rotinas que executam periodicamente. Do mesmo modo é resolvido outro problema, relacionado com o valor máximo do *XID*.

5.4. *Deadlocks*

Porque o *MVCC* utiliza *locks* para coordenar escritas concorrentes e porque o *Postgres* permite a declaração de *locks* em qualquer ordem, *deadlocks* podem ocorrer. Para os contornar, o sistema dispõe de um algoritmo de deteção e quebra de *deadlocks*.

O *PostgreSQL* foi concebido de modo a que operações rotineiras (em que há atribuição ou liberação de *locks*) corram rapidamente quando não existem *deadlocks*, evitando o tratamento de *deadlocks* o quanto possível. Tal é conseguido utilizando uma abordagem otimista na qual uma transação, ao não conseguir adquirir um *lock*, é adormecida sem qualquer verificação de existência de *deadlocks*. Ao ser adormecida, a transação inicia um contador interno. Quando o contador atinge um determinado tempo (configurável, por omissão 1 segundo), se a transação ainda não tiver conseguido adquirir o *lock*, o algoritmo de deteção e quebra de *deadlocks* é executado. Se nenhum *deadlock* for detetato a transação volta a adormecer até o seu *lock* ser adquirido, caso contrário o *deadlock* é resolvido, normalmente por abortar a transação que o detetou (o processo de escolha da vítima não é fidedigno). Desta forma evita-se tratamento de *deadlocks* sempre que o tempo de espera por um *lock* é menor do que o tempo do contador.

O sistema *Oracle*, em contraste, para quebrar um *deadlock*, ignora o último comando da transação mais antiga que esteja envolvida no *deadlock*, de modo a tentar minimizar a

probabilidade de ocorrências seguidas de *deadlocks* (o sistema assume que, na maioria dos casos, é efetuado um *rollback a* uma transação assim que um *deadlock* é detetado).

Deadlock em *PostgreSQL*

```
BEGIN;                                      BEGIN;
UPDATE accounts
    SET balance = balance - 100.00
    WHERE name = 'Alice';
-- Update occurs
                                            UPDATE accounts
                                                SET balance = balance - 100.00
                                                WHERE name = 'Bob';
                                            -- Update occurs
UPDATE accounts
    SET balance = balance + 100.00
    WHERE name = 'Bob';
-- Transaction blocks
                                            UPDATE accounts
                                                SET balance = balance + 100.00
                                                WHERE name = 'Alice';
                                            -- Deadlock is detected after 1 second
                                            and
-- Update occurs and transaction            -- transaction is aborted
proceeds
```

Deadlock em *Oracle*

```
COMMIT; -- End last transaction             COMMIT; -- End last transaction
UPDATE accounts
    SET balance = balance - 100.00
    WHERE name = 'Alice';
-- Update occurs
                                            UPDATE accounts
                                                SET balance = balance - 100.00
                                                WHERE name = 'Bob';
                                            -- Update occurs
UPDATE accounts
    SET balance = balance + 100.00
    WHERE name = 'Bob';
-- Transaction blocks
                                            UPDATE accounts
                                                SET balance = balance + 100.00
                                                WHERE name = 'Alice';
                                            -- Transaction blocks
-- Deadlock is detected and update
command
-- is ignored, transaction proceeds
```

Como algoritmo de deteção e quebra de *locks* o *Postgres* utiliza o método *standard*, no qual cada transação é vista como um nó num grafo orientado (*wait-for graph*). Informação

sobre os *locks* mantidos por cada transação é guardada na tabela *pg_locks* (que pode facilmente ser consultada).

5.5. Níveis de granularidade de *locks*

À semelhança do sistema *Oracle*, o *PostgreSQL* oferece diferentes tipos de *locks*, que podem ser adquiridos manualmente (no caso do *Postgres* através do comando `LOCK`), de forma a implementar protocolos de controlo de concorrência que não estejam de acordo com as especificações do *MVCC*. A maioria dos comandos executados, no entanto, adquirem certos *locks* automaticamente, de modo a que modificações concorrentes de tabelas referenciadas pelo comando não causem problemas.

O sistema a ser estudado suporta *locking* a nível da tabela e a nível do tuplo. A cada nível de *locking* corresponde um conjunto de modos de *locking* que especificam quais os *locks* que entram em conflito entre si. Tal funcionalidade é análoga ao sistema *Oracle*, embora os nomes dos modos de *locking* difiram ligeiramente.

O *Postgres* suporta ainda a utilização de *locks* cujo significado é definido pelo utilizador, aos quais dão o nome de *advisory locks*. Tais *locks* possuem a propriedade especial de poderem ser definidos ao nível da sessão, isto é, de não obterem o comportamento esperado dos restantes tipos de *locks*, que são automaticamente libertos aquando do fim da transação que os adquiriu.

5.6. Níveis de isolamento

O sistema estudado suporta todos os níveis de isolamento especificados no *standard SQL*, embora as garantias que cada nível impõe são por vezes mais fortes do que as definidas no *standard*. O *Postgres* suporta apenas, na realidade, três níveis distintos de isolamento: correspondentes aos níveis *read commited*, *repeatable read* e *serializable*. Uma transação que especifique o nível *read uncommited* executará, na verdade, em nível *read commited* (pelo que não será alvo do fenómeno *dirty read*). Do mesmo modo, uma transação iniciada em modo *repeatable read*, de facto, executará num modo equivalente ao *serializable* do sistema *Oracle* (*snapshot isolation* - que não é realmente *serializable*), o que garante que *phantom reads* não ocorrerão.

O *PosgreSQL*, em contraste com o *Oracle*, implementa serialização verdadeira (evita *write skews*) quando em modo *serializable* - tal é conseguido através de um protocolo de controlo de concorrência implementado sobre o *MVCC* denominado de *serializable snapshot isolation* (*SSI*). No exemplo apresentado de seguida demonstramos a diferença entre o nível *serializable* oferecido pelo *PostgreSQL* e pelo *Oracle*. O exemplo assume um esquema de base de dados em que existe uma tabela `dots` com o atributo `color`. A tabela contém 2 tuplos, um com o valor `'white'`, o outro com o valor `'black'`. Duas transações são executadas concorrentemente: uma altera todos os tuplos com `color` a `'white'` para `'black'` e a outra altera todos os tuplos com `color` a `'black'` para `'white'`. Um escalonamento em série destas transações deverá produzir uma tabela `dots` na qual ambos os tuplos têm o mesmo valor.

Nível *serializable* em *PostgreSQL*

```
BEGIN;
UPDATE dots
    SET color = 'black'
    WHERE color = 'white';
-- Update occurs
```

```
BEGIN;
```

```
UPDATE dots
    SET color = 'white'
    WHERE color = 'black';
-- Update occurs, conflict has been
-- detected: second to commit fails
```

```
COMMIT;
-- Successfully commits
```

```
COMMIT;
-- Throws error, could not serialize
-- transactions

SELECT * FROM dots;
-- Shows:
-- color
-- ------
-- black
-- black
```

Nível *serializable* em *Oracle*

```
COMMIT; -- End last transaction
UPDATE dots
    SET color = 'black'
    WHERE color = 'white';
-- Update occurs
```

```
COMMIT; -- End last transaction
```

```
UPDATE dots
    SET color = 'white'
    WHERE color = 'black';
-- Update occurs, no conflict detected
```

```
COMMIT;
-- Successfully commits
```

```
COMMIT;
-- Successfully commits

SELECT * FROM dots;
-- Shows:
-- color
-- ------
-- black
-- white
```

Como se pode observar pelo exemplo apresentado, e como já foi referido, o sistema *Oracle* não implementa serialização verdadeira, mas sim *snapshot isolation*, ficando deste modo desprotegido de *write skews* - o nome que se dá a este tipo de inconsistências.

Por omissão, em ambos os sistemas, qualquer transação iniciada terá nível de isolamento *read commited*. O nível de isolamento pode ser alterado através do comando `SET TRANSACTION ISOLATION LEVEL`.

5.7. Verificação de consistência

A consistência, no *PostgreSQL*, ao nível de toda a base de dados e no contexto transacional, pode ser garantida através da utilização de níveis fortes de isolamento, como o caso do *serializable*. Ao utilizar níveis mais fracos, como o nível por omissão (*read commited*) é importante utilizar *locks* explícitos através de comandos como `SELECT FOR UPDATE`, `SELECT FOR SHARE` ou `LOCK TABLE`, de modo a proteger os dados de atualizações concorrentes.

Restrições de integridade impostas através da declaração de *constraints* como chaves estrangeiras são, por omissão, verificadas após cada comando. De forma a ultrapassar limitações da linguagem *SQL*, que não permite, por exemplo, a inserção de tuplos em múltiplas tabelas através de um só comando (o que pode ser problemático, no caso em que temos relações *muitos para muitos, totais em ambos os lados*), é possível declarar restrições como *deferrable*, de modo a poder especificar, aquando de uma transação, o momento de verificação da mesma (após cada operação ou no final da transação).

Utilizar modo *deferrable* para introduzir dados numa relação *muitos para muitos, total em ambos os lados*

```
CREATE TABLE husbands (
    name VARCHAR PRIMARY KEY,
    wife VARCHAR
);

CREATE TABLE wives (
    name VARCHAR PRIMARY KEY,
    husband VARCHAR REFERENCES husbands (name) DEFERRABLE INITIALLY DEFERRED
);

ALTER TABLE husbands ADD FOREIGN KEY (wife) REFERENCES wives (name) DEFERRABLE
INITIALLY DEFERRED;

BEGIN;

INSERT INTO husbands VALUES('Bob', 'Alice');
INSERT INTO wives VALUES('Alice', 'Bob');

COMMIT;
```

5.8. Atomicidade e durabilidade

A atomicidade é garantida no sistema *Postgres*, assim como no *Oracle*, graças às políticas de visualização de tuplos inerentes ao *MVCC*. Quando uma transação aborta, quaisquer tuplos modificados pela transação não são visíveis nas outras transações, criando a ilusão de que nada sucedeu. Por outro lado, quando uma transação entra em estado *commited*,

novas transações (ou transações que estavam já a executar, dependendo do nível de isolamento) passam a ver as modificações efetuadas. Desde modo, obtemos o comportamento de *tudo ou nada* esperado de uma transação.

Com o objetivo de garantir durabilidade, o sistema *PostgreSQL* utiliza *Write-Ahead Logging* (*WAL*). À semelhança dos *Redo Logs* implementados pelo sistema *Oracle*, o objetivo do *WAL* é escrever para ficheiros de *logging*, localizados em memória estável, informação relativa às modificações efetuadas na base de dados, antes de estas serem de facto aplicadas nos locais a que pertencem. Na eventualidade de uma falha do sistema, existirá informação suficiente nos ficheiros de *logging* para deixar a base de dados num estado consistente - quaisquer alterações que ainda não tenham sido aplicadas na base de dados são reexecutadas a partir dos registos de *log* (sistema de recuperação conhecido como *REDO logging*).

Checkpoints são utilizados como mecanismo para forçar o *output* dos dados para a base de dados - após um *checkpoint*, todas as transações que efetuaram *commit* em algum momento anterior ao do *checkpoint* têm as suas modificações refletidas na base de dados. O intervalo de tempo entre *checkpoints* é um parâmetro configurável em ambos os sistemas (*Postgres* e *Oracle*).

É possível configurar o sistema *Postgres* de modo a aumentar a *performance* do sistema em custo da garantia de durabilidade. Um exemplo de tal configuração é a desativação dos *commits* síncronos, isto é, uma transação poderá ser dada como concluída antes de a informação de *logging* ser armazenada em memória estável.

6. Suporte para bases de dados distribuídas

6.1. Bases de dados distribuídas homogéneas

Bases de dados distribuídas homogéneas são suportadas tanto pelo *Oracle* como pelo *PostgreSQL*. Um mecanismo base para a configuração de bases de dados distribuídas em ambos os sistemas consiste na criação de *links* entre diferentes bases de dados.

Utilizando o módulo `postgres_fdw`, que fornece um serviço de *foreign data-wrapping*, é possível aceder a dados existentes num servidor *Posgres* externo. Embora existam outros módulos disponíveis para comunicação com bases de dados remotas, tanto quanto pudemos observar, o suporte para bases de dados distribuídas no *PostgreSQL* está ainda muito aquém do que é fornecido pela *Oracle*, nomeadamente através da especificação de vistas materializadas em bases de dados remotas.

Criação de uma ligação a outro servidor (em *localhost*)

```
-- Cria um link
CREATE SERVER s FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'localhost', port
'5432', dbname 'name');

-- Cria um mapping de um servidor para outro (pode precisar de informação de login)
CREATE USER MAPPING FOR user SERVER s;

-- Deve-se criar uma tabela estrangeira para cada tabela remota a que se queira
-- aceder, neste caso a tabela t com a coluna c do tipo INTEGER
CREATE FOREIGN TABLE t_remote (c INTEGER) SERVER s OPTIONS (table_name 't');

-- Operações em t_remote devem funcionar
SELECT * FROM t_remote;
```

6.2. Mecanismos de suporte à replicação

A replicação é suportada em ambos os sistemas. No *PostgreSQL*, esta é assegurada através da definição de dois tipos de servidores, *master* e *standby*. Os servidores do tipo *master* podem enviar dados, enquanto os servidores do tipo *standby* são recetores de dados replicados. É ainda possível utilizar replicação em cascata, ou seja, é possível ter servidores *standby* a servir de *master* a outros servidores. Tais servidores *standby* passam a aceitar conecções de replicação e a reencaminhar dados e registos do tipo *WAL (Write-Ahead Logging)*. Esta funcionalidade pode ser usada para reduzir o número de conecções ao servidor *master*, assim como para minimizar a utilização da largura de banda entre diferentes servidores.

O *PostgreSQL* fornece dois tipos de replicação de dados, com modos distintos de funcionamento. Ambos os tipos de replicação criam uma cópia completa da base de dados.

- O primeiro tipo de replicação utiliza o servidor *standby* em modo *warmStandby*, também chamado *log shipping*. O objetivo primário deste tipo de replicação é obter um nível alto

de disponibilidade. Neste modo de funcionamento, o servidor *master* envia, após encher um ficheiro de *logs*, esse mesmo ficheiro para o servidor *standby* - que, em modo de armazenamento, vai aplicando as operações recebidas. Em caso de falha do servidor principal, o servidor *standby* está (quase) atualizado e pronto para tomar a posição de servidor *master*. É possível que no momento de falha do servidor *master* certas transações tenham sido *commited* sem que tal informação chegasse ao segundo servidor, causando perda de dados. Para evitar tal perda de dados o *Postgres* disponibiliza a opção de utilização de replicação síncrona, na qual cada transação é apenas *commited* após confirmação por parte do servidor *standby* de que a informação foi recebida.

- O segundo tipo de replicação coloca o servidor *standby* em modo *HotStandby*, tal modo permite que perguntas sejam efetuadas no servidor. O seu modo de funcionamento é semelhante ao descrito acima, para o caso do *WarmStandby*.

6.3. Mecanismos de suporte à fragmentação de dados

A nossa pesquisa levou-nos a constatar que fragmentação de dados a nível distribuído não é (ainda) suportada pelo *PostgreSQL*, embora tenhamos encontrado sugestões de possíveis implementações do mesmo.

6.4. Mecanismos de transparência em *queries* distribuídas

Tal como foi referido na secção 6.1. é possível criar, em *Postgres*, tabelas estrangeiras, que referenciam tabelas numa outra base de dados. Tal mecanismo fornece, portanto, embora parcialmente e à semelhança com o sistema *Oracle*, transparência de localização.

Transparência de replicação é também fornecida parcialmente. Um utilizador não notará, por exemplo, novos servidores replicados - assim como não deverá notar a atribuição de um novo servidor *master* em caso de falha do atual.

6.5. Atomicidade em transações globais

De forma a assegurar a atomicidade em transações globais, tanto o *Oracle* como o *Postgres* utilizam o protocolo *2-phase commit*. Para a sua implementação, o servidor no qual a transação executa é utilizado como *transaction coordinator*, responsável por iniciar, distribuir e coordenar transações.

6.6. *Locks* globais

Tanto o *Oracle* como o *PostgreSQL* implementam apenas *locking* na cópia primária. Na verdade, o *PostgreSQL*, como não dispõe de mecanismos de replicação que permitam escritas nas réplicas, não necessita de se preocupar com tais *locks*.

6.7. Formas restritivas de uso

Como já foi anteriormente referido, réplicas *HotStandby* não podem ser modificadas explicitamente pelo utilizador, funcionando apenas para efetuar perguntas.

No contexto de acesso a bases de dados remotas, é possível configurar, em *Postgres*, as permissões de cada utilizador à base de dados, de modo que é possível proibir operações de escrita ou outras, de acordo com o desejado.

7. Outras características do sistema estudado

7.1. Suporte para a *web*

O sistema *Postgres* suporta, como tipos de dados nativos, o *XML* e o *JSON*. Em relação ao *XML*, o sistema disponibiliza funções utilitárias para a exportação de resultados de *queries* em formato *XML*.

7.2. Suporte de procedimentos

O *PostgreSQL* executa procedimentos em múltiplas linguagens de programação, incluido *Java*, *Python*, *Ruby*, *C*, entre outras. O sistema dispõe ainda de uma linguagem própria, semelhante ao *PL/SQL* da *Oracle* chamada *PL/pgSQL* - a linguagem suporta a definição de funções próprias e até mesmo de funções de agregação e disponibiliza por omissão um conjunto de funções utilitárias.

7.3. Mecanismos de suporte a vários utilizadores e segurança

O *PostgreSQL* gere as permissões de acesso às bases de dados utilizando o conceito de *roles*. Um *role* pode ser visto como um utilizador da base de dados ou um grupo de utilizadores, dependendo da forma como é definido. Permissões podem ser associadas a *roles* para que um utilizador que pertença a um *role* adquira as permissões inerentes ao mesmo. Essas permissões podem controlar o acesso a objetos da base de dados.

7.4. Tipos de dados

O armazenamento de dados no *Postgres* é inteiramente relacional. Embora seja possível utilizar tipos não atómicos, como *arrays*, *XML*, ou *JSON*. O sistema disponibiliza ainda mecanismos para a criação de novos tipos, consoante o utilizador possa achar necessário.

7.5. Ferramentas associadas

Múltiplas ferramentas desenvolvidas, de *reporting*, *administração*, *tuning* e *data warehousing* podem ser utilizadas com o *PostgreSQL*. Exemplos de tais ferramentas são: *PostgreStats* para *reporting*, *pgAdmin* para administração e *data warehousing* e *pgBadger* para *tuning*.