Mestrado Integrado em Engenharia Informática
Sistemas de Base de Dados
2013/2014

Estudo do PostgreSQL 9.3

Relatório

Realizado por:
Catarina Albino, 42181
José Arjona, 41741
Rúben Cachucho, 41937

Grupo 02

Docente:
José Júlio Alferes

1 de Junho de 2014

Índice

1 Introdução

Desenvolvido no âmbito da disciplina de Sistemas de Bases de Dados, o presente trabalho tem por objetivo a análise de um sistema de gestão de bases de dados (SGBD) no que respeita aos diversos tópicos lecionados na disciplina. O SGBD por nós escolhido foi o **PostgreSQL**, que será analisado ao longo deste documento no que diz respeito à sua evolução histórica, cobertura que faz ao standard SQL, o modo de armazenamento e estruturas de ficheiros usadas, bem como as várias formas de indexação e *hashing* suportadas, como é feito o processamento e otimização de perguntas, a gestão de transações e o controlo de concorrência.

Por último, efetuaremos uma comparação das funcionalidades implementadas relativamente ao Oracle 11g (SGB usado para a experimentação da componente prática da cadeira).

1.1 Estrutura do documento

Em termos de organização, o documento segue a estrutura proposta pelo docente para a sua elaboração, tendo as seguintes secções:

1. Introdução;
2. Cobertura do SQL;
3. Armazenamento e Estrutura de Ficheiros;
4. Indexação e Hashing;
5. Processamento e Otimização de Perguntas;
6. Gestão de Transações e Controlo de Concorrência;
7. Suporte para Bases de Dados Distribuídas;
8. Outras Características;
9. Comparação com o Oracle 11g.

1.2 Introdução histórica e aplicabilidade do PostgreSQL

O PostgreSQL é um sistema de gestão de base de dados orientado a objetos (SGBDOR) *open-source*, baseado no sistema Postgres (Versão 4.2). Desenvolvido na Universidade da Califórnia em Berkeley por uma equipa liderada pelo professor Michael

Stonebraker, em 1986, o antecessor Postgres foi pioneiro em muitos dos conceitos que se tornaram disponíveis nos SBD comerciais muito mais tarde.

Criado para resolver a lacuna que o modelo relacional tinha em compreender tipos (atualmente denominados de objetos), o Postgres introduziu funcionalidades que incluíam a definição de suporte a tipos e também a capacidade de descrever relações. Apesar de ser até então principalmente usado no contexto académico, este sistema foi posteriormente comercializado (com o nome Illustra).

Em 1995, dois estudantes do professor Michael Stonebraker, Andrew Yu e Jolly Chen, adicionaram um interpretador SQL (Structured Query Language) ao Postgres, e o projeto foi renomeado para Postgres95. Em 1996, o projeto foi divulgado pela Internet, passando a ser *open source*. Nesse mesmo ano, o projeto foi novamente renomeado para PostgreSQL, refletindo a utilização da linguagem de acesso à base de dados, sendo este o nome atualmente utilizado.

A primeira versão do PostgreSQL (6.0, devido aos anos de desenvolvimento anteriores) foi lançada em 1997, e desde essa altura que tem sido mantido por programadores e voluntários de todo o mundo, que têm contribuído para o desenvolvimento de novas funcionalidades e melhorias no sistema. Atualmente, o PostgresSQL vai na versão 9.3.4, lançada a 20 de Março de 2014, tendo sido entretanto lançada a *beta release* da versão 9.4, a 15 de Maio deste ano.

O projeto é sustentado por doações e pelo patrocínio de diversas empresas. É um SGBD que tem adquirido prestígio na comunidade Linux, e apesar de estar muito presente na comunidade *open source*, tem grandes empresas internacionais e órgãos governamentais de vários países entre os seus utilizadores e contribuidores.

Apesar de ser essencialmente um projeto com origem académica, o PostgreSQL tem capacidade para lidar com bases de dados de diversas dimensões, podendo ser usado sobre diversas plataformas, como o Windows ou Linux.

1.3 Cobertura do SQL

O PostgreSQL suporta grande parte das funcionalidades da linguagem standard SQL, e oferece muitas funcionalidades atuais tais como consultas complexas, chaves estrangeiras, *triggers*, vistas, integridade transacional e controlo de concorrência multiversão.

Além disso, o PostgreSQL pode ainda ser estendido pelo utilizador de diversas

formas entre as quais a adição de novos tipos de dados, funções, operadores, métodos de indexação, entre outros.

No que diz respeito às principais funcionalidades da linguagem SQL, esta pode ser dividida em subconjuntos de acordo com as operações que se pretende efetuar sobre uma base de dados. Dois desses subconjuntos são o **DML** (*Data Manipulation Language*) e o **DDL** (*Data Definition Language*).

O subconjunto DML é utilizado para realizar inserções, consultas, alterações e remoções de dados. Estas tarefas podem ser executadas em vários registos de diversas tabelas ao mesmo tempo. Os comandos PostgreSQL que realizam tais operações são descritos na tabela seguinte, bem como algumas das principais extensões que contém relativamente ao standard SQL:

Comando	**Funcionalidade**	**Compatível com o standard SQL**	**Algumas extensões existentes**
SELECT	Consultar a BD	Sim	• Cláusula DISTINCT ON.
INSERT	Inserir dados em tabelas da BD	Sim	• Cláusula RETURNING; • Possibilidade de uso de WITH com o INSERT.
UPDATE	Alterar dados existentes em tabelas da BD	Sim	• Cláusulas FROM e RETURNING; • Possibilidade de uso de WITH com o UPDATE.
DELETE	Apagar dados das tabelas da BD	Sim	• Cláusulas USING e RETURNING; • Possibilidade de uso de WITH com o DELETE.

Tabela 1: – Principais comandos DML do PostgreSQL e respetivas extensões ao standard SQL.

O subconjunto **DDL** é utilizado para manipular as estruturas para armazenar dados na BD. Assim sendo, esta componente define como é feita a criação, alteração e remoção de tabelas, *triggers* e vistas. Entre os principais comandos que realizam as funções acima referidas encontram-se os abaixo indicados:

Estes comandos podem ser aplicados a *triggers*, regras e *views*, substituindo *obj*

Comando	Funcionalidade	Compatível com o standard SQL	Algumas extensões existentes
CREATE *obj*	Criar objetos na BD	Sim	• Cláusula TEMPORARY, que permite a criação de tabelas temporárias em memória, que são automaticamente apagadas no fim de cada sessão.
DROP *obj*	Apagar objetos da BD	Sim	• Cláusula IF EXISTS; • Possibilidade de remoção de mais que uma tabela em simultâneo.
ALTER *obj*	Alterar objetos da BD	Sim	• O comando ALTER TABLE DROP COLUMN pode ser usado para apagar a única coluna de uma tabela, deixando-a sem colunas.

Tabela 2: – Principais comandos DDL do PostgreSQL e respetivas extensões ao standard SQL.

por TABLE, TRIGGER, RULE ou VIEW, para manipulação de tabelas, *triggers*, regras e vistas, respetivamente.

O PostgreSQL permite que os utilizadores adicionem os seus próprios tipos de dados, através do comando CREATE TYPE.

2 Armazenamento e estrutura de ficheiros

Qualquer base de dados tem que estar armazenada num computador, nomeadamente sob a forma de uma coleção de ficheiros. É por isso relevante estudar como é que o PostgreSQL gere esta questão.

2.1 Organização

O PostgreSQL baseia-se no file system do sistema operativo, armazenando os dados numa hierarquia de diretorias e ficheiros, localizada normalmente a partir da directoria `PGDATA` que, por sua vez, contém:

1. Outros ficheiros:
    - `PG_VERSION` – indica a versão do PostgreSQL;
    - `postmaster.opts` – contém as opções da última inicialização do servidor na linha de comandos;
    - `postmaster.pid` – ficheiro de lock com PID do servidor e ID do segmento de memória partilhado e que é apagado aquando do encerramento do sistema.
2. Outras subdirectorias:
    - `base` – contém várias subdiretorias, cada uma correspondendo a uma das bases de dados;
    - `global` – contém as tabelas válidas em todo o cluster;
    - `pg_clog` – contém os *logs* do estado de *commit* das transacções;
    - `pg_multixact` – contém os *logs* do estado das multi-transacções;
    - `pg_notify` – contém os dados de estado de `LISTEN/NOTIFY`;
    - `pg_stat_tmp` – contém ficheiros temporários de estatística;
    - `pg_subtrans` – contém os *logs* do estado de sub-transacções;
    - `pg_tblspc` – contém os *symbolic links* para as tablespaces;
    - `pg_twophase` – contém os dados de estado das transacções preparadas;
    - `pg_xlog` – contém os *Write ahead logs* (WAL), que armazenam os registos das transações de modo a prevenir perdas de dados em circunstâncias graves, como falhas elétricas.

As diretorias que representam as várias bases de dados têm como nome o OID (Object Identifier), um atributo que, apesar de não ser introduzido nas colunas das tabelas pelo utilizador, é introduzido pelo sistema e que é usado como chave primária.

Para cada tabela ou índice é criado um conjunto de ficheiros apelidados de *forks*. Um deles, o principal, é o *main fork*, que está organizado em *heap*, na forma de *slotted page,* e que guarda os tuplos, sendo cada registo identificado pelo seu id. Já os outros são apelidados através da concatenação do *filenode* respetivo com um sufixo que os distingue: `_fsm` (*Free Space Maps*, que mantém a informação do espaço disponível no momento) ou `_vsm` (*Visibility Maps*, que indicam quais as páginas que contêm apenas tuplos visíveis a todas as transacções ativas).

Cada tabela ou índice é também guardada em *arrays* de páginas de tamanho fixo, cada uma constituída pelo *header*, pelos apontadores para os registos, pelo espaço livre, pelos registos em si e pelos dados necessários para garantir o acesso a índices.

Para aquelas tabelas que têm atributos cujos valores possam ser de grande dimensão recorre-se à técnica TOAST: os dois primeiros bits da *double-word* que codifica o tamanho do atributo são usados para indicar as condições em que os dados são guardados. Se ambos forem 0, o tipo de dados é *un-Toasted*; se o primeiro for 1, significa que os dados foram comprimidos; se o segundo for 1, os dados que se seguem representam um apontador para a zona em memória onde o objecto se encontra.

2.2 Buffers

O PostgreSQL possui controlo quer de *shared buffers* (utilizados para partilhar dados entre o servidor e o sistema operativo) , quer de *temporary buffers* (utilizados em cada sessão para aceder temporariamente às tabelas).

2.3 Clustering

O comando que permite ao utilizador criar clusters com base num índice e numa só tabela é:

```
CLUSTER table_name [ USING índex_name]
```

De notar que o PostgreSQL não suporta *multitable clustering*, pelo que só pode ser utilizada uma tabela.

2.4 Partições

O PostgreSQL suporta ainda as partições, sob duas formas diferentes, com o objectivo de fragmentar uma grande tabela lógica em várias tabelas físicas pequenas, o que traz vantagens a nível da performance no acesso e manipulação dos dados.

2.4.1 Tipos de partições

a) Partição por intervalo

A tabela é dividida por intervalos definidos por uma ou mais colunas chave e sem haver qualquer sobreposição dos intervalos de valores associados a diferentes partições.

b) Partição em lista

A tabela é dividida indicando explicitamente quais os valores da chave que irão aparecer em cada partição.

2.4.2 Especificação das partições

Para definir uma nova partição, o utilizador deve seguir os seguintes passos:

1. Criar a *master table*, da qual as partições irão herdar. Esta não irá conter quaisquer dados e, a não ser que também devam ser aplicadas nas partições, também não deverão estar definidas quaisquer restrições;
2. Criar as *child tables*, que herdam da *master table* e que constituem as partições;
3. Adicionar as restrições às partições para definir os valores das chaves em cada uma delas, atendendo a que não pode haver qualquer sobreposição de valores;
4. Para cada partição, criar, pelo menos, um índice nas colunas chave;
5. Opcionalmente, definir um *trigger* para direccionar os dados inseridos na *master table* à partição adequada;
6. Verificar que a `constraint_exclusion` não está desligada no ficheiro `postgresql.conf`.

3 Indexação e *hashing*

O sistema de bases de dados em estudo suporta mecanismos de indexação, mecanismos esses que pretendem tornar o acesso aos dados mais rápido.

3.1 Sintaxe geral de criação de um índice

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

3.2 Tipos de índices

O sistema PostgreSQL suporta várias estruturas de indexação, entre as quais se destacam as seguintes:

- **B-Tree**

Este é o tipo de índice definido por defeito e encontra-se adaptado para a maioria das *queries*, nomeadamente para aquelas que envolvem comparações e intervalos. Assim sendo, poderá ser utilizado aquando da utilização dos operadores $<$, $<=$, $=$, $>=$, $>$ ou outros equivalentes a combinações dos mesmos, tais como `BETWEEN` ou `IN`. Também podem ser usadas em comparações usando `LIKE` ou `~` mas as *strings* a utilizar devem ser iniciadas por uma constante e não por `%`.

É de notar que também pode ser utilizado quando na presença de condições IS `NULL` ou `IS NOT NULL`, assim como na devolução de resultados ordenados (embora este

último caso nem sempre seja eficiente).

Permite ser definido através de várias colunas, sendo mais eficiente quando existem restrições nas primeiras ou em *queries* de igualdade ou desigualdade.

Este é também o único tipo de índice que permite a ordenação dos resultados. Por defeitos, são ordenados de forma ascendente e considerando os valores nulos por último, mas ambos os critérios podem ser definidos pelo utilizador.

É igualmente o único que pode ser declarado `UNIQUE`, isto é, que força a unicidade do valor de uma coluna ou da combinação dos valores de várias colunas. No entanto, tal restrição é preferencialmente feita na própria tabela, a partir do comando `ALTER TABLE ... ADD CONSTRAINT`.

O comando para a sua criação é:

```
CREATE INDEX index ON table(column)
```

- **Hash**

Apesar de estar disponível, este tipo de indexação não é recomendada porque apresenta problemas de performance. Por outro lado, só pode ser utilizado para comparações de igualdade (=) e não suporta índices multi-coluna.

Recorre a um algoritmo de *hashing* dinâmico, pelo que não possui qualquer *rehashing*.

O comando a utilizar para a sua criação é:

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

- **GiST (*Generalized Search Tree*)**

Mais do que um simples índice, permite implementar várias formas de indexação e, dependendo da estratégia, os operadores com os quais pode ser utilizado podem variar. No entanto, é essencialmente utilizada para comparações com os seguintes operadores:

$$<< \ ; \ \&< \ ; \ \&> \ ; \ >> \ ; \ <<| \ ; \ \&<| \ ; \ |\&> \ ; \ |>> \ ; \ @> \ ; \ <@ \ ; \ = \ ; \ \&\&$$

Este tipo de índices optimiza pesquisas de *nearest-neighbor* e também suporta índices multi-coluna. O comando para a sua criação é:

```
CREATE INDEX name ON table USING gist(column)
```

- **GIN (*Generalized Inverted Index*)**

São índices invertidos que lidam com valores multichave, tais como arrays. Pode suportar diferentes estratégias e operadores, tal como no caso anterior. Em particular, é capaz de tratar das comparações <@, @>, = e &&.

O comando para a sua criação é:

```
CREATE INDEX name ON table USING gin(column)
```

São também suportadas outras variantes mais complexas tais como índices parciais ou por expressões.Cabe referir a importância da utilização ponderada de índices multi-coluna, uma vez que nem sempre constituem o mecanismo mais eficiente, e que, para cada índice, podem ser especificadas até 32 colunas.

Também é relevante mencionar que é o próprio sistema, e não o utilizador, a escolher, se for caso disso, o índice a utilizar numa dada *query*, consoante aquele que melhor se adaptar.

3.3 Índices para organização de ficheiros

Apesar de, no sistema PostgreSQL, a tabela ser organizada em ficheiros através de uma *heap*, o utilizador pode associá-la a uma dada estrutura de indexação usando o seguinte comando:

```
CLUSTER [VERBOSE] table_name [USING índex_name]
```

Quando os registos são actualizados ou inseridos na tabela, estes não ficam organizados pelo índice, pelo que é necessário executar novamente o comando.

3.4 Estruturas temporariamente inconsistentes

O acesso às estruturas é controlado a partir do mecanismo de LOCK, mas mesmo assim as actualizações concorrentes podem dar origem a inconsistências entre uma tabela

e um índice dada a separação lógica que o sistema faz entre acesso a tabelas e acessos a índices.

Através da clausula `DEFERRABLE`, o PostgreSQL permite adiar para o final da transacção a verificação das restrições de integridade `UNIQUE, PRIMARY KEY, EXCLUDE` e `REFERENCES`, sendo que, por defeito, estas e outras restrições são `NOT DEFERRABLE` (e portanto verificadas em cada operação).

4 Processamento e optimização de perguntas

A importância de um sistema de base de dados reside, não só na forma como os dados são armazenados, mas também na forma como é processada a sua consulta.

4.1 Fases do processamento

O processo de consultas no PostgreSQL é constituído pela seguinte sequência de fases:

4.1.1 Ligação

Estabelece-se uma ligação da aplicação ao servidor do PostgreSQL segundo um "processo por utilizador" baseado no modelo cliente/servidor. O processo *postgres* recebe pedidos por uma porta TCP/IP específica e gera um processo de servidor para cada pedido de ligação recebido.

4.1.2 *Parsing*

Após verificar se a *query* está correcta em termos de sintaxe, cria uma estrutura de dados, denominada *parse tree*, com o mesmo significado da pergunta original mas estruturada formalmente de acordo com a sintaxe do SQL. De acordo com todas as interpretações semânticas das tabelas, junções, funções e operadores utilizados na *query*, esta *parse tree* é transformada numa *query tree*. Por exemplo, à query

```
SELECT name, address FROM friends WHERE age > 18 ORDER BY name
```

corresponderia a seguinte query tree:

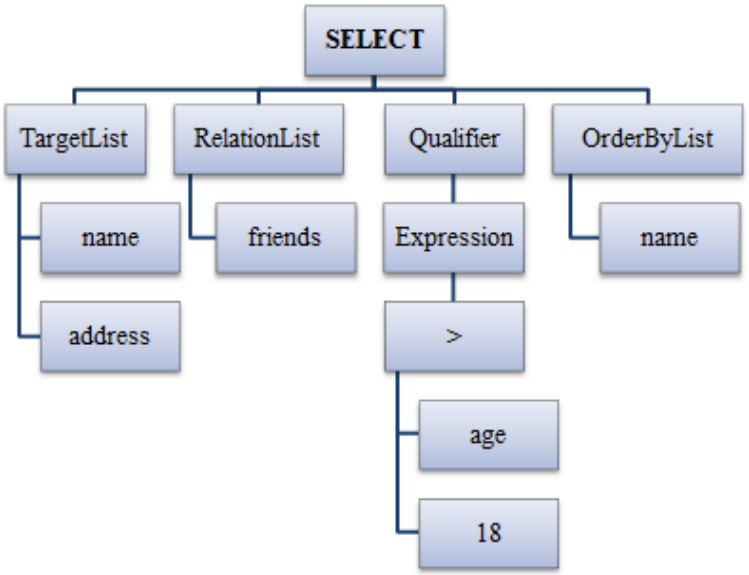

Figura 1: – *Parse tree* correspondente à query anteriormente referida.

4.1.3 Reescrita

Aplica todas as *rules* presentes no *Rules System* à *query tree* de modo a simplificá-la para a fase seguinte, do ponto de vista do planeador/optimizador.

4.1.4 Planeador/Optimizador

O objectivo desta fase é encontrar, se possível, o plano óptimo de execução, isto é, o plano aplicável à consulta com menor custo, a partir da query tree simplificada. Para gerar todos os planos possíveis, o PostgreSQL procura utilizar o *Genetic Query Optimizer*, que recorre a uma combinação de optimização heurística com um algoritmo genético não exaustivo e não determinista.

4.1.5 Execução

O plano de execução escolhido é processado recursivamente para obter cada uma das linhas do resultado. À medida que os nós do plano vão sendo chamados, estes têm de devolver um ou mais tuplos ou a informação de que já terminou. Para cláusulas do tipo `SELECT`, o executor devolve todos os tuplos retornados. Para o `INSERT` ou `UPDATE` cada tuplo é inserido ou actualizado, respectivamente, na tabela correspondente. Para o

`DELETE` devolve os identificadores dos tuplos a serem removidos e remove-os.

4.2 Algoritmos suportados

De acordo com a query, o optimizador escolherá sempre aqueles algoritmos que estima minimizarem os custos na obtenção dos resultados. De seguida apresenta-se um conjunto desses algoritmos.

4.2.1 No SELECT

- ***Sequencial Scan***

Sendo o algoritmo mais básico, consiste apenas numa pesquisa linear, percorrendo todos os tuplos e adicionando-o ao resultado caso satisfaça as condições da query.

- ***Index Scan***

Faz a pesquisa em índices, quer sejam primários ou secundários, para as operações de comparação <, <=, =, >= e >. Assim sendo, poderá não ter de percorrer todos os tuplos da tabela se for dado um intervalo. Por outro lado, o resultado estará ordenado segundo a ordem do índice.

- ***Bitmap Index Scan***

Recorrendo à indexação múltipla, utiliza os diferentes índices criados numa tabela para optimizar e aumentar a velocidade das pesquisas.

- ***TID Scan***

Apesar de não ser utilizado frequentemente, este algoritmo aproveita-se do identificador único atribuído a cada tuplo e que pode ser acedido através do atributo *ctid*.

4.2.2 No JOIN

- ***Nested loop join***

Consiste em comparar cada tuplo da primeira relação com todos os tuplos da segunda e, caso as condições de junção sejam satisfeitas, esse par de tuplos é adicionado aos resultados.

- ***Merge join***

Após ordenar as relações de acordo com o atributo de junção, elas são percorridas de forma quase paralela e uma única vez.

- ***Hash join***

A segunda relação é carregada numa tabela de dispersão que utiliza como chave o *hash* dos atributos de junção, seguida de um varrimento sequencial da primeira relação cujo *hash* dos atributos de junção verifica ou não a existência de um tuplo correspondente na tabela de dispersão.

4.2.3 Na ordenação

Dependendo de se a dimensão dos dados cabe em memória ou não, a ordenação executa-se através dos algoritmos de *quicksort* ou *external sort-merge*, respectivamente, ou através de índices.

4.3 Avaliação de expressões complexas

De modo a avaliar as expressões, o PostgreSQL pode analisá-las guardando o seu resultado em relações temporárias na base de dados, armazenando os resultados de cada operação intermédia e utilizando-os em avaliações de operações nos níveis superiores (mecanismo de *Materialização*), ou passar os tuplos para as operações acima durante a execução da operação corrente (mecanismo de *Pipelining*). Todavia, essa decisão cabe ao próprio sistema, nomeadamente ao planeador/optimizador, e não ao utilizador.

4.4 Parametrização

O utilizador pode activar ou desactivar a utilização de determinados algoritmos através dos seguintes métodos:

- `enable_bitmapscan` (`boolean`)

- `enable_hashagg` (`boolean`)
- `enable_hashjoin` (`boolean`)
- `enable_indexscan` (`boolean`)
- `enable_indexonlyscan` (`boolean`)
- `enable_material` (`boolean`)
- `enable_mergejoin` (`boolean`)
- `enable_nestloop` (`boolean`)
- `enable_seqscan` (`boolean`)
- `enable_sort` (`boolean`)
- `enable_tidscan` (`boolean`)

4.5 Visualização dos planos

O utilizador pode visualizar o plano de execução de uma dada query através do comando

```
EXPLAIN [ANALYZE] [FORMAT] query
```

Para cada etapa do plano, é mostrado o tipo de operação, o custo estimado e, se a opção `ANALYZE` estiver definida, o custo real (exigindo, nesse caso, a execução da query).

Por outro lado, é possível mostrar o resultado em diferentes formatos (`TEXT, XML, JSON` ou `YAML`). Para isso, deve-se indicar o formato desejado na cláusula `FORMAT`.

4.6 Estimativas

O comando `ANALYZE` permite actualizar as informações do catálogo do sistema, onde constam dados como a quantidade de tuplos por tabela. Esta é uma funcionalidade importante, uma vez que, ao actualiza-lo, podem ser feitas boas estimativas que serão úteis na construção dos planos.

O comando pode ser executado para todas as tabelas do sistema ou apenas para uma em particular, devendo para esse efeito parametrizar o nome da tabela em questão.

5 Gestão de transações e controlo de concorrência

De forma a garantir a consistência dos dados, o PostgreSQL usa um modelo multi-versão - *Multiversion Concurrency Control* (*MVCC*). Este mecanismo consiste em criar uma nova versão dos dados para cada interrogação feita à base de dados.

O modelo MVCC é mais eficiente do que um mecanismo de concorrência baseado em *locks*, pois as instruções são executadas sobre a versão mais recente dos dados, garantido que a transação não visualiza dados inconsistentes provocados por outras transações concorrentes. Desta forma as leituras não provocam conflitos com as escritas de dados, pelo que leituras nunca bloqueiam escritas e vice-versa.

O PostgreSQL mantém esta garantia mesmo quando o nível de isolamento de transações mais restrito é usado, o nível *Serializable Snapshot Isolation* (SSI).

5.1 Transações

Para definir transacções em PostgreSQL, os comandos SQL que a definem têm que estar delimitados pelos comandos BEGIN e COMMIT. Neste sistema, cada declaração SQL é executada como uma transação, pois se o comando BEGIN não for explicitado, então cada declaração está de forma implícita rodeada por BEGIN e COMMIT.

Este sistema não permite a declaração de *nested transactions*, no entanto é possível simular este comportamento através da declaração de *savepoints* no interior de uma transacção. Estes pontos intermédios permitem que todos os comandos executados após a sua declaração sejam revertidos, através da declaração do comando ROLLBACK, restaurando assim o estado da transação para aquele em que se encontrava quando atingiu o *savepoint*.

5.2 Isolamento de Transacções

Em PostgreSQL é possível usar os quatro níveis de isolamento do standard *SQL*, sendo o modo *Serializable* o mais restrito, o qual garante que a execução de várias transações concorrentemente produz o mesmo resultado que a execução das transações por uma determinada ordem. Os restantes três níveis são definidos de acordo com certos fenómenos que não devem de ocorrer aquando da interação concorrente de transações, sendo estes o seguintes:

- ***Dirty read***: A transação consegue ler dados escritos por uma transacção concorrente que não esteja *comitted*.
- ***Nonrepeatable read***: A transação relê dados que leu previamente e deteta que esses mesmos dados foram modificados por outra transação que efetuou *comit* desde a leitura inicial.
- ***Phantom read:*** A transação volta a executar uma determinada *query* retornando um conjunto de tuplos que satisfazem a condição de pesquisa e deteta que esses tuplos foram modificados por outra transação que efetuou *commit* recentemente.

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

Tabela 3: – Descrição dos níveis de isolamento do SQL standard.

No entanto, internamente no sistema, apenas existem três níveis de isolamento distintos que correspondem aos níveis *Read Committed*, *Repeatable Read* e *Serializable*. Quando é selecionado o nível *Read Uncommitted* o que de facto é aplicado é o *Read Commited*, pelo que se obtém um isolamento mais restrito do que o selecionado. É de notar que tal é permitido pelo standard *SQL*, pois os quatro níveis de isolamento apenas definem o que não pode acontecer. Contudo, estes não definem o que deve de acontecer.
O *PostgreSQL* disponibiliza apenas estes 3 níveis, pois esta é a única forma de mapear o standard de níveis de isolamento para a arquitetura MVCC.

Para definir o nível da transacção deve-se usar o comando SET TRANSACTION. No entanto, este pode ser dispensando em detrimento de BEGIN TRANSACTION ou START TRANSACTION.

Tal como em muitos sistemas de base de dados, como o Oracle, no PostgreSQL o nível de isolamento pré-definido é o *Read Committed*, embora o nível predefinido no standard seja o *Serializable*.

5.2.1 *Read Committed Isolation Level*

Este é o nível usado por omissão no sistema PostgreSQL. A técnica usada para implementar este nível é semelhante à do Oracle, pois usa *snapshots* da base de dados aquando de uma *query* SELECT, em que a mesma apenas visualiza os tuplos que foram *comitted* por outras transações antes da sua execução.

Os comandos UPDATE, DELETE, SELECT FOR UPDATE, e SELECT FOR SHARE têm o mesmo comportamento que o SELECT mencionado em termos de pesquisa de tuplos, ou seja, apenas visualiza tuplos *comitted* aquando do início dos comandos. No entanto, um determinado tuplo alvo pode já ter sido atualizado por outra transação concorrente. Neste caso a transação que efetua a atualização fica em espera até que a primeira (caso ainda esteja em execução) efetue *commit* ou *roll back*. Se a primeira efetuar *roll back*, então as suas modificações deixam de ter efeito e a transação em espera pode efetuar as suas atualizações sobre o tuplo original.

Caso a primeira transação efectue *comit*, a segunda acaba por ignorar o tuplo se a primeira o apagar. Caso contrário, acaba por efetuar as modificações necessárias à versão atualizada do tuplo. A condição de pesquisa (cláusula WHERE) é reavaliada a fim de verificar que a versão atualizada do tuplo continua a verificar a condição de pesquisa.

5.2.2 *Repeatable Read Isolation Level*

Este nível de isolamento é semelhante ao anterior (*Read Comitted*), mas neste caso a *query* executada numa transação em R*epeatable Read* visualiza um *snapshot* aquando do início da transação e não do início da *query* corrente existente no interior da transação. Considerando sucessivos comandos SELECT numa única transação, estes visualizam os mesmos dados, pois após o início da sua transação não é possível visualizar dados *committed* por outras transações a executar concorrentemente.

Sendo este nível mais restrito, é necessário que as aplicações que o usem, estejam preparadas para reexecutarem as transacções devido a falhas de serialização que possam ocorrer.

Os comandos UPDATE, DELETE, SELECT FOR UPDATE, e SELECT FOR SHARE têm o mesmo comportamento que o comando SELECT, sendo o comportamento de espera semelhante ao do nível anterior para o caso em que uma transação (T1) tenta alterar um tuplo já alterado por outra (T2). A diferença no comportamento reside quando T2 sucede na sua execução, ou seja, efetua *commit*, pelo que T1 aplica *roll back* e a mensagem de erro "**ERROR: could not serialize access due to concurrent update**".

É de notar que apenas é necessário abortar e reexecutar transacções que efectuam alterações, e que transacções de leituras apenas nunca geram conflitos.

5.2.3 *Serializable*

Este nível de isolamento tem o mesmo comportamento que o nível anterior, mas neste caso são monitorizadas as condições que podem provocar a execução inconsistente de transações concorrentes serializáveis, com qualquer ordem de serialização possível entre essas transações. Esta monitorização não introduz qualquer bloqueio para além dos introduzidos pelo modo *Repeatable Read*, mas introduz algum peso na sua execução, tal como uma falha de serialização aquando da deteção de condições que podem provocar anomalias de serialização.

A fim de garantir verdadeira serialização, o PostgreSQL utiliza *predicate locking*, que consiste em manter um *lock* que permite determinar quando é que uma escrita pode causar impacto no resultado de uma leitura anterior efetuada por uma transacção concorrente. Estes *locks* não causam nenhum bloqueio nas transações, pelo que não originam *deadlocks*, e têm como objetivo detetar dependências entre transações serializáveis concorrentes, que em certas combinações de serialização podem originar falhas.

Este nível de isolamento é o mais restrito. A sua monitorização de operações de leitura e escrita têm um custo, tal como a reinicialização das transações terminadas devido a falhas de serialização, que é equilibrado face ao custo e bloqueios associados a *locks* explícitos.

5.3 *Locks*

O PostgreSQL disponibiliza a utilização de *locks*, em alternativa ao modelo multi-versão MVCC, para o controlo de transações concorrentes. O sistema de *locks* tem duas granularidades, já que estes podem ser efetuados sobre tabelas ou tuplos. Para além destas dois níveis de granularidade é ainda possível usar *locks* ao nível de páginas (partilhados ou exclusivos) para controlar escritas e leituras sobre a tabela de paginação na *pool* partilhada.

Este sistema disponibiliza a utilização da tabela do sistema *pg_locks*, que permite aceder a informação sobre os *locks* aplicados por transações correntes na base de dados. Esta tabela contém uma linha por cada objeto bloqueado, modo de bloqueio, e transação relevante, pelo que um determinado objeto bloqueado pode aparecer várias vezes, caso

múltiplas transações possuam um *lock* ou estejam em espera para o obter, sobre esse objeto.

Por oposição ao PostgreSQL, o *Oracle* não implementa nenhum mecanismo semelhante ao MVCC, pelo que realiza o controlo de transações concorrentes através de *locks*. Neste sistema existem também as duas granularidades enunciadas para o PostgreSQL, sendo a de nível de tuplos usada por omissão.

5.3.1 Table-level Locks

Para aplicar explicitamente qualquer um dos modos de bloqueio que aqui serão mencionados é necessário usar o camando LOCK:

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

Em que ***lockmode*** é um de:

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

Todos os modos de bloqueio são *locks* do nível tabela, independentemente da existência da palavra "row", pois os nomes dos modos de bloqueio devem-se a motivos históricos.

A grande diferença entre dois modos de bloqueio reside no conjunto de modos de bloqueio conflituosos em cada um, pois duas transações não podem obter, ao mesmo tempo, locks cujos modos gerem conflitos sobre a mesma tabela. Por sua vez, modos de bloqueio que não sejam conflituosos entre si podem ser atribuídos a várias transações.

Sempre que um *lock* seja adquirido, este é mantido até ao final da transacção. No entanto se um *lock* for adquirido após a declaração de um *savepoint*, então este é imediatamente libertado caso seja efectuado um *roll back* para esse *savepoint*.

Para cada um dos modos de bloqueio seguintes, são especificados os modos de conflito tal como os comando que os adquirem.

- **ACCESS SHARE**

Conflito com o modo ACCESS EXCLUSIVE.

Comandos: SELECT

No geral, este modo é obtido por qualquer comando que apenas efectue leituras sem modificar os dados.

- **ROW SHARE**

Conflito com os modos EXCLUSIVE e ACCESS EXCLUSIVE.

Comandos: SELECT FOR UPDATE e SELECT FOR SHARE.

- **ROW EXCLUSIVE**

Conflito com os modos SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, e ACCESS EXCLUSIVE.

Comandos: UPDATE, DELETE, e INSERT.

No geral, este modo de bloqueio é obtido por qualquer comando que modifique dados da tabela.

- **SHARE UPDATE EXCLUSIVE**

Conflito com os modos SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE,EXCLUSIVE, e ACCESS EXCLUSIVE.

Comandos: VACUUM (sem o parâmetro FULL), ANALYZE, CREATE INDEX CONCURRENTLY, e algumas formas de ALTER TABLE.

Este modo protege uma tabela de alterações concorrente sobre o *schema* e de execuções VACUUM.

- **SHARE**

Conflito com os modos ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE,EXCLUSIVE e ACCESS EXCLUSIVE.

Comandos: CREATE INDEX (sem o parâmetro CONCURRENTLY)

Este modo previne alterações concorrentes dos dados da tabela especificada.

- **SHARE ROW EXCLUSIVE**

Conflito com os modos de bloqueio ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE.

Comandos: Este modo não é adquirido automaticamente por nenhum comando em PostgreSQL.

Este modo previne alterações concorrentes dos dados da tabela especificada, sendo auto-exclusivo, ou seja, nenhuma outra sessão pode obter este modo no mesmo espaço de tempo.

- **EXCLUSIVE**

Conflito com os modos ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE.

Comandos: Este modo não é adquirido automaticamente por nenhum comando em PostgreSQL.

Este modo de bloqueio permite apenas locks ACCESS SHARE concorrentes, ou seja, apenas leituras sobre a tabela podem proceder em paralelo com uma transação que possua este mesmo modo de bloqueio.

- **ACCESS EXCLUSIVE**

Conflito com todos os modos de bloqueio.

Comandos: ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER e VACUUM FULL.

Este modo de bloqueio garante que a transação que o obtém é a única a aceder à tabela, sendo este o modo o selecionado por omissão para as declarações LOCK TABLE que não especifiquem explicitamente o seu modo de bloqueio.

Exemplo de um *lock* no modo SHARE EXCLUSIVE sobre a chave primária de uma tabela aquando de uma operação de remoção:

```
BEGIN WORK;

LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
```

```
DELETE FROM films_user_comments WHERE id IN

   (SELECT id FROM films WHERE rating < 5);

DELETE FROM films WHERE rating < 5;

COMMIT WORK;
```

5.3.2 *Row-level Locks*

Para além dos bloqueios a nível de tabelas, existem ainda bloqueios a nível de tuplos que podem ser de dois tipos:

- **Exclusivos**: um *lock* exclusivo na granularidade tuplo é automaticamente adquirido sobre um determinado tuplo quando este é atualizado ou apagado, sendo o *lock* mantido até a transação executar *commit* ou *roll back*.
  Os *locks* desta granularidade não afetam *queries* sobre os dados, sendo apenas bloqueantes apenas quando solicitadas escritas por diferentes transacções sobre os mesmos tuplos.
  Para adquirir um *lock* exclusivo nesta granularidade sobre um tuplo sem ter de o modificar, é necessário efetuar a seleção do tuplo usando: **SELECT FOR UPDATE**.

- **Partilhados**: para adquirir *locks* partilhados na granularidade de tuplos é necessário usar a selecção **SELECT FOR SHARE** para o tuplo em causa.
  Um lock partilhado não impede outras transações de obter o mesmo lock partilhado. Contudo, não é permitido a nenhuma transação realizar atualizações, remoções ou *locks* exclusivos sobre um tuplo, existindo uma transação com um *lock* partilhado sobre esse mesmo tuplo.

5.3.3 *Deadlocks*

Através do uso de *locks* é de esperar que a probabilidade de ocorrência de *deadlocks* seja maior. Para tal, basta que duas transações concorrentes tenham *locks* atribuídos, em que cada uma das transações esteja em espera pelos *locks* que a outra possui.

Consideremos as duas transações concorrentes seguintes, as quais apresentam uma enumeração nas instruções correspondente à ordem de execução das mesmas:

```
 Transação T1
(1) UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;
(4) UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
 Transacção T2
(2) UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;
(3) UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;
```

1. A transação T1 obtém um *lock* ao nível tuplo sobre o tuplo especificado pelo número de conta **acctnum=11111**.
2. Entretanto a transação T2 inicia, cuja instrução obtém um *lock* ao nível tuplo sobre o tuplo especificado pelo numero de conta **acctnum=22222**;
3. No entanto a instrução seguinte a executar é ainda em T2, a qual requer um *lock* de nível tuplo existente sobre o tuplo com número de conta **acctnum=11111**, pelo que a transação T2 fica em espera até à libertação desse mesmo *lock*.
4. T1 executa a sua segunda intrução que requer um *lock* de nível tuplo também existente, sobre o número de conta **acctnum=22222**, logo fica em espera ate que T2 liberte esse *lock*. No entanto, T2 está em espera pela libertação do lock que T1 possui.

Assim ocorre uma situação de *deadlock*, que o PostgreSQL deteta, abortando uma das transações.

5.3.4 *Advisory Locks*

Para além dos *locks* enunciados, o PostgreSQL disponibiliza a criação de *advisory locks*, os quais têm significados específicos de acordo com as aplicações que os usam. Existem duas formas de adquirir estes tipos de *locks* em PostgreSQL:

- **Nível da sessão**: quando um *advisory lock* é adquirido neste nível, este mantém-se até que seja explicitamente libertado ou que a sessão termine.
  Contrariamente aos *locks* habituais, estes tipo de *advisory locks* não respeitam a semântica das transações. Por exemplo, um *lock* que seja obtido por uma transação que posteriormente efetua *roll back*, será mantido pela mesma.

- **Nivel da transação**: este tipo de *advisory locks* são libertados automaticamente no final da transação, sem a necessidade de explicitar a operação para os libertar.

É necessário ter cuidado em alguns casos em que se usam *adisory locks*, especialmente em *queries* que envolvem ordenação e a cláusula LIMIT, em que para esta última pode não está garantida a sua execução antes do *lock* da instrução ser executado. Isto pode levar a que, implicitamente, existam *locks* inesperados a serem efectuados sobre os recursos, que no final não são devolvidos, pelo que será necessário consultar a vista ***pg_locks*** para explicitamente terminar esses *locks* inesperados.

5.4 Consistência e durabilidade

É possível assegurar consistência dos dados através do uso de transações serializáveis ou com a utilização explícita de *locks* bloqueantes.

- **Consistência através de transacções serializáveis**: se o nível se isolamento de transações *Serializable* for usado para todas as escritas e para todas as leituras que necessitem de uma visualização consistente dos dados, então não necessário efetuar alguma coisa para garantir essa consistência.
- **Consistência através de *locks* explícitos**: Quando escritas não serializáveis são permitidas, para garantir a validade de um tuplo e o proteger de atualizações concorrentes, é necessário usar SELECT FOR UPDATE, SELECT FOR SHARE ou um LOCK TABLE apropriado.

  O modo *Read Committed* deve ser usado quando se pretende prevenir modificações concorrentes através do uso de *locks* explícitos, pois no caso do modo *Repeatable Read* é necessário ter atenção porque é possível obter *locks* antes da execução das *queries.*

Conclui-se que usando serialização é a melhor forma de garantir isolamento, uma vez que as transações são automaticamente abortadas (*rolled back*) quando existe uma falha de serialização.

O utilizador pode ainda configurar uma transação para verificar a integridade da base de dados no fim da transação ou entre operações/*queries*, respetivamente através dos parâmetros DEFERRED e IMMEDIATE aquando do comando SET CONSTRAINT.

6 Suporte para bases de dados distribuídas

O PostgreSQL não possui suporte nativo para bases de dados distribuídas. Ainda assim, disponibiliza um conjunto de mecanismos de replicação baseados numa arquitetura cliente/servidor para suportar alta disponibilidade, balanceamento de carga (*load-balancing*) e replicação de dados. Existe um servidor primário (*master*) e um ou mais servidores secundários (*slaves*), que a qualquer momento podem assumir o papel de primário em caso de falha deste. Estas soluções baseiam-se igualmente no uso de WAL (*Write-Ahead Logs*), como se mostra na imagem abaixo:

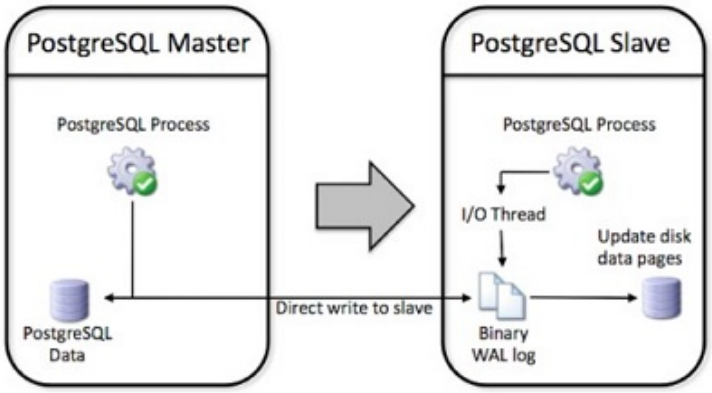

Figura 2: – Imagem ilustrativa do funcionamento do mecanismo de replicação do PostgreSQL.

Todos os objetos e dados (incluindo o esquema do *master*) são gravados no WAL diretamente na máquina *slave* para garantir a segurança (evitando a perda completa dos dados em caso de uma falha do *master*). Os WAL também garantem que nenhuma transação é confirmada no *master* até que tenha ocorrido um *write* com sucesso no seu registo WAL.

6.1 *Log-Shipping Standby Servers*

O *Log-Shipping* é uma técnica que permite que os servidores secundários sejam atualizados com as alterações do servidor principal, que envia as alterações do WAL para os servidores secundários em segmentos de 16MB, sendo a sua transferência sobre a rede fácil e sem custos muito elevados.

A transferência do WAL para os servidores secundários é efetuada de forma assíncrona, o que abre uma janela temporal na qual transações que ainda não foram transferidas po-

dem ser perdidas, no caso de falha do servidor *master*. Os *slaves* vão executando as operações que recebem no WAL.

6.1.1 *Streaming Replication*

Este mecanismo de replicação surgiu com a versão 9.0 do PostgreSQL e permite a replicação dos dados de um servidor para um ou mais servidores *standby*.

Essa replicação é feita através de *streams* de registos WAL de modo assíncrono, permitindo que as réplicas estejam o mais atualizadas possível, já que recebem as alterações ao WAL assim que são geradas, sem terem de ficar à espera que o ficheiro do WAL seja preenchido. Por ser um modo assíncrono, existe assim uma maior velocidade nas transações, mas existe a possibilidade de se perderem dados em caso de falha.

6.1.2 *Cascading Replication*

Esta funcionalidade permite que algumas réplicas, ao receberem atualizações ao WAL, também possam enviar essa informação para outras réplicas, servindo como servidor de reencaminhamento.

Esta abordagem permite reduzir o número de conexões ao servidor principal, diminuindo a sua carga. Cada réplica pode-se ligar a várias réplicas para as quais envia as atualizações recebidas, mas apenas se pode ligar a uma réplica, ou diretamente ao primário, para receber as atualizações. Quando uma réplica que serve como intermediária para outras é promovida a primária, esta termina imediatamente as ligações que tem com outras réplicas, para quem reencaminha as atualizações.

6.1.3 *Synchronous Replication*

Na versão 9.1 do PostgreSQL surgiu a opção de replicação síncrona. Esta técnica permite que se confirme que todas as alterações realizadas por uma transação foram transferidas para um servidor *standby*. Neste caso, cada *commit* de uma transação de escrita irá esperar até receber confirmação que o *commit* foi escrito no *log* de transações no disco de ambos os servidores (primário e *standby*). Desta forma, a única maneira de os dados se perderem ocorre apenas se ambos os servidores falharem ao mesmo tempo.

Este mecanismo fornece uma confiança maior ao utilizador ao garantir que as alterações não serão perdidas caso o servidor principal falhe, mas têm o acréscimo do tempo que tem de ficar à espera que a réplica execute as operações.

6.2 *Failover*

No caso de falha do servidor principal, o servidor secundário é promovido a principal e tem de executar os procedimentos de *failover*. Se for o servidor secundário a falhar este apenas tem de iniciar o processo de recuperação. Se não for possível ao servidor secundário reiniciar, deve ser criado um novo servidor secundário.

6.3 *Hot Standby*

O Hot Standby é um mecanismo que permite que o servidor consiga responder a *queries* apenas de leitura, enquanto ainda está no processo de recuperação. Permite igualmente que, após a recuperação do sistema, as *queries* que estão a correr não tenham de ser interrompidas, e que nenhuma conexão tenha que ser fechada.

6.4 Outras soluções

Para além destes mecanismos, e dado que o PostgreSQL é um sistema *open source* e facilmente extensível, um conjunto de empresas criaram algumas soluções comerciais com capacidades integradas de *failover*, replicação e *load balancing*. A tabela seguinte apresenta algumas delas, bem como os seus métodos de replicação e de sincronismo:

Extensão	Método de Replicação	Sincronismo
PgCluster	Master-Master	Síncrono
pgpool-I	Statement-Based Middleware	Síncrono
Pgpool-II	Statement-Based Middleware	Síncrono
slony	Master-Slave	Assíncrono
Bucardo	Master-Master, Master-Slave	Assíncrono
Londiste	Master-Slave	Assíncrono
Mammoth	Master-Slave	Assíncrono
rubyrep	Master-Master, Master-Slave	Assíncrono

Tabela 4: – Principais soluções comerciais para extensão do PostgreSQL no suporte a bases de dados distribuídas.

7 Outras características do PostgreSQL

O PostgreSQL contém muitas outras características interessantes de analisar. Todavia, devido à sua numerosidade e à falta de documentação sobre algumas delas, iremos abordar as mais relevantes, de forma breve, nomeadamente o suporte de XML, o suporte de linguagens procedimentais e aspetos de segurança.

7.1 Suporte de XML

O PostgreSQL suporta documentos XML, cuja grande vantagem é a possibilidade de armazenar dados de forma estruturada. Para permitir o suporte para tipos de dados XML, é necessário instalar o PosgreSQL com a biblioteca libxml. Para o fazer, durante a instalação o utilizador tem que colocar o comando ***configure –with-libxml***.

Após esta configuração, o utilizador pode usar os comandos referentes ao tipo de dados XML. Por exemplo, para verificar se um determinado documento é do tipo XML, utiliza-se o comando apresentado abaixo, onde xmldoc é o documento que desejamos verificar.

xmldoc IS DOCUMENT

Para criar um documento XML a partir de uma string utiliza-se o comando:

XMLPARSE ({ DOCUMENT | CONTENT } value)

E para realizar a operação inversa (criar uma string a partir de um documento XML), utiliza-se o comando:

XMLSERIALIZE ({ DOCUMENT | CONTENT } value AS { character | character varying | text })

De forma semelhante, o PostgreSQL suporta também o tipo de dados JSON.

7.2 Linguagens Procedimentais

O PostgreSQL permite que sejam adicionadas funções definidas pelos utilizadores noutras linguagens para além do SQL. Essas linguagens são chamadas de linguagem procedimentais.

Para estas funções, o interpretador do SGBD não tem informação que lhe permita interpretar esse código fonte. Em vez disso, a tarefa é passada para uma rotina de tratamento (*handler*) que conhece os detalhes da linguagem.

Na distribuição *standard* do PostgreSQL estão disponíveis as linguagens procedimentais PL/pgSQL, PL/TCL, PL/Perl e PL/Python. Existe outras linguagens que não estão incluídas na distribuição base mas que podem ser utilizadas, sendo que também é permitido ao utilizador definir uma nova linguagem.

7.3 Segurança e Autenticação

O PostgreSQL permite a criação de uma ou mais bases de dados numa instância de um servidor, cada uma composta por uma lista de utilizadores com diferentes níveis de acesso. Esses diferentes níveis de acesso são denominados de *roles*. Quando um utilizador tenta aceder à base de dados, é pedido que se autentique e, dependendo do seu *role*, são-lhe dadas as permissões e privilégios a que tem acesso.

Estas informações são armazenadas no ficheiro ***pg hba.conf***, que guarda todos os utilizadores que estão autorizados a aceder à base de dados e os privilégios associados.

Para garantir a segurança no acesso à bases de dados, o PostgreSQL suporta vários mecanismos de autenticação do cliente, entre os quais: Trust, Password, GSSAPI, SSPI, Kerberos, Ident, Peer, LDAP, RADIUS, Certificate e PAM. A existência de um número tão elevado de formas de autenticação prende-se novamente com o o carácter *open source* deste sistema. Para uma descrição mais aprofundada de cada um deles aconselha-se a consulta do manual do PostgreSQL.

Existe uma separação entre os nomes dos utilizadores da base de dados e os utilizadores do sistema operativo no qual o servidor corre. Se todos os utilizadores de um servidor têm contas nessa máquina, faz sentido atribuir *usernames* para a BD que correspondam aos *usernames* do SO. No entanto, como são permitidas conexões remotas, o servidor aceita várias conexões de utilizadores que não têm conta no SO dessa máquina, o que já não permite a correspondência entre os *usernames*.

O PosgreSQL suporta igualmente a encriptação das comunicações entre o cliente e o servidor através de SSL. Para isso, é necessário que o OpenSSL esteja instalado tanto no cliente como no servidor e que o parâmetro *ssl* seja especificado no ficheiro de configuração ***postgresql.conf***.

8 Comparação com o Oracle 11g

Esta secção visa comparar o sistema estudado com o Oracle 11g. Para cada um dos principais tópicos abordados, são comparadas as características dos dois sistemas.

8.1 Armazenamento e Estrutura de Ficheiros

Nesta matéria, o PostgreSQL apresenta algumas semelhanças, mas também algumas diferenças em relação ao Oracle 11g.

Enquanto que o PostgreSQL utiliza o sistema de ficheiros do sistema operativo, o Oracle utiliza

o seu próprio sistema de ficheiros. Apesar de ter uma complexidade maior, o sistema de ficheiros do Oracle pode ganhar a nível de desempenho, caso a performance do *file system* do sistema operativo seja mais fraca do que o desejado.

Tanto o PostgreSQL como o Oracle têm o seu próprio sistema de gestão de *buffers*, ambos utilizando a técnica LRU (*Least Recently Used*).

8.2 Indexação e *hashing*

Os dois sistemas são bastante semelhantes em termos de indexação, principalmente no que se refere aos índices mais usados, baseados em árvores B+. No entanto o PostgresSQL suporta tipos de indexação mais sofisticados, como o GiST e o GIN, que não são suportados pelo Oracle. A seguinte tabela resume as principais diferenças entres os dois sistemas:

Funcionalidade	**Oracle**	**PostgreSQL**
Índices Hash	Sim	Sim
Índices Árvores B+	Sim	Sim
Indices GiST	Não	Sim
Indices Bitmap	Sim	Não
Indices Parciais	Sim	Sim
Indices Invertidos	Sim	Sim
Combinação de Índices	Não	Sim

Tabela 5: – Comparação dos mecanismos de *hashing* suportados pelo PostgreSQL e pelo Oracle.

O PostgreSQL assume também que, devido ao seu poder de otimização, escolhe sempre o melhor plano, impedindo por isso o utilizador de forçar o uso de um determinado índice para a execução de uma *query*.

8.3 Optimização de perguntas

O PostgreSQL investe no processamento de consultas de grandes dimensões, ao contrário do Oracle, que tem algumas limitações devido às heurísticas usadas.

O Oracle tem uma gama menor de algoritmos, pois a combinatória entre tabelas é menor, não compensando suportar mais algoritmos. Ao fazer a junção de tabelas, o PostgreSQL apresenta mais algoritmos de otimização que o Oracle, o que pode ser prejudicial para a sua performance. A desvantagem de suportar muitos algoritmos prende-se com as combinatórias que surgem quando se pretende fazer uma junção. Existe a possibilidade de o PostgreSQL poder encontrar um plano melhor que o escolhido pelo Oracle, mas pode também gastar mais tempo para chegar a esse plano.

8.4 Concorrência

O PostgreSQL é semelhante ao Oracle neste tema. Suporta os quatro tipos de isolamento, utiliza o mecanismo de *Snapshot Isolation*, permite declarar *savepoints* e possui a possibilidade de obter explicitamente *locks*. Para além disto, o PostgreSQL oferece também *advisory locks*.

8.5 Base de dados distribuídas

O Oracle oferece um suporte muito mais alargado nesta área do que o PostgreSQL, cujas funcionalidades para suporte à replicação foram apenas introduzidas em versões mais recentes. Tal como o Oracle, o PostgreSQL suporta replicação de dados mas apenas numa arquitectura de servidor *master/slaves*, enquanto que o Oracle permite também a replicação *master/master*.

9 Referências

[1] *Postgresql 9.3.4 documentation*, http://www.postgresql.org/docs/9.3/static/release-9-3-4.html.

[2] *Wikipedia PostgreSQL*, http://pt.wikipedia.org/wiki/PostgreSQL.

[3] Silberschatz, A., Korth, H. F., and Sudarshan, *S. Database System Concepts*, McGraw-Hill Hightstown, 6th edition, 2010.

[4] *Wikipedia SQL*, http://en.wikipedia.org/wiki/SQL.

[5] *Comparing MySQL and Postgres 9.0 Replication*, http://www.theserverside.com/feature/Comparing-MySQL-and-Postgres-90-Replication.