FACULDADE DE
CIÊNCIAS E TECNOLOGIA
**UNIVERSIDADE NOVA** DE LISBOA

# Análise ao PostgreSQL

## Sistemas de Bases de dados

31/05/2014

Grupo 23 composto por:

- Filipa Ferreira (Nº 41737)
- Luís Antunes (Nº 41809)
- Miguel Aniceto (Nº 41805)

# Índice

# Introdução

Neste documento serão observadas as características principais do Sistema de Base de Dados PostgreSQL. A versão em análise é a 9.3, disponibilizada a partir do dia 9 de Setembro de 2013.

O PostgreSQL é um sistema de gestão de base de dados objecto-relacional (ORDBMS) open-source, o PostgreSQL não é propriedade de uma empresa ou pessoa. Dado que o PostgreSQL é um projeto *open-source*, este é principalmente gerido por uma comunidade de voluntários, e trabalhadores a full-time cujo salário é pago através de empresas patrocinadoras.

## Introdução Histórica

O PostgreSQL é um dos resultados de uma ampla evolução que se iniciou com o projeto *Ingres*, sendo que ambos foram desenvolvidos por uma equipa da Universidade de Berkeley na Califórnia.

Originalmente, o PostgreSQL não suportava a linguagem SQL (consulta estruturada) mas suportava a linguagem de consulta QUEL que foi utilizada até 1994. Só em 1995 a linguagem QUEL foi substituída por um sub-set extendido do SQL.

Actualmente o PostgreSQL é utilizado por diversas entidades de relevo, entre as quais a Cisco, ADP, NOAA e diversas outras. Além disso, o PostgreSQL é também usado em vários ambientes de produção complexos.

## Contexto

Este projeto foi desenvolvido no contexto da disciplina de Sistema de Base de Dados (SBD) e tem como objetivo efetuar a análise, o mais completa possível, de um sistema de gestão de base de dados. Desta forma, neste documento, apresentamos toda a matéria que foi lecionada durante o semestre, sobre o nosso tema.

Assim, o tema escolhido foi o PostgreSQL porque, este é um sistema de base de dados bastante utilizado tanto em ambientes de pequena como de larga escala, apresenta diversas features relativamente a escalabilidade e controlo de concorrência, bem como muitas outras que vão ser analisadas nos capítulos seguintes.

# Armazenamento e File Structure

Neste capítulo pretendemos apresentar a forma como os ficheiros são armazenados e como a sua estrutura é organizada.

Relativamente ao armazenamento de dados, sabemos que os ficheiros são divididos em blocos e que cada bloco contém uma variedade de tuplos.

Assim, o PostgreSQL suporta de forma eficiente grandes tamanhos de informação nas tabelas, apresentando assim, um resumo da capacidade de armazenamento deste sistema de base de dados:

| Limite | Valor |
|---|---|
| Tamanho máximo da base de dados | Ilimitado |
| Tamanho máximo de uma tabela | 32 TB |
| Tamanho máximo de um tuplo | 1.6 TB |
| Tamanho máximo por campo | 1 GB |
| Máximo de linhas por tabela | Ilimitado |
| Máximo de colunas por tabela | 250-1600 dependendo do tipo de colunas |
| Máximo de indexes por tabela | Ilimitado |

**Tabela 1: Capacidade de Armazenamento**

## Sistema de Ficheiros

Nesta subsecção pretendemos explicar como se encontra organizado o sistema de ficheiros, os tuplos e as suas tabelas. Bem como, a gestão de buffers, as partições suportadas e o suporte a clustering.

O sistema PostgreSQL implementa o uso de sistemas de ficheiros, não utilizando o do sistema operativo. Assim, todos os dados são armazenados apenas numa diretoria do cluster, designado *PGDATA*, onde a localização mais conhecida é */var/lib/pgsql/data*. Onde é possível existirem vários clusters na mesma máquina.

Desta forma, a organização da diretoria é efetuada por várias subdiretorias e controladores, como podemos verificar na seguinte tabela.

| Item | Descrição |
|---|---|
| PG_VERSION | Arquivo que contém o número da versão principal do PostGreSQL. |
| base | Sub-directoria que contém as sub-directorias por base de dados. |
| global | Sub-directoria que contém as tabelas por todo o cluster, como pg_database. |
| pg_clog | Sub-directoria que contém os dados sobre o estado da transação. |
| pg_subtrans | Sub-directoria que contém os dados sobre o estado da subtransação. |
| pg_tblspc | Sub-directoria que contém os vínculos simbólicos para os espaços das tabelas. |
| pg_xlog | Sub-directoria que contém os arquivos do *WAL*, ou seja, registo prévio da escrita. |
| postmaster.opts | Arquivo que contém as opções de linha do comando com as quais o postmaster foi inicializado da ultima vez. |
| postmaster.pid | Arquivo de bloqueio que contém o PID corrente do postmaster, e o ID do segmento de memória partilhada. |

**Tabela 2: Subdiretorias e Controladores**

*Nota:* WAL (Write Ahead Log) é utilizado para proporcionar atomicidade e durabilidade em sistemas de base de dados. Sendo que, todas as modificações são guardadas num log antes de serem aplicadas.

Eventualmente, para cada base de dados que se encontra no cluster, existe uma sub-directoria dentro de *PGDATA/base*, efetuando assim a referência a *OID* da base de dados no *pg_database*. Esta subdiretoria permite a localização dos arquivos desta base de dados, especialmente o armazenamento dos registos do sistema.

Desta forma, cada tabela e cada índice são guardados num arquivo individual, com o nome da tabela ou o *filenode* do índice que se encontra no *pg_class.relfilenode*.

Cada tabela e índice têm um *free space map*, que guarda as informações relativas ao espaço livre que se encontra disponível na relação. O *free space map* é guardado num arquivo designado através do filenode com o sufixo *_fsm*. Assim, as tabelas também têm um *visibility map* que permite controlar as páginas que são conhecidas por não terem tuplos vazios e são guardadas num *fork* com o sufixo *_vm.*

No caso de existir uma tabela em que as suas colunas têm entradas demasiado grandes, será associada a esta, uma tabela *TOAST* que tem como objectivo guardar os valores *out-of-line* e assim, é possível manter as linhas adequadas à tabela.

Em relação ao *tablespace* sabemos que a cada espaço livre definido pelo utilizador existe um link na diretoria *PGDATA/pg_tblspc*, que tem um apontador para a diretoria física da tabela. Posteriormente, na diretoria da tabela é possível encontrar uma

subdiretoria específica que tem as subdiretorias associadas aos elementos para cada base de dados. Assim, as tabelas e os índices são guardados nessa diretoria utilizando a nomenclatura *filenode*.

# Free Space Map

Cada *heap* e cada relação do índice (com exceção dos índices *hash*) têm *free space map* (*FSM*) que permitem manter o controlo do espaço que se encontra disponível na relação. Assim, este é guardado junto aos dados principais da relação numa relação *fork*.

Desta forma, o *FSM* é organizado de forma idêntica ao de uma árvore de páginas *FSM*. Sendo que, nestas páginas, o nível inferior da árvore permite guardar o espaço livre em cada *heap*/índice e cada uma dessas páginas têm a dimensão de um byte. No que toca ao nível superior da mesma árvore, este nível permite agregar todas as informações relativas aos níveis inferiores deste.

Cada página *FSM* é definida por uma árvore binária que guarda um vector com um byte por nó. Por sua vez, cada nó representa uma *head* de página ou uma página *FSM* do nível inferior. Em cada nó que não seja folha, o filho que seja maior é guardado e o máximo valor das folhas é guardado na sua raiz.

# Visibility Map

Cada relação apresenta uma *heap* que tem *visibility map* (*VM*) e, este acompanha as páginas que contêm apenas os tuplos que são conhecidos por serem visíveis por todas as operações que se encontram ativas. Este é guardado junto aos dados principais da relação numa relação *fork*.

O *VM* guarda, por cada página *heap*, apenas um bit. Um conjunto desses bits representa que todos os tuplos na página são conhecidos por serem visíveis a todas as transações. Ou seja, a página não contém tuplos que precisam de ser limpos.

# TOAST

O PostgreSQL utiliza um tamanho de páginas fixo, definido por 8 Kb, e não permite a extensibilidade dos tuplos por várias páginas, o que faz com que não seja possível guardar diretamente os valores que são muito grandes. Assim, para combater estas limitações, utiliza-se a técnica *TOAST* (*The Oversized-Attribute Storage Technique*) que permite aos dados serem comprimidos ou divididos por várias linhas físicas.

Nem todos os tipos de dados têm suporte para *TOAST*. Assim, para que seja possível suportar esta técnica, o tipo de dados tem de possuir uma representação de comprimento variável. Sendo que, a primeira palavra de 32 bits para qualquer valor guardado que contenha o comprimento total do valor em bytes.

Neste sentido, a técnica de *TOAST* tem quatro estratégias para permitir guardar as colunas. Sendo as estratégias apresentadas de seguida:

- *PLAIN* - Esta estratégia impede a compressão e o armazenamento *out-of-line*. Sendo esta a única estratégia para as colunas com o tipo de dados não particionados.
- *EXTENDED* - Esta estratégia permite a compressão e o armazenamento *out-of-line*, sendo este padrão para a maioria dos tipos de dados. Para esta estratégia, inicialmente é efetuada a compressão e caso a linha continue demasiado grande é, então, efetuado o armazenamento *out-of-line*.
- *EXTERNAL* - Em relação a esta estratégia, esta não permite efetuar a compressão mas permite o armazenamento *out-of-line*. A utilização desta estratégia faz com que as *substrings* das operações nas colunas dos tipos de dados *Text* e *Byte* sejam mais rápidas, porque estas operações são otimizadas para efetuar a pesquisa das partes necessárias do valor *out-of-line*, quando este não se encontra comprimido.
- *MAIN* - Por fim, esta estratégia permite efetuar a compressão mas não permite o armazenamento *out-of-line*.

Assim sendo, cada tipo de dados "fatiável" fornece a estratégia padrão para as colunas deste tipo de dados, mas a estratégia para uma coluna de uma determinada tabela pode ser alterada através do comando *ALTER TABLE SET STORAGE: OPTIONS*.

# Gestão de Buffers

O PostgreSQL utiliza um sistema de gestão de *buffers* que facilita a utilização dos *buffers* partilhados para melhorar a sua execução.

Desta forma, o número de *buffers* pode ser especificado se o valor de *shared_buffers* for alterado. Sendo que o valor ideal de *buffers* é aquele que permite dar os resultados pela cache à maioria dos pedidos, de forma a evitar a troca constante de páginas.

# Partições

A partição de tabelas refere-se a dividir uma grande tabela em partes físicas menores, dando assim alguns benefícios como melhorar o desempenho de consultas em determinadas situações, quando consultas ou atividades acedem a uma grande parte de uma partição, exclusões que são conseguidas através da adição/remoção de partições e quando os dados que raramente são utilizados são enviados para partes de armazenamento mais baratos. Em geral, os benefícios das partições são vantajosos quando a tabela é demasiado grande, ou seja, quando a tabela em si excede a memória física da base de dados.

O PostgreSQL suporta a partição de tabelas através das heranças das mesmas. Assim sendo, este sistema de base de dados tem duas formas de partição:

- *Range Partitioning* - A tabela é dividida em *ranges* definidas por uma coluna de *keys* ou por um conjunto de colunas, não existindo sobreposição de atributos designados a partições diferentes.
- *List Partitioning* - A tabela é dividida de forma a listar os *values-keys* que aparecem em cada partição.

Desta forma, uma dada partição é implementada através da criação da tabela principal e todas as restantes partições a herdaram. De seguida são criadas as várias tabelas que herdam a tabela principal e, por fim, adiciona-se as restrições para as tabelas de partições de maneira a que, seja possível definir os valores de chave permitidos.

*CHECK ( county ID ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ) )*

*CHECK ( outletID BETWEEN 100 AND 200 ) )*

Para cada partição deve ser criado um índice na chave ou outros índices que forem necessários. Caso a partição seja utilizada para ajudar a realizar as diferentes exigências, é utilizado o *measurement* da seguinte forma:

```
CREATE TABLE measurement_y2008m01 (
    CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
) INHERITS (measurement);
```

# Clustering

*Cluster* significa o agrupamento de uma tabela de acordo com um dado índice. E como tal, este tem o objetivo de instruir o PostgreSQL a agrupar uma tabela especifica baseada no índice especificado, como por exemplo:

```
CLUSTER table_name [ USING index_name ]
```

Desta forma, *clustering* é uma operação *one-time*, que quando uma tabela é atualizada, as alterações não estão agrupadas, ou seja, não é efetuada nenhuma tentativa para guardar as novas linhas ou atualizá-las na ordem do índice.

Quando a tabela em si é agrupada, este sistema de base de dados recorda por que o índice foi agrupado. O *table_name* reagrupa a tabela utilizando o mesmo índice anteriormente utilizado. Caso o *Cluster* não tenha nenhum parâmetro, este reagrupa todas as tabelas que anteriormente estavam agrupadas na base de dados atual que o utilizador possui, ou todas as tabelas se for considerado um superutilizador.

Por fim, quando uma tabela está a ser agrupada, um bloqueio de *Exclusive Access* é adquirido sobre ele. Com isto, é possível evitar que outras operações da base de dados executem na tabela até o *Cluster* estar concluído.

# Indexação e Hashing

A capacidade de indexação oferecida pelos Sistemas de Gestão de Bases de Dados permite um acesso mais rápido aos dados. Sem esta capacidade, a cada pesquisa seria necessário percorrer todos os tuplos de uma determinada tabela, sequencialmente, de modo a obter o resultado de uma query, o que se torna extremamente ineficiente no caso em que uma tabela tenha milhares (ou mesmo milhões) de tuplos.

Estes índices são criados através da especificação de um atributo (ou vários) da tabela, sobre diferentes estruturas, sendo todos estes dados especificados na instrução que permite a criação do índice. Assim, a criação de índices é feita a partir do seguinte comando:

CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [name] ON table [ USING method ]

( { column | ( expression ) }

[ COLLATE collation ] [ opclass ]

[ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )

[ WITH ( storage_parameter = value [, ... ] ) ]

[ TABLESPACE tablespace ]

[ WHERE predicate ]

Após a criação, o sistema é responsável pela atualização e utilização do mesmo, deixando o utilizador livre dessa responsabilidade. Ou seja, é o sistema que escolhe quando atualiza o índice de acordo com as modificações realizadas nas tabelas correspondentes.

## Estruturas Suportadas

Os índices são criados sobre diferentes tipos de estrutura, sendo esse tipo indicado através do parâmetro USING, no comando de criação do índice.

### B-Tree

Esta é a estrutura utilizada por omissão aquando da criação de um índice no PostgreSQL, pois é uma estrutura muito versátil. A B-Tree lida essencialmente com

*queries* de igualdade e desigualdade sobre dados que possivelmente possam estar ordenados, portanto, suportando os seguintes operadores: <, <=, =, >, >=, >. Além destes operadores mais "simples" e comuns, a B-Tree lida ainda com outro tipo de operações, como BETWEEN, IN, IS NULL ou IS NOT NULL.

## Hash

Como qualquer tipo de índice *hash*, esta estrutura suporta apenas operações de igualdade (=), sendo apenas preferida pelo optimizador quando se verificada este tipo de condição. Os índices *hash* utilizados são dinâmicos, pelo que não têm necessidade de efetuar um *rehash* completo (quando é feita uma expansão, os custos são divididos pelas várias operações, sendo os *rehash's* limitados aos buckets afetados).

Esta estrutura tem-se mostrado menos eficiente que a B-Tree (e daí a árvore ser utilizada por omissão pelo sistema), com custos de manutenção bastante maiores e incapacidade de recuperar de falhas no sistema, pois não são WAL-logged.

CREATE INDEX name ON table USING hash (column);

## GiST (Generalized Search Tree)

O GiST é uma árvore de procura balanceada que serve de base à implementação de outras estruturas de indexação baseadas em árvores, como as B-Tree. Estes índices são especialmente úteis para utilizadores experientes com determinados tipos de dados, pois permite "criar" tipos de dados e métodos de acesso personalizáveis sem que tenha um conhecimento detalhado de um sistema. Por defeito, o PostgreSQL inclui operadores de classe GiST para vários tipos de dados, inclusivamente dados geométricos (bidimensionais) e que suportam operadores para estes tipos de dados, como <<, >>, <@, |&>, etc.

CREATE INDEX name ON table USING gist (column);

Os índices GiST podem ainda ser utilizados na optimização de *queries* "nearest neighbours", como por exemplo, a procura dos 10 pontos mais próximos de um determinado ponto (x,y).

Combinado com o índice GIN (explicado em seguida), este é extremamente útil para acelerar pesquisas totais de texto.

## GIN (Generalized Inverted Index)

Os índices GIN são índices invertidos que lidam com valores que possuem mais que uma chave. Como foi dito, acima, combinado com o índice GiST, estes dois são utilizados frequentemente para pesquisas em texto, onde uma palavra é encarada como chave e a sua localização é o seu valor. Tal como o GiST, o GIN suporta várias estratégias de indexação definidas pelo utilizador.

CREATE INDEX name ON table USING gin (column);

Por defeito, o PostgreSQL inclui operadores para vetores unidimensionais, que suportam *queries* utilizando os operadores <@, @>, =, &&.

Dadas as semelhanças entre os dois últimos tipos de índice, é necessário algum tipo de comparação entre ambos e onde um seja mais vantajoso que utilizar que o outro. Os índice GIN são, regra geral, muito mais rápidos em operações de leitura e muito mais lentos no que toca a atualização das suas tabelas, o que faz desta estrutura ideal para dados estáticos, onde existem poucas atualizações das tabelas. Por sua vez, pelas razões acima mencionadas, os índices GiST, devido a serem mais eficientes na modificação (e na construção), são apropriados para utilização sobre dados dinâmicos.

# Índices Multi-Atributo

O PostgreSQL permite a criação de um índice sobre várias colunas, sendo estes índices suportados pela estruturas B-Tree, GiST e GIN. Um índice multi-atributo é útil para pesquisas que envolvem qualquer subconjunto das colunas representadas no índice, embora seja mais eficiente quando existem restrições nas colunas mais à esquerda. Estes índices são especialmente úteis quando utilizadas restrições de igualdade sobre o primeiro atributo (a coluna mais à esquerda do índice) e qualquer restrição de desigualdade nos restantes atributos, limitando assim o número de entradas que são necessárias percorrer dentro do índice.

CREATE INDEX name ON table (column1, column2);

Apesar de serem vantajosos, estes índices devem ser utilizados com cuidado, porque muitas das vezes, um índice sobre um único atributo é suficiente para poupar tempo de pesquisa e espaço na base de dados.

# Combinação de Índices

Os índices multi-atributos são a primeira escolha do optimizador quando a pesquisa a esse índice utilizada a condição AND nos atributos de pesquisa. Contudo, quando a condição utilizada é um OR, estes índices não podem (ou não convêm) ser utilizados, já que iríamos percorrer praticamente todo o índice para obter os resultados. O PostgreSQL possui a habilidade de combinar múltiplos índices (incluindo várias iterações sobre o mesmo índice) de modo a lidar com os casos em que os índices multi-atributo (ou a pesquisa em um só índice) não são úteis para a pesquisa requerida. Dando como exemplo uma *query* do tipo WHERE x = 20 AND y = 20, em que existem dois índices, um sobre cada um dos atributos, requer que ambos os índices sejam percorridos para obter os resultados correspondentes à restrição de igualdade de cada um e, posteriormente, combinados para construir o resultado final.

Para combinar múltiplos índices, o sistema itera sobre cada índice e constrói um bitmap, em memória, para cada um dos índices, com as localizações dos tuplos que satisfazem a condição de pesquisa para cada atributo. Estes bitmaps são depois

combinados através das cláusulas de pesquisa (AND ou OR) presentes na condição WHERE.

Raramente são utilizadas estas combinações de índices, o optimizador opta por escolher a iteração sobre um único índice (tanto de apenas um atributo como multi-atributo), já que a combinação dos índices por vezes resulta numa ordem diferente da qual se dispunha no próprio índice, devido à utilização dos bitmaps.

# Estruturas Temporariamente Inconsistentes

Devido à existência de concorrência no PostgreSQL, podem acontecer momentos de inconsistência entre as tabelas e as respetivas estruturas de índice, pois o PostgreSQL permite definir a altura em que é efetuada a verificação das restrições, através das cláusulas DEFERRED ou IMMEDIATE no comando SET CONSTRAINTS. Assim, com a cláusula DEFERRED, as restrições só são verificadas no final da transação, antes da realização do *commit*, e assim permitindo a inconsistência entre tabela e índice. No caso da cláusula IMMEDIATE, esta indica que as restrições são imediatamente verificadas após cada instrução.

# Índices Parciais e Índices Únicos

No PostgreSQL é permitida a definição de índices parciais e índices únicos. Os índices parciais são definidos sobre um subconjunto dos atributos de uma tabela, através de expressões condicionais, e apenas irão conter os valores que satisfaçam essa expressão. Os índices parciais tornam-se úteis devido ao facto de evitarem a indexação de valores comuns, pois as *queries* que os procurem não vão utilizar um índice de qualquer forma, reduzindo o tamanho do índice e acelerando o processamento das *queries* que de facto o utilizam.

Um índice único é utilizado para forçar a unicidade de um valor de uma ou várias colunas, funcionando exatamente como a restrição UNIQUE aplicada às tabelas (os índices únicos são inclusivamente criados automaticamente para cada restrição UNIQUE ou para cada chave primária de uma tabela). Os índices únicos são construídos com a estrutura B-Tree.

```
CREATE UNIQUE INDEX name ON table ( column [, ...]);
```

# Indexação para Organização de Ficheiros

O PostgreSQL organiza os ficheiros sobre uma *heap*, sem qualquer processo de *clustering* envolvido. Apesar disso, o sistema permite que seja possível manipular esta

organização, para que esta seja baseada numa estrutura de indexação, através do comando CLUSTER.

```
CLUSTER [VERBOSE] table_name [ USING index_name ]
```

Ao ser invocado, este comando reordena fisicamente uma tabela de acordo com a informação indexada sobre a própria tabela (ou seja, tem de existir um índice sobre essa tabela). Esta operação é feita apenas uma vez, portanto, consequentes atualizações à tabela não serão clustered, tendo de ser executado novamente o comando para as atualizações serem devidamente ordenadas.

# Oracle 11g vs. PostgreSQL

Sendo dois dos sistemas de base de dados mais utilizados, é normal que surjam comparações entre as funcionalidades e capacidades de ambos. No que corresponde à indexação, estes são bastante parecidos, especialmente nos índices mais vulgarmente utilizados, como o índice Hash ou o índice B+ (B-Tree).

- Apesar do PostgreSQL utilizar bitmaps para a combinação de índices, este não implementa estruturas desse tipo como índices, enquanto o Oracle as implementa.
- O Oracle não suporta árvores de procura genérica que permitem a criação de novos índices para tipos de dados definidos pelo utilizador (GiST).
- O Oracle não suporta a combinação de índices nas pesquisas, apesar de implementar índices bitmap (que o PostgreSQL utiliza para as combinações.

A tabela seguinte tenta ilustrar a comparação entre ambos, apontando os vários aspetos de indexação que cada uma possui ou não possui.

| Tipo de funcionalidade | Oracle | PostgreSQL |
|---|---|---|
| Índice B+ (B-Tree) | Sim | Sim |
| Índice Hash | Sim | Sim |
| Índice GiST | Não | Sim |
| Índice GIN | Sim | Sim |
| Índice Bitmap | Sim | Não |
| Índice Parcial | Sim | Sim |
| Índice Único | Sim | Sim |
| Organização de Ficheiros | Sim | Sim |
| Inconsistência Temporária | Sim | Sim |

**Tabela 3: Comparação entre Oracle e PostgreSQL**

# Processamento e Otimização de Perguntas

## Processamento de Consultas

Os passos que o PostgreSQL utiliza para o processo de uma consulta são:

- Recebe a consulta
- Parser
- Constrói um plano de consulta
- Executa esse plano

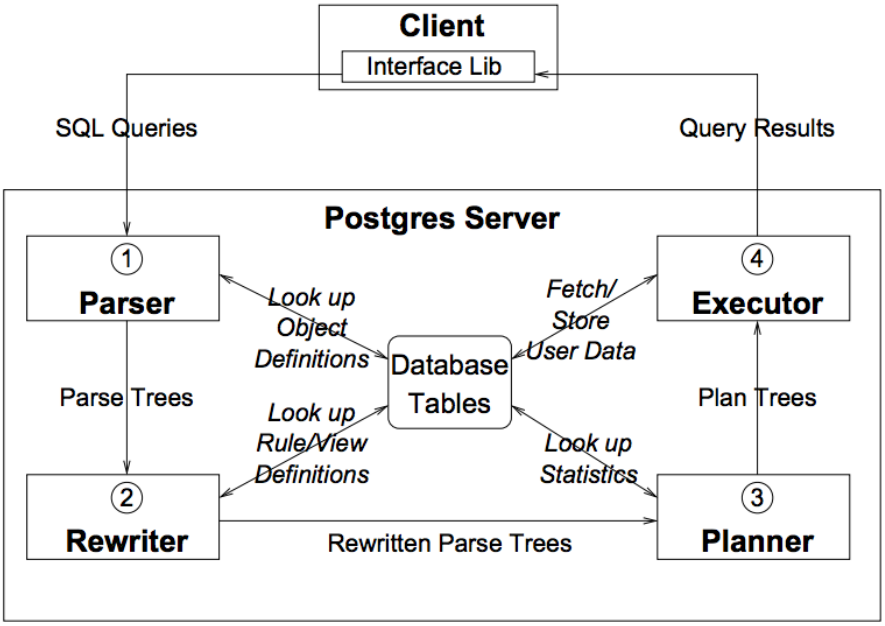


**Figura 1:** **Esquema do Processamento de Consultas**

# Parser

O *parser* tem de verificar a string de consulta, para que esta tenha a sintaxe válida. Se a sintaxe estiver correta a *parse tree* é construída e devolvida mas, caso contrário, é devolvido um erro.

Desta forma, o *parser* consiste num conjunto de regras gramaticais e nas ações que são efetuadas quando uma dada regra é colocada a funcionar. O código destas ações é utilizado para construir a *parse tree*.

A fase de *parser* cria uma *parse tree* utilizando regras fixas sobre a estrutura sintática do SQL. Assim que este *parser* se encontra concluído, o processo de transformação conduz a árvore que foi devolvida por este como entrada, e a interpretação semântica que é necessária para apreender quais são as tabelas, as funções e os operadores a serem referenciados pela consulta. Desta forma, a estrutura de dados que é delineada para representar esta informação é designada por *query tree.*

# Reescrita de Consultas

A reescrita de consultas é a responsável pelo sistema de regras do PostgreSQL, sendo que as regras só são definidas pelos utilizadores e pela definição de *views*. Assim, uma regra é registada no sistema utilizando o comando *create rule* e a informação sobre esta regra é guardada num catálogo. Este catálogo é utilizado durante a reescrita de consultas para descobrir todas as regras para uma determinada consulta.

Desta forma, a fase de reescrita de uma dada consulta é efetuada da seguinte maneira:

- Primeiramente, é efetuado o tratamento de todos os *updates*, *deletes* e *inserts* das declarações utilizando todas as regras apropriadas.
- De seguida, trata as restantes regras envolvendo apenas instruções de *select* que foram acionadas.
- Por fim, as regras são todas verificadas até que não existam regras a precisarem de ser descartadas.

### Planeamento e Otimização de Consultas

A tarefa do planeamento e otimização de consultas é criar um plano de execução ideal. Numa dada consulta, esta pode ser executada de várias maneiras, sendo que cada uma irá produzir um dado conjunto de resultados.

Em determinadas situações tem de se avaliar a quantidade em excesso de tempo e o espaço ocupado em memória que a consulta leva a ser executada. Estas situações ocorrem quando a execução de consultas devolvem uma grande quantidade de operações de associações.

Desta forma, para se determinar um plano para estas consultas num período de tempo razoável, utiliza-se um *Genetic Query Optimizer*, quando o número de junções excede um dado limite.

Na figura abaixo apresentamos um exemplo de um dado plano de consultas:

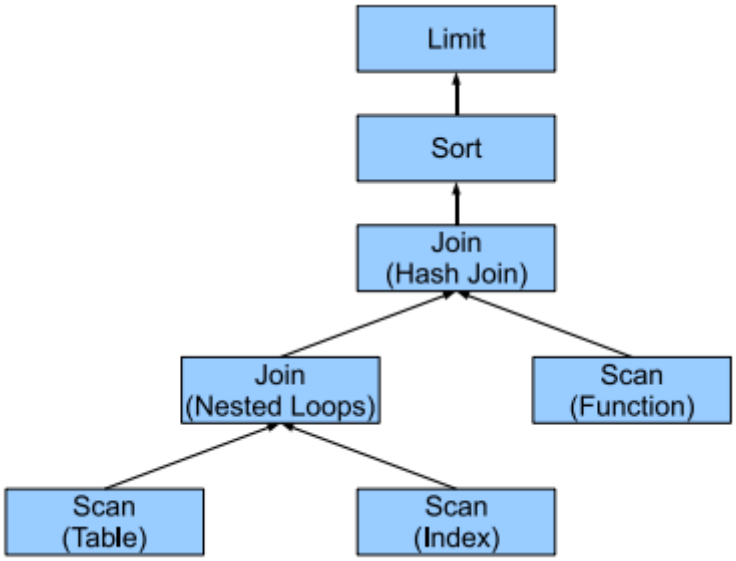


**Figura 2: Plano de Consultas**

# Execução de Consultas

O executor é o responsável por realizar as perguntas SQL com o plano criado pelo optimizador.

Desta forma, o executor de consultas é caracterizado pelos seus métodos de acesso, métodos de junção, ordenação e agregação que apresentamos de seguida.

## Métodos de Acesso:

- *Sequential Scans* - Efetua a digitalização sequencial do primeiro até ao último bloco. Desta forma, cada tuplo é devolvido para o interlocutor se este se encontrar visível. Neste tipo de acesso é possível avaliar os predicados que se referem a esta *table.* Assim, existe um problema neste acesso, pois apesar de precisar apenas de algumas linhas para satisfazer a consulta, este examina sempre a *table* por completo.

- *Index Scans* - Este tipo de acesso utiliza uma estrutura de dados secundária para encontrar rapidamente os tuplos que satisfazem um determinado predicado. Os tipos de índice mais populares incluem as árvores, as tabelas *hash* e bitmaps. Caso existam muitas linhas a corresponder a um dado predicado, este tipo de acesso torna-se ineficiente. Neste tipo de casos, existe um problema, pois o índice ao ser atualizado em cada inserção, este ocupa demasiado espaço no *buffer*.

- *Bitmap Index Scans* - Este tipo de acesso serve para separar os índices de scan do scan do *heap.* Para cada índice na tabela de destino verifica-se o índice para encontrar os tulos de qualificação e faz-se o registo de qualificação dos tuplos definindo bits num *bitmap* em memória. Desta forma, é utilizado um mapa de bits para digitalizar o *heap* ordenado. Este acesso apresenta alguns benefícios no que toca à leitura sequencial do *heap,* em vez de ler na ordem do índice. Permite a combinação de vários índices numa única tabela, sendo mais flexível do que os índices de várias colunas.

## Métodos de Join:

O PostgreSQL neste tipo de métodos suporta os tipos *como merge join, nested-loops* e *hybrid hash join.*

## Ordenação:

Em relação à ordenação, dada uma relação, caso esta não se enquadre na memória é utilizado o *sort-merge* que combina duas listas ordenadas como um fecho. Sendo que ambos os lados da junção devem ser resolvidos pelos predicados participantes. Mas, caso a relação caiba na memória é usado o *quicksort.*

Os operadores de ordenação são usados para efetuar a remoção de duplicados e também para fazer a remoção dos duplicados no operador *join*.

## Agregação:

O PostgreSQL suporta este tipo de funções de agregação, sendo que uma de agregação avalia o seu resultado para várias linhas de entrada. Exemplo de funções de agregação são o *count, sum, avg, max* e *min*. Este tipo execução é bastante útil para as combinações da cláusula *GROUP BY*.

Existe interação entre as agregações e as cláusulas WHERE e HAVING. A diferença entre estas cláusulas é que a WHERE seleciona as linhas de entrada antes dos grupos e das agregações serem avaliados, enquanto que a HAVING seleciona as linhas de um grupo após os grupos e as agregações serem avaliados. Ou seja, a cláusula WHERE não pode conter funções de agregação pois não faz sentido tentar utilizar uma agregação para determinar quais as linhas que serão a entrada da agregação. Já a cláusula HAVING contém funções de agregação.

## Triggers e Restrições

A implementação de *triggers* e restrições não é efetuada na fase da reescrita, mas sim como parte do executor da consulta.

Quando o registo é efetuado pelo utilizador, os dados são associados com as informações do processo, sendo que as verificações do executor para os *triggers* candidatos e as restrições de tuplos são alterados para efetuar a atualização, inserção ou remoção.

# Gestão de Transações e Controlo de Concorrência

Neste capítulo é abordado como são geridas as transações e como o PostgreSQL lida com acessos simultâneos.

## Controlo de Concorrência

O controlo de concorrência de um SGBD permite mais do que uma sessão a aceder aos mesmos dados ao mesmo tempo. Os subcapítulos abaixo apresentados apresentam o modo como o PostgreSQL maximiza a eficiência de todas as sessões enquanto mantém a integridade dos dados.

## Explicit Locking

Em PostgreSQL podemos declarar vários modos de *locks* para aceder a uma tabela de modo a controlar o acesso concorrente. Estes modos de *locks* podem ser aplicados quando o MVCC não providencia o comportamento desejado.

### Locks para Tabelas

Na lista abaixo apresentam-se os vários modos de *locks* e o contexto em que são usados automaticamente pelo PostgreSQL, caso se deseje fazer uso destes locks explicitamente através do comando LOCK.

#### Access Share

Por norma, qualquer *query* que seja apenas de leitura a uma tabela adquire este modo de *lock.*

Conflitos: *Access Exclusive*.

#### Row Thare

Os comandos SELECT FOR UPDATE e SELECT FOR SHARE adquirem este modo de *lock* sobre a tabela a realizar a operação.

Conflitos: *Exclusive* e *Access Exclusive*.

### Row Exclusive

Por norma, este modo de *lock* é adquirido por qualquer comando que modifique dados na tabela, ou seja os comandos UPDATE/DELETE e INSERT. Além deste modo são também obtidos *Access Share locks* em qualquer tabela referenciada.

### Share Update Exclusive

Este modo de *lock* protege a tabela de mudanças ao *schema* concorrentes e execuções VACUUM (garbage collection de tuplos que estão mortos ou obsoletos mas ainda não foram fisicamente apagados da tabela).

É adquirido pelas operações: VACUUM (sem o parâmetro FULL), ANALYZE, e CREATE INDEX CONCURRENTLY.

Conflitos: SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, e ACCESS EXCLUSIVE.

### Share

Este modo protege a tabela contra alterações na tabela concorrentes e é adquirido pela operação CREATE INDEX (sem o parâmetro concorrência).

Conflitos: ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, e ACCESS EXCLUSIVE.

### Share Row Exclusive

Este modo protege a tabela contra alterações de dados concorrentes, e apenas pode ser usado por uma sessão de cada vez.

Conflitos: ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE.

### Exclusive

Este modo apenas permite *locks* Access Share concorrentes, ou seja caso uma transação tenha obtido este *lock* numa tabela, apenas são permitidas leituras a essa tabela em paralelo. Este modo não é obtido automaticamente por tabelas criadas pelo utilizador, no entanto é adquirido por algumas operações internas executadas pelo sistema.

Conflitos: ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE e ACCESS EXCLUSIVE.

### Access Exclusive

Este modo garante que a transação que adquiriu este *lock* é a única a aceder à tabela, e é adquirido pelas operações ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, e VACUUM FULL.

Conflitos: Todos os modos acima.

## Locks para Tuplos

Além dos *locks* para tabelas que se analisou no ponto prévio, existem também *locks* para tuplos que podem ser exclusivos ou partilhados. Um *lock* exclusivo para um tuplo é automaticamente obtido quando se atualiza ou apaga esse tuplo. Este *lock* é mantido até a transação realizar *commit* ou *rollback*.

Nota: Do mesmo modo que os *locks* para tabelas funcionam, os *locks* para tuplos não afetam *queries* aos dados, afetam exclusivamente processos de escrita no mesmo tuplo.

### Exemplos

#### Row Lock Exclusive

Para se obter um *lock* exclusivo de um tuplo sem modificarmos esse tuplo, realiza-se a operação SELECT com SELECT FOR UPDATE. A partir do momento em que o *lock* foi adquirido, a transação pode modificar várias vezes esse tuplo sem haver a possibilidade de conflitos.

#### Shared Row Lock

De modo a obter-se um *lock* partilhado num tuplo, faz-se uso da operação SELECT with SELECT FOR SHARE. Este *lock* partilhado não impede outras transações de obterem esse *lock* partilhado (e fazerem leituras neste tuplo), mas se tentarem atualizar, apagar, ou obter um *lock* exclusivo, ficam bloqueadas até aos *locks* partilhados serem libertados.

## Deadlocks

O uso de *locks* explicitamente aumenta a probabilidade de se criar um *deadlock*, um *deadlock* é quando cada uma de 2 ou mais transações obtiveram *locks* que a outra precisa. Por exemplo a transação A obteve um *lock* na tabela X e depois tenta obter um *lock* exclusivo na tabela Y, mas simultaneamente está a correr uma transação B que já tem o *lock* exclusivo de Y e agora quer o *lock* exclusivo de X. O resultado desta situação é um *deadlock*, onde nenhuma das transações consegue avançar.

O PostgreSQL deteta automaticamente *deadlocks*, e resolve-os abortando uma das transações envolvidas.

O melhor modo de evitar *deadlocks* é evitá-los ao se assegurar que certas aplicações a usar a base de dados obtém os *locks* em vários objetos de uma ordem consistente.

### Advisory Locks

Os Advisory *locks* são *locks* que estão definidos num ambiente aplicacional específico. O nome "advisory" origina do facto de serem aconselhados e não de uso obrigatório pelo sistema, a aplicação é que fica encarregue de fazer uso destes *locks*.

É possível obter estes *locks* através de 2 modos, a nível de sessão e a nível transacional. A nível transacional, uma vez obtido o *lock* este só é libertado no fim da sessão ou quando é explicitamente libertado. Estes *locks* apresentam algumas diferenças em relação aos *locks* convencionais, entre as quais:

- Podem ser adquiridos várias vezes (e naturalmente têm de ser libertados o mesmo número de vezes);
- Os *locks* de sessão não seguem a semântica transacional, ou seja se uma transacção adquirir um *lock* e depois for *rolled back*, o *lock* não é libertado automaticamente;
- Uma *query* que faz uso do comando LIMIT, não há garantias que o comando LIMIT é aplicado antes de se obter o *lock*, o que leva a que sejam obtidos *locks* que a aplicação não previu, como tal estes *locks* precisam de ser libertados explicitamente.

## Transacções

### Write-Ahead Logging (WAL)

O *WAL* é um mecanismo para assegurar a integridade dos dados. Essencialmente, as mudanças a ficheiros de dados apenas são aplicadas quando essas mudanças ficaram registadas num *log*. Deste modo mesmo numa situação de falha catastrófica é possível recuperar os dados ao se executar esse log, ou noutro caso propagar esse log para outros servidores (como se irá verificar na secção "Suporte para Base de dados Distribuídas".

### Funcionamento

Uma transação é uma agregação de várias operações que ou corre totalmente com sucesso e as alterações são aplicadas na base de dados, ou então nenhuma das alterações

é efetuada. O estado intermédio de uma transação não é visível pelas outras transações concorrentes.

Para identificar uma transação faz-se uso do comando *BEGIN* antecedendo as operações desejadas na transação. No fim da transação utiliza-se o comando *COMMIT* para denotar que se deseja aplicar as alterações na base de dados, ou caso a transação não tenha decorrido como desejado utiliza-se o comando *ROLLBACK* para anular a transação.

As transações em PostgreSQL podem ser controladas com uma maior granularidade através do uso de *SavePoints*, os *SavePoints* permitem descartar seletivamente partes da transação e realizar *commit* a outras partes.

Em PostgreSQL cada operação SQL é tratada como sendo uma transação, estando encapsulada por um *BEGIN* no início e um *COMMIT* no fim.

## Isolamento de Transações

Para se compreender os níveis de isolamento de um sistema de Base de Dados é necessário entender o porquê de existirem. O motivo dos níveis de isolamento existirem deve-se a 3 ocorrências com o nome de:

- *Dirty read*:
    - Uma transação lê dados escritos por uma transação que ainda não fez *commit*.
- *Nonrepeatable read*:
    - Uma transação lê dados que já tinha previamente lido e verifica que os dados mudaram (pois outra transação fez *commit*).
- Phantom read:
    - Uma transação executa novamente uma *query* que retorna um conjunto de tuplos que satisfazem uma condição, e verifica que o conjunto de tuplos que satisfazem essa condição mudou devido a outra transação que fez *commit*.
    -

Em PostgreSQL apenas há 3 níveis distintos de isolamento, estes são (ordenados):

- *Read Commited*;
- *Repeatable Read*;
- *Serializable*.

Os níveis de isolamento vêm aplicar restrições na execução concorrente das transações de modo a que as ocorrências não acontecem, ver fig.3.

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

**Figura 3: Tabela com os Níveis de Isolamento e Ocorrências**

Nota: Apesar de na tabela estarem dispostos 4 níveis, o PostgreSQL apenas oferece os 3 acima indicados. Se um utilizador requisitar o nível *Read Uncommited*, na realidade o nível aplicado é o *Read Commited*.

## Isolamento por Snapshots

Nos níveis de isolamento tem de ser possível fazer a gestão de versões de modo a cada transação ter acesso a dados consistentes (dependendo do nível escolhido). Em PostgreSQL esta gestão é realizada pelo Multiversion Concurrency Control (MVCC).

A diferença principal entre o MVCC e os modelos de *lock* tradicionais é que os *locks* obtidos pelo MVCC para ler dados não entram em conflito com *locks* obtidos para realizar a escrita de dados, e como tal ler nunca impede a escrita de blocos, e a escrita de blocos nunca impede a leitura.

## Análise aos Níveis de Isolamento

### Read Commited

Este nível é o utilizado por default. Quando uma transação usa este nível uma *query* de SELECT apenas consegue ver dados que foram commited antes da *query* iniciar.

Este nível funciona através do uso de snapshots, antes de cada operação de uma transação o Sistema tira uma snapshot da base de dados e trabalha com base nessa snapshot. No entanto apesar deste uso das snapshots uma transação pode ver 2 dados diferentes ao realizar 2 operações SELECT seguidas. Esta situação pode acontecer caso outras transações realizem COMMIT durante a execução do 1º SELECT.

As operações de UPDATE, DELETE, SELECT FOR UPDATE e SELECT FOR SHARE apresentam o mesmo comportamento que a operação SELECT em termos da pesquisa pelos tuplos alvo, ou seja apenas encontram tuplos alvo que foram commited na altura da execução da operação.

Este nível de isolamento não é adequado para operações de pesquisa complexa, devido à possibilidade de uma operação UPDATE operar sobre uma snapshot inconsistente.

### Repeatable Read

Este nível oferece um nível de isolamento superior ao *Read Commited*, pois além de apenas permitir à transação ver dados que foram *commited* antes da *query* iniciar, também garante que quaisquer dados lidos pela transação não podem mudar durante a transação.

Este nível também faz uso de *snapshots* (tal como o modo *Read Commited*), no entanto ao invés de tirar uma *snapshot* antes de cada operação da transação, a *snapshot* é tirada no início da transação. Consequentemente sucessivas operações de SELECT dentro de uma transação resultam sempre nos mesmos dados.

Naturalmente com base neste mecanismo a possibilidade de repetir as transações é maior devido a falhas de serialização.

Devido a estas características este nível funciona melhor aliado ao uso de *locks* para evitar que surjam transações em conflito e a repetição seja necessária.

### Serializable

Este nível é o mais restrito de isolamento oferecido pelo PostgreSQL. Que simula execução das transações em série, ou seja como se as transações tivessem sido executadas uma a seguir à outra, ao invés de concorrentemente.

Este nível funciona exatamente do mesmo modo que o *Repeatable Read*, com a diferença que monitoriza as condições que podem tornar a execução de várias transações serializáveis inconsistente comparando com a execução em série dessas mesmas transações.

Relativamente ao tempo de overhead deste nível comparando com o *Repeatable Read*, apenas é incrementado o overhead de monitorizar as condições, e caso surja uma condição que causa uma anomalia de serialização então lança-se um "serialization failure".

Relativamente à implementação deste nível, o PostgreSQL faz uso de *predicate locking*, ou seja mantém *locks* que permitem determinar se um *write* tem impacto no resultado de um *Read* prévio de um transação concorrente caso esta tivesse corrido primeiro.

Nota: Estes *locks* não bloqueiam as transações e como tal não podem levar a *deadlocks*. São usados apenas para identificar e anotar dependências nas transações serializáveis.

# Suporte para Bases de Dados Distribuídas

Existem diversas abordagens para escalar um Sistema de Base de Dados PostgreSQL, cada uma destas abordagens apresenta vantagens e desvantagens, pelo que é necessário compreender os requisitos da base de dados e ajustar uma ou mais abordagens que se enquadram. Nos seguintes pontos serão abordados as vantagens e desvantagens dos seguintes mecanismos:

- Servidores *Standby*;
- Replicação por *streaming*;
- Replicação em cascata;
- Replicação síncrona.

## Servidores Standby

O uso de 1 ou mais servidores *Standby* permite ter servidores prontos a tomar controlo em caso de falha no servidor primário.

Caso se tenha um ou mais servidores *standby*, o servidor primário vai funcionar em modo de arquivo continuo e os servidores *standby* em modo continuo de recuperação através da leitura dos *WAL (write ahead log)* files do servidor primário.

O uso dos servidores *standby* não exige que se realizem alterações às tabelas, pelo que comparando com as outras soluções de replicação, esta exija um *overhead* de administração menor. Outra vantagem de este mecanismo é o baixo impacto que o mesmo tem sobre o servidor primário.

Por outro lado, uma das desvantagens associada a este mecanismo advém da transferência dos registos *WAL*, denominada por *log shipping*. Esta técnica envolve transferir um segmento *WAL* de cada vez, e naturalmente depende da taxa de transações do servidor primário, bem como da velocidade da rede entre o servidor primário e os servidores *standby*.

Outra das vantagens é a existência de uma janela de tempo onde é possível perder informação. A técnica de log shipping é asíncrona, ou seja os registos *WAL* são enviados para o servidor standby após serem *commited*. Como tal numa situação de falha crítica, as transações que ainda não foram enviadas serão perdidas. Esta janela de tempo pode ser minimizada consoante a redução do tempo de timeout, no entanto esta redução implica um maior uso da rede para transferir mais frequentemente informação.

# Replicação por Streaming

A replicação por streaming permite a um servidor standby estar mais *up-to-date* com o servidor primário do que através do uso do *log shipping*. O funcionamento deste mecanismo ocorre do seguinte modo, o servidor *standby* liga-se ao servidor primário, que através de streams envia os registos WAL para o servidor standby, não sendo necessário esperar que o ficheiro WAL seja totalmente preenchido.

A replicação por *streaming* é assíncrona por default, o que indica que há um pequeno atraso entre fazer *commit* de uma transação no servidor primário e as alterações ficarem ativas no servidor standby. Este pequeno atraso é inferir a 1 segundo assumindo que o servidor de *standby* tem recursos suficientes para lidar com os dados enviados continuamente.

# Replicação em Cascata

Através da replicação em cascata podemos reduzir o número de ligações diretas entre o servidor primário e os servidores "slave". Esta técnica funciona ao permitir que um servidor *standby* aceite replicar conexões e faça *stream* dos registos WAL para outros servidores standby, passando a funcionar como um *relay*.

Um servidor *standby* que funciona tanto como um recetor e um emissor denomina-se servidor *cascading standby*. Denomina-se também os servidores *standby* mais próximos do servidor primário de servidores *upstream*, e por outro lado os servidores *standby* mais distantes de servidores *downstream*.

# Replicação Síncrona

Como referido nos pontos acima a replicação por streaming é assíncrona por default. Caso o servidor primário tenha uma falha catastrófica então as transações que foram commited mas que não foram replicadas para os servidores *standby* são perdidas.

A replicação síncrona proporciona a capacidade de confirmar que todas as alterações efetuadas foram comunicadas a um servidor *standby*.

Como tal aumenta o tempo para se realizar *commit* mas reduz-se a probabilidade de perda de dados.

# Consistência dos Dados

O PostgreSQL faz uso de um modelo multi-versão, quando os dados são modificados os *locks* exclusivos são mantidos até ao fim da transação. No caso do modelo multi-versão uma *copy* do tuplo original é mantido para os *readers*, antes de realizar modificações pelos *writers*. Os *readers* não precisam de esperar pelos *writers* como em outros Sistemas de Bases de Dados (e.g Informix), cada transação trabalha sobre uma cópia privada da base dados.

No nível de isolamento serializable todas as versões usadas por readers e writers deverão ser consistente com os dados correntes, não é necessário mais nenhum cuidado para garantir consistência.

# Outras Características

## Autenticação

Para garantir a confidencialidade os utilizadores deverão identificar-se junto da Base de Dados, o PostgreSQL suporta os seguintes modos de autenticação:

- TRUST;
- Password;
- Generic Security Services Application Program Interface(GSSAPI);
- Security Support Provider Interface (SSPI);
- Kerberus;
- Ident-based authentication;
- LDAP (Lightwight Directory Access Protocol);
- RADIUS;
- Certificate;
- Pluggable Authentication Modules (PAM)

Através da configuração destes modos de autenticação, é também possível definir *Access Control Lists* (*ACLs*).

Nota: Os utilizadores da base de dados e os utilizadores do servidor não são partilhados.

# Comparação com o Oracle 11g e o MySQL

O PostGreSQL é um servidor de base de dados unificados com um único *storage engine*, por outro lado o MySQL apresenta-se dividido em 2 *layers*, uma camada superior de SQL e um conjunto de *storage engines*. Como tal para comparar o PostGreSQL com o MySQL é necessário qual dos *storage engines* será usado no MySQL, pois os *engines* têm um impacto significativo na performance, *availability* e até nas *features* básicas que se oferece.

O storage engine mais comum no MySQL é o InnoDB, que oferece quase na totalidade ACID (Atomicidade, consistência, isolamento, e durabilidade), bem como boa performance quando o sistema está a executar grandes operações em concorrência.

## Performance

A performance de um SGBD pode ser optimizador de acordo com o ambiente em que está a correr, através de configurações, isto leva a que seja difícil apresentar uma comparação exata, sem ser analisado o ambiente do SGB e as respetivas configurações.

### Conceitos Base

O MySQL foi inicialmente planeado de modo a ser rápido, e por outro lado o PostgreSQL foi planeado de modo a ser robusto em *features* e em definir *standarts*. Consequentemente o MySQL era frequentemente o mais rápido dos 2, isto também porque grande parte dos *benchmarks* realizados por utilizadores comuns foram elaborados com as configurações *default* (que está afinado para sistemas pequenos).

Em termos de performance ambos os DBMS apresentam melhores resultados em *benchmarks* associados às respetivas qualidades, no MySQL a velocidade em operações simples, e no PostgreSQL mais fiável e rápido em operações complexas.

No PostGreSQL os dados de uma tabela e de um *index* são armazenados em estruturas de dados diferentes. Por outro lado no MySQL com o *engine InnoDB* faz-se uso da *index organized table*, que requer que todas tabelas tenham uma chave primária (ou um *row id*, caso a chave primária não exista). No caso do Oracle ambas as maneiras de estruturação são aceites.

## Features Relativas a Performance do PostgreSQL

O PostgreSQL apresenta diversas *features* que conferem a eficiência desejada, entre as quais:

- Optimizador avançado baseado em custo;
- Executor eficiente tanto para SQL estático como parametrizado;
- Indexação: parcial, funcional, combinação de vários índices, scans de apenas índice, e 5 tipos diferentes de índice;
- Compressão de dados *TOAST* (The Oversized-Attribute Storage Technique);
- Gestão de cache melhorada a partir das versões 8.1 e 8.2;
- Elevada capacidade de escabilidade em períodos de escrita intensiva (a partir das versões a partir da 8.1);
- Commit assíncrono;
- Replicação assíncrona (a partir da versão 9.0);
- Replicação síncrona (a partir da versão 9.1).

## Features Relativas a Performance do MySQL

- O *InnoDB* guarda a chave primária com os dados , por isso os look ups por chave primária são rápidos;
- Geração automática de entradas *hash* ao processar SELECTS;
- INSERT/UPDATE Buffer que faz *caching* de updates a entradas secundárias de índices;
- Compressão de tabelas em tempo real.

## ACID

O modelo de Atomicidade, consistência, isolamento e durabilidade (ACID) é usado para avaliar a integridade dos dados em DBMS.

A maioria dos DBMS garantem ACID ao fazerem uso de transações e aplicarem *triggers*/chaves estrangeiras.

O PostgreSQL garante na totalidade o modelo ACID.

O *InnoDB engine* garante na totalidade o modelo ACID, mas usar o *InnoDB* em MySQL causa a falha de *ACIDity*, isto porque o MySQL não propaga os *triggers* para chaves estrangeiras para este *engine*.

O PostgreSQL é reconhecido como o que apresenta uma abordagem mais robusta e rigorosa relativamente à integridade dos dados.

## Indexação PostGreSQL

A indexação em PostGresQL funciona de um modo semelhante ao Oracle, mas como se verificou acima apresenta algumas *features* que oferecem uma vantagem, estas são o *GiST* (Generalized Search Tree) e o *GIN* (Generalized Inverted Index) abordados em profundidade nos capítulos anteriores.

O PostgreSQL não permite forçar o índice a usar, escolhendo sempre o melhor plano de execução.

## Concorrência e Optimização

O Oracle 11g apenas oferece 2 tipos de isolamento, *Read Committed* e *Serializable*, mas oferece um tipo auxiliar de transação, o *Read Only* que é equivalente ao *Repeatable Read* ou a uma transação serializável que não realiza modificações.

O PostgreSQL oferece os 4 tipos mas na realidade são apenas 3, uma vez que ao se selecionar o nível *Read Uncommitted*, o nível que é usado é o *Read committed* (Ver capítulo Transações.

O PostgreSQL oferece também outra *feature* relevante na concorrência, que é o *advisory lock*.

Relativamente à otimização, o PostgreSQL faz uso de mais algoritmos do que o Oracle 11g para obter o melhor plano de uma *query*. O que pode relevar-se excessivo pois o tempo despendido a analisar os diversos planos pode ser superior ao tempo de executar um plano menos eficiente.

## Distribuição

O PostgreSQL e o Oracle 11g suportam replicação de dados numa arquitetura de servidores primários e réplicas.

# Features em Desenvolvimento

A versão 9.4 do PostgreSQL será disponibilizada a partir de Julho de 2014, as *features* incluídas nesta versão podem mudar ao longo do desenvolvimento, no entanto as mais importantes planeadas são as seguintes:

- *Logical streaming replication*;
- *Materialized views*;
- Índices *JSON*;
- Segurança a nível de tuplo.

# Referências

[1] http://leadit.us/hands-on-tech/PostgreSQL-vs-MySQL-Pros-Cons

[2] http://www-css.fnal.gov/dsg/external/freeware/pgsql-vs-mysql.html

[3] http://www.postgresonline.com/journal/index.php?/archives/130-Cross-Compare-of-PostgreSQL-8.4,-SQL-Server-2008,-MySQL-5.1.html

[4] http://www.devx.com/dbzone/Article/29480?trk=DXRSS_DB

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

[6] http://www.devx.com/dbzone/Article/20743

[7] http://www.postgresql.org/docs/9.3/static/index.html

[8] http://dev.mysql.com/doc/refman/5.6/en/index.html

[9] http://docs.oracle.com/cd/B28359_01/index.htm

[10] http://www.oracle.com/technetwork/issue-archive/2005/05-nov/o65asktom-082389.html