Relatório de Análise sobre PostgreSQL

Sistemas de Bases de Dados
Mestrado Integrado em Engenharia Informática FCT/UNL
Ano Lectivo 2013/14

Grupo 17
Celso Alexandre, 41853
Miguel Janicas, 29393
Pedro Rodrigues, 41889

Índice

1 – Introdução

Este relatório aparece no âmbito do projeto final da cadeira de Sistemas de Bases de Dados na Faculdade de Ciências e Tecnologias da Universidade Nova de Lisboa (FCT/UNL) no segundo semestre do ano lectivo de 2013/2014.

O trabalho consiste em analisar um SGBD escolhido pelo grupo e explorar os temas que foram lecionados nas aulas teóricas de modo a complementar as bases teóricas sobre os diferentes métodos que distintos sistemas aplicam na sua abordagem. O contexto teórico aplicado nas aulas foi o do Oracle que servirá como ponto de comparação com a escolha do nosso foco de estudo, o **PostgreSQL**.

O PostgreSQL foi um projeto iniciado em 1986 na universidade da Califórnia em Barkeley nos Estados Unidos da América. Desenvolvido maioritariamente por voluntários e com um propósito didático, este projeto revelou ser bastante inovador ao introduzir funcionalidades que resolviam problemas que existiam neste tipo de sistemas.

Até 1996 o PostgreSQL era apenas aplicado em projetos internos à universidade, mas a partir desse ano foi lançada a primeira versão externa. Foi-lhe atribuída uma licença de open-source o que acabou por chamar muitos mais investigadores e entusiastas da comunidade de software livre que contribuíram com a estabilização do sistema e com o desenvolvimento de novas funcionalidades, correções de erros e nova documentação.

Sendo neste momento um sistema robusto e avançado é utilizado em projetos de dimensões pequenas e médias onde os investimentos monetários não abundem, bem como em empresas maiores nas mais diversas áreas, por exemplo algumas instituições mais conhecidas são IMDB, Fujitsu, Skype ou Cisco.

A sobrevivência deste projeto passa por doações e patrocínios de várias empresas.

2 - Armazenamento e File Structure

O PostgreSQL recorre ao sistema de ficheiros do SO para guardar a sua base de dados, resultando numa hierarquia de diretorias cuja raiz normalmente é **PGDATA**, onde encontramos os ficheiros de controlo, postgresql.conf, pg_hba.conf, pg_ident.conf e mais diretorias. Seria interessante referenciar algumas, como pg_twophase ou pg_subtrans, que contêm o estado dos ficheiros que estão preparados para uma transação e links para as tabelas, respectivamente. Em relação à Base de Dados propriamente dita, deparamo-nos com a diretoria/base.

Existe um identificador *OID*, classificado como um identificador em PostgreSQL de objetos, que reconhece as tabelas e índices. Cada tabela tem um nome (filenode), que sendo um identificador tem um número correspondente a *OID*. Apesar de haver esta relação existe a possibilidade de, com algumas operações ou alterações de tabela, mudar o filenode e preservar o *OID*.

A linguagem contém um conjunto de ficheiros que estão guardados em Heap que referenciam cada tabela ou tuplo. Os ficheiros, estão organizados em Fork. Dentro das diretorias encontramos outros ficheiros, como é o caso de organização/ map filenode, estes com duas estruturas diferentes, representadas por sufixos: Temos o caso do *Free Space Map* "*_fsm", que é composto por uma árvore binária através da qual se obtêm facilmente os espaços vazios, localizados no fundo da mesma e que os níveis superiores agregam informação dos inferiores. O outro meio é denominado por *Visivlity Map* "*_vm", sendo que este contém bitmap das transações ativas.

No caso de um ficheiro ultrapassar 1 Gigabyte é necessário segmentá-lo em ficheiros mais pequenos. Estes ficheiros devem possuir o mesmo nome, apenas com o incremento do número de ficheiros gerados. e.g. filename.1, filename.2.

Na tabela seguinte encontram-se os limites que é suposto/esperado este sistema suportar:

Limite	Valor
Tamanho máximo da BD	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 dependente do tipo de colunas
Máximo de indexes por tabela	Ilimitado

2.1 - TOAST

A capacidade máxima de uma página, que é constituída por tabelas e índices, em PostgreSQL é de 8KB, visto isto não é possível armazenar diretamente valores muito grandes. Dada essa limitação estes são comprimidos e/ou partidos em diferentes tuplos físicos. A esta transformação, dá-se o nome TOAST – The Oversied Attribute Storage Technique.

Para suportar TOAST, a base de dados tem de ter um tamanho variável (varlena). A composição de cada valor guardado tem um cabeçalho de 32bits que contêm o tamanho do mesmo.

Os primeiros dois bits codificam o tamanho do campo que são usados para indicar as condições em que aqueles dados estão guardados. Se ambos os bits forem 0, então o tipo de dados é un-Toasted. Se o primeiro bit for 1, infere-se que os dados foram previamente comprimidos. Se o segundo bit for 1, então os dados que se seguem representam um apontador para outra zona da memória onde se deve encontrar o objecto pretendido. Por último, quando temos os dois bits com sinal positivo, os dados também serão comprimidos.

2.2 - Clustering

O PostgreSQL suporta single table clustering através do comando CLUSTER. Este apenas suporta clustering utilizando índices existentes (ordenando a tabela fisicamente de acordo com o índice). Quando a tabela é modificada, a ordenação física da tabela não se mantém (necessário executar outro comando de CLUSTER).

```
CLUSTER [nome da tabela [USING nome do índice]]
```

Se o nome do índice não estiver definido, tenta-se aplicar o comando à tabela utilizando o índice que foi usado anteriormente. Caso o comando não contenha argumentos, faz recluster a todas as tabelas anteriormente clustered usando os mesmos indexname.

3 - Índices

Existem 4 tipos de índices em PostgreSQL, dois que já lecionados e conhecidos por *B-Tree e Hash* e outros dois referentes à linguagem, *GiST* e o *GIN*.

A utilização de índices serve como um identificador e mecanismo de ordenação, isto é, tendo um número único e de pequena dimensão conseguimos uma pesquisa com um menor custo.

Criar:

```
CREATE INDEX nome do índice ON tabela (coluna) [USING tipo índice]
```

Remover:

```
DROP INDEX nome do índice
```

Em tabelas muito grandes a criação de um índice pode demorar muito tempo. Por defeito, em PostgreSQL é possível fazer um Select antes de acabar a criação do índice. No caso de uma inserção ou de um update, já é necessário esperar que o índice fique criado.

3.1 - Tipos de índices

B-Tree

Devido ao facto de cada tuplo ser identificado por uma chave, este índice dispõe de uma ordenação dos dados, tal como garante ser único. É também o índice que permite uma maior facilidade de inserção, remoção e leituras rápidas dos seus tuplos. Por norma é mais eficiente para operadores de comparação <; <=; =; =>; > . A sua eficácia também se verifica em operações mais complexas como o caso de BETWEEN, IN e LIKE. Através do índice *B-Tree* também é possível trabalhar a partir de restrições de preenchimento, como é o caso de IS (NOT) NULL.

```
CREATE INDEX nome do índice ON tabela (coluna) USING btree
```

Hash

Tem maior eficácia em comparações de igualdade (=) . No caso de ocorrer um "crash" na Base de Dados, é necessário criar novamente, fazer um "rebuild" completo.

```
CREATE INDEX nome do índice ON tabela (coluna) USING hash
```

Os PosgreSQL tem dois índices diferentes *GiST e GIN*, aos quais oferece diferentes estruturas de indexação. Além dessas estruturas, a par com o índice *B-Tree* têm capacidade de suportar múltiplos atributos.

GiST (Generalized Search Tree)

Com este tipo é possível implementar vários tipos de índices, *B-Tree* e *R-Tree*. Visto isto, é possível implementar diversas formas de estratégias de indexação.

```
CREATE INDEX nome do índice ON tabela (coluna) USING gist
```

Os operadores que o índice *GiST* pode usar possibilitam a utilização de "queries" indexadas: <<, &<, &>, >>, <<|, &<|, |&>, |>>, @>, <@, ~=, &&

Os índices *GiST* são também capazes de optimizar procuras do tipo "vizinho mais próximo", por exemplo:

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10
```

O *GiST* suporta tipos de índice tais como:

• Multi-coluna / Expressão / Parcial

Este tipo de índices pertence a uma funcionalidade disponível pelo PostgreSQL, denominada por Full-Text Search.

GIN (Generalized Inverted Index)

Funciona com uma *B-Tree* com valores compostos (número coluna, chave) podendo ter tipos diferentes, onde cada chave pode conter vários elementos.

O *GIN* é considerado um índice para fazer updates, tendo uma técnica chamada *Fast Update*. Esta maneira de fazer o update tem as suas desvantagens, isto porque cria uma estrutura temporária que funciona como um FIFO. Pode chegar a ter dimensões muito grandes o que dificulta uma pesquisa posterior, dado o facto de que para além de verificar na tabela ter de pesquisar na estrutura temporária as atualizações que estão em espera.

Como mencionado anteriormente o *GIN* tem a técnica *Fast Update* que mostramos um exemplo da sua sintaxe:

```
CREATE INDEX nome do índice ON tabela (coluna)
USING  gin [WITH (fastupdate = on/off)]
```

Além das classes de operadores apresentados em GiST, o GIN tem a possibilidade de operar sobre vetores unidimensionais, dos quais suporta: <@, @>, =, &&.

Contudo também permite Multi-coluna, Expressão e Parcial.

Este tipo de índice tal como GiST dispõe funcionalidade de Full-Text Search.

3.2 - Fill Factor

O PostgreSQL tem uma ferramenta através da qual é possível definir uma percentagem de uma página que pode ser preenchida, deixando o resto para futuras alterações, como UPDATE ou INSERT. Desta forma melhora a performance uma vez que já tem espaço reservado para novos dados.

O fill factor pode conter valores dos 10 aos 100%.

```
CREATE INDEX nome do índice ON tabela (coluna) WITH (fillfactor = 70)
```

3.3 - Índices Multi-coluna

Como referido anteriormente os índices multi-coluna apenas são suportados pelos tipos *B-Tree, GiST* e *GIN* e tem um limite de 32 colunas. Evidentemente que a performance da Base de Dados piora quanto maior for o número de índices.

```
CREATE INDEX nome do índice ON tabela (coluna1, colunan)
```

3.4 - Ordenação

Por defeito o índice *B-Tree* ordena os tuplos por ordem crescente e com os valores *NULL's* no final da tabela.

order by x <-> order by x ASC NULLS LAST

Se pretendermos uma ordenação decrescente, por defeito os valores *NULL's* aparecem no inicio da tabela, acontecendo exatamente o contrário na ordenação crescente.

É possível ajustar a ordenação com a inclusão das opções já conhecidas (ASC, DESC, NULLS FIRST/LAST).

```
CREATE INDEX  nome do índice ON tabela (ID DESC NULLS LAST)
```

3.5 - Expressões em Índices

Um índice não precisa de ser um atributo simples, mas pode ser uma expressão escalar aplicada a uma ou mais colunas de uma tabela. Esta funcionalidade é útil para aceder rapidamente e obter os resultados de dados compostos.

```
CREATE INDEX nome do índice ON tabela (lower(col1))
```

Índices sobre expressões são relativamente caros de manter, pois as expressões têm de ser calculadas para cada tuplo sempre que quisermos inserir ou modificar algum tuplo. Contudo o sistema não está constantemente a calcular o valor do índice uma vez que este foi calculado para a sua inserção.

3.6 - Índices Parciais

A utilização destes índices é exclusivamente para aumentar a performance, reduzindo o número de tuplos dependendo do predicado do índice associado.

Existem alguns tipos mais comuns, como é o caso de valores frequentes são valores que ocorrem na maior parte dos tuplos em questão, assim, eliminando estes valores reduzimos o número de índices e aumentamos a performance.

```
CREATE INDEX nome do índice ON tabela (coluna) WHERE condition
```

4 – Processamento e optimização de perguntas

Esta secção tem como objetivo explicar a sequência de passos que o SGBD aplica às queries efetuadas, desde do processo de inicial de *parsing* até à seleção dos melhores algoritmos e cálculo de custos, com a posterior apresentação dos resultados finais.

4.1 - Verificação lexical (Parser)

Esta fase pode ser dividida em dois pontos:
• Parser (avaliação sintática).
• Transformation Process (avaliação semântica).

No primeiro ponto, é lida uma sequência de caracteres e através das regras gramaticais é feita a tentativa de validação da sintaxe da *query*. No caso da sintaxe estar correta é criada uma **AST** (*Abstract Syntax Tree*) com as operações presentes na *query*, se estiver errada o processamento é interrompido e lança uma mensagem de erro.

O processo de transformação é o mecanismo que faz a avaliação semântica da árvore resultante do Parser, ou seja, em cada nó da AST encontra quais são as tabelas, operações e/ou funções que a *query* refere. À estrutura resultante dá-se o nome de *query tree*, e embora pareça estruturalmente semelhante à AST que deriva do Parser, esta tem mais informação contendo tipos de dados e resultados adjacentes aos nós.

4.2 - Planeamento e optimização (Rewriter)

Nesta fase, tendo como base a *query tree* proveniente da fase anterior, serão gerados exaustivamente **planos de execução** diferentes entre si, sendo o resultado final sempre o mesmo. No melhor dos casos seriam gerados todos os planos possíveis e de acordo com a estimativa de tempo de execução de cada um, seria escolhido aquele que demora-se menos tempo.

No entanto esta situação não é a mais comum pois em *query trees* grandes ou que contêm um número elevado de operações *join* a criação de todos os planos acaba por ser computacionalmente inviável (tanto em tempo como em espaço), e é nestes casos que é usado o **Genetic Query Optimizer** (explicado mais adiante).

Independentemente do número de planos criados tem de existir uma medida de comparação entre elas, sendo usado o método **baseado no custo mínimo**. Esta medida é efetuada através tanto de custos do CPU em processar tuplos como do

peso das operações de I/O no disco, onde as diferenças de tempo entre Reads e Writes de modo sequencial ou aleatório são também tidas em conta.

Outra propriedade importante nesta fase é no caso da *query* ser um comando utilitário (CREATE, UPDATE, DROP, etc.), bastar produzir o seu efeito na base de dados, não sendo necessário a criação de um plano de execução.

Para gerar planos, ao Oracle usa o mesmo tipo de princípios, mas em vez de usar algoritmos genéticos utiliza um método que passa por calcular um misto entre o custo de cada plano e heurísticas para obtenção das primeiras linhas do resultado final.

4.3 - Scan e seleção nas relações (Planner)

Primeiramente, o planeamento opta por gerar planos para percorrer as tabelas referidas na *query*. A forma mais simples e mais comum de percorrer uma tabela é através de **Sequencial Scan**, sendo sempre criado um plano com esse modo de seleção.

Mas em certos casos as *queries* sobre as tabelas não necessitam de as percorrer na totalidade (operações de seleção de igualdade ou maior/igual) e assim a existência de índices traria melhorias significativas usando **Index Scan**.

Enquanto que o Index Scan procura as entradas de índice e de seguida vai buscar os dados à respetiva tabela de dados, existe a possibilidade de utilizar o **Index Only Scan**, caso os dados e índice respetivo sejam construídos para tal efeito, que ao chegar à entrada de índice tem logo acesso aos tuplos, não necessitando de aceder à tabela de dados. Existe ainda a possibilidade de efetuar **Bitmap Index Scan**, que reduz a possibilidade de acessos aleatórios a tuplos encontrados num Index Scan.

Para que sejam gerados os melhores planos, por vezes o optimizador pode decidir por construir um índice sobre a tabela, se este considerar a sua criação uma mais valia para a performance dos resultados.

Como suporte à escolha dos melhores planos podem ser utilizas **estatísticas**. Com esta componente pode ser estimado o número de linhas a devolver pela *query*, obtido através de elementos que estão guardados, tais como, o número de tuplos de cada tabela e respetivo índice ou o número de blocos ocupados por estes.

Em relação à Oracle, esta tem um mais variado e refinado leque de opções sobre o Index Scan, fazendo distinção entre índice único, intervalos de índices ou scan total de índices por uma certa ordem.

4.4 - Operação de join e ordenações (Planner)

Depois de ser decidido como percorrer as tabelas, é obtida a forma como serão percorridas as operações de *join*, sendo que estão disponíveis três estratégias diferentes:

• **Nested Loop Join**: para cada linha da relação da esquerda é feito um varrimento na tabela da direita. Esta estratégia é fácil de implementar mas pode demorar muito tempo até estar concluída, por isso se possível pode utilizar índices (**Index Nested Loop Join**).

- Usado quando ambas as relações são pequenas ou quando existe índice no atributo *join* da tabela da esquerda.

• **Merge Join**: Cada relação deve estar previamente sorteada, através de **External Merge Sort**, ou de **Quicksort** ou ainda através de um índice sobre o atributo de *join* (se existir). De seguida cada relação é iterada em paralelo e se dois registos em cada relação são conciliáveis pelos atributo de *join* então é combinado o resultado e adicionado aos tuplos da resposta final.

- Bom funcionamento quando ambas as relações cabem em memória ou se já estiverem ordenadas ou com índices.

• **Hybrid Hash Join**: Através dos atributos de *join* é criada uma função de *hashing*. Essa função é aplicada aos tuplos da relação da direita e são distribuídos pelos buckets respetivos. O mesmo sucede com a relação da esquerda. Com os buckets de cada relação preenchidos basta agarrar em cada *bucket* com a mesma função de *hashing* e percorrer os tuplos até encontrar os tuplos que coincidem e adicioná-los ao resultado.

- Pode ser escolhido quando as comparações a serem feitas são de equidade.

Neste campo a Oracle implementa os mesmos três tipos principais de estrutura, com algumas pequenas variantes (e.g. Block Nested Loop Join).

No caso particular de existirem demasiados relações com *joins* (limite estipulado por um *threshold*), o optimizador utiliza o **Genetic Query Optimizer** para determinar que sequência de *join* deve considerar.

O conceito de Algoritmo Genético é aplicado nos mais variados métodos de pesquisa em muitas áreas da computação e consiste em algoritmos probabilísticos que tentam encontrar um exemplar num determinado espaço de hipóteses de modo a maximizar a função de fitness (qualidade do exemplar), não garantindo a devolução do máximo absoluto pois pode ficar retido num máximo local.

O **GEQO** codifica cada plano numa *string* de inteiros onde cada inteiro representa um dos planos e a ordem pela qual são efetuados os *joins*. Com isto, o

algoritmo tenta minimizar o fitness (por ser um custo basta inverte-lo) percorrendo o espaço de hipóteses que é o conjunto de todas as avaliações possíveis de *joins*.

4.5 - Modo de execução (Executor)

Depois da parte de planeamento e optimização, o executor opera sobre a *query tree* resultante e usa-a para extrair os tuplos recursivamente através de **pipeline Demand-Pull**. Este mecanismo funciona de forma similar a um iterador, onde cada nó tem um estado a referenciar o próximo tuplo a devolver, de modo a que o nó pai peça tuplos aos nós filhos (operador Next do iterador). A produção de tuplos a devolver pelos nós filhos acontece apenas quando estes são requisitados pelo nó pai.

Sendo o *pipelining* o mais usado, existe também a possibilidade de efetuar materialização. Este mecanismo aparece em casos especiais onde seja necessário voltar a percorrer parcialmente ou todos os tuplos de um certo nó (e.g. Nested Loop) ou ser necessário ter o resultado totalmente concluído (e.g. External Sort).

Nesta fase de a Oracle tem mais opções pois acrescenta *pipelining* producer-push (nós filhos geram resultados para um *buffer* onde o nó pai acede) e adiciona ainda a possibilidade de efetuar *queries* em paralelo, funcionalidade que está em desenvolvimento no PostGres.

4.6 – Comandos e exemplos

Um plano de execução pode ser examinado chamando o comando EXPLAIN, que junto trará as estatísticas associadas.

A sintaxe genérica é a seguinte:

```
EXPLAIN options query
```

Onde options pode ser uma das seguintes possibilidades:

• ANALYSE: leva a que o comando seja mesmo executado, dando os valores reais de *runtime*, em vez de apenas estima-los. Útil também para atualizar as estatísticas do sistema.

• VERBOSE: apresenta mais informação sobre o plano a ser executado, por exemplo funções ou *triggers* usados para obter o resultado final.

• COSTS ou TIMING: apresentam estimativas de tempo e de número de tuplos resultantes mais pormenorizadas (ao nível dos nós do plano).

• FORMAT: específica o modo como o output é apresentado. Entre as que existem salienta-se o XML e o JSON por serem formatos de texto úteis para a leitura de dados por parte de outros programas.

De seguida são ilustrados alguns exemplos de utilização.

```
EXPLAIN ANALYSE SELECT * FROM city
```

Com o output:

```
                                QUERY PLAN
-----------------------------------------------------------------
Seq Scan on city (cost=0.00..2.21 rows=100 width=1024)
                 (actual time=0.08..1.00 rows=87 loops=1
 Total runtime: 0.679 ms
(2 row)
```

E num caso em que utilize um índice:

```
EXPLAIN SELECT pop FROM city; where pop < 100000
```

Com output:

```
                                QUERY PLAN
———————————————————————————————————————————————————————————————
                  Index Scan using pop on city
                 (cost=0.00..0.76 rows=10 width=1024)
                 Index Cond: (pop < 100000)
                 (2 row)
```

Veja-se que em ambos os casos apresentam estimativas do número de linhas a devolver (“rows=”)e do intervalo de tempo que leva a executar (“cost=”).

Já o primeiro plano apresenta um novo conjunto de valores devido à real execução da *query* com a inserção da opção ANALYSE, onde os valores fornecidos pelo “actual time” e o “Total runtime” são concretos.

No segundo exemplo apresentado é de salientar que o plano utilizou um índice para efetuar a *query* (“Index Scan using pop on city”) melhorando as estimativas.

É também importante salientar a utilidade da mnemónica PREPARE que é utilizada para preparar queries em que seja necessário utilizar parâmetros, por exemplo:

```
PREPARE query(int,int) AS SELECT $1
FROM city
WHERE pop > $1 AND pop < $2
```

Podendo de seguida efetuar o plano de execução:

```
EXPLAIN EXECUTE query(100000, 500000)
```

No Postgres, ao contrário do Oracle, não existe meio de forçar os planos de execução a utilizar um algoritmo escolhido pelo utilizador, ou seja esse tipo de decisões cabe ao optimizador.

Existe a possibilidade de desligar certos parâmetros e tentar simular Query Hints, mas no entanto é uma forma infrutífera de chegar ao mesmo tipo de configuração. Como exemplo, um desses comandos pode ser o de desligar o Sequencial Scan:

```
SET ENABLE_SEQSCAN TO OFF
```

5 – Gestão de transacções e controlo de concorrência

5.1 - Conceitos

O conceito de transacção é fundamental em todos os sistemas de bases de dados. Esta agrupa um conjunto de várias alterações a realizar a vários campos da base de dados, em que para o utilizador final é visível como sendo apenas uma única operação, como por exemplo, ao realizar uma transferencia bancaria de uma conta para outra.

Uma transacção no sistema de base de dados PostgreSQL, tal como no Oracle, tem de respeitar um conjunto de propriedades denominadas como propriedades ACID. Estas propriedades correspondem à atomicidade, consistência, isolamento e durabilidade de uma transacção. Uma transacção diz-se atómica quando tem um comportamento do tipo “all-or-nothing”, isto é, ou todas as suas operações se realizam ou na falha de alguma, nenhuma se processa. A consistência implica que todas as restrições de sistema definidas têm de ser respeitadas aquando o processamento do resultado final de uma transacção. A propriedade relacionada com o isolamento obriga a que uma transacção, que está a ser processada em concorrência com um conjunto de outras transacções, não é afectada pelo comportamento das outras transacções em processamento concorrente. Por último, a durabilidade é a propriedade que diz que quando uma transacção é processada sem qualquer tipo de falha intermédia, as várias alterações realizadas por essa transacção têm que se manter no tempo, independentemente caso exista alguma falha como falta de corrente, problemas de hardware ou software.

Uma *nested transaction* é uma transacção que é iniciada dentro do escopo de uma transacção que já esteja a ser processada. Comparativamente com o Oracle o PostgreSQL também permite a realização *nested transactions.* Este conceito pode ser aplicado utilizando *savepoints* tal como mostra o seguinte exemplo:

```
BEGIN;
        INSERT INTO table1 VALUES (1);
        SAVEPOINT my_savepoint;
        INSERT INTO table1 VALUES (2);
        ROLLBACK TO SAVEPOINT my_savepoint;
        INSERT INTO table1 VALUES (3);
COMMIT;
```

Savepoints PostgreSQL

É possível ao utilizador realizar todo o tipo de operações após o *savepoint* ser definido, correspondendo desta forma, esse conjunto de operações, a uma "inner transaction". Quando se realiza o *rollback* a transacção inicial volta ao estado anterior à definição do *savepoint*. No exemplo acima apresentado, apenas os valores 1 e 3 serão inseridos na table1, visto que a inserção do valor 2 se encontra dento do *savepoint* definido como my_savepoint.

5.2 - Métodos de Isolamento

Apesar do standard do SQL definir quatro métodos de isolamento e de todos eles poderem ser solicitados, o PostgreSQL apenas implementa três desses métodos. Esses métodos são denominados como *Read Committed Isolation Level*, *Repeatable Read Isolation Level* e o *Serializable Isolation Level*. Também é possível definir o modo *Read Uncommitted Isolation Level*, mas internamente é utilizado o *Read Committed*. A razão pela qual o PostgreSQL apenas implementa três modos de isolamento deve-se ao facto de esta ser a única maneira viável para mapear os modos de isolamento standard numa arquitectura de controlo de concorrência multiversão.

```
SET TRANSACTION transaction_mode [, ...]
SET TRANSACTION SNAPSHOT snapshot_id
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]

where transaction_mode is one of:

        ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ
                COMMITTED | READ UNCOMMITTED }
        READ WRITE | READ ONLY
        [ NOT ] DEFERRABLE
```

Definir nível de isolamento de uma transacção em PostgreSQL

Read Committed Isolation Level

O *Read Committed Isolation Level* apresenta-se como o nível de isolamento usado por defeito no PostgreSQL. Quando uma transacção usa este nível de isolamento, as operações que se encontram dentro da transacção, apenas têm acesso a informação da base de dados que foi committed antes de uma qualquer query se iniciar. Nunca vê informação uncommitted ou alterações committed, de transacções concorrentes, durante a execução da query. Por outro lado todas as alterações feitas à base de dados por queries de uma transacção são visíveis pelas

restantes queries dessa mesma transacção mesmo que a transacção não tenha sido committed.

Repeatable Read Isolation Level

O *Repeatable Read Isolation Level* apenas vê a informação da base de dados antes da transacção se iniciar. Desta forma, uma transacção após se iniciar, nunca vê a informação da base de dados que foi committed por transacções concorrentes. Por outro lado uma das query de uma determinada transacção vê a informação alterada, por outras queries dessa mesma transacção, na base de dados. Este nível de isolamento apresenta-se diferente do anterior (*Read Committed Isolation Level*), visto que este usa, ao iniciar uma transacção, o mesmo estado da base de dados, para todas as suas queries enquanto que o *Read Committed* usa o estado da base de dados corrente aquando o inicio de cada uma das queries de uma transacção.

Serializable Isolation Level

O *Serializable Isolation Level* é o método de isolamento mais meticuloso de todos os apresentados anteriormente. Este processa todas as transacções serialmente (em serie), uma a seguir à outra, em vez de as processar concorrentemente.

Comparação com o Oracle

No que diz respeito aos métodos de isolamento, comparando os métodos disponibilizados pelo PostgreSQL e pelo Oracle, chegamos à conclusão que o Oracle apenas permite o uso do *Read Committed Isolation Level e do Serializable Isolation Level*, sendo o primeiro usado como default.

5.3 - Controlo da Concorrência

O PostgreSQL fornece um conjunto de ferramentas de controlo de concorrência para o acesso aos dados. Internamente, a consistência dos dados é mantida através da utilização do modelo de controlo de concorrência multiversão (MVCC). Com este modelo as transacções estão protegidas de aceder a dados inconsistentes que podem ser gerados, através de alterações à mesma informação realizada por transacções concorrentes, fornecendo isolamento a cada transacção. A principal vantagem de usar o modelo MVCC comparativamente aos locks tradicionais é que no modelo MVCC os locks sobre operações de leitura não influenciam as operações de escrita nem o contrário.

O PostgreSQL permite locks ao nível de toda a tabela ou apenas ao nível de um determinado tuplo.

Locks ao Nível da Tabela

A lista abaixo apresentada os modos de locks associados a uma tabela disponíveis e em que contexto são usados automaticamente pelo PostgreSQL

- Access Share: instruções, como o SELECT, que apenas consultam dados e não os modificam obtêm este tipo de lock.
- Row Share: Os comandos SELECT FOR UPDATE/SHARE obtêm este tipo de lock.
- Row Exclusive: Este tipo de lock é obtido por qualquer instrução que modifique dados numa determinada tabela, como por exemplo o UPDATE, DELETE ou o INSERT.
- Share Update Exclusive: Este tipo de lock é obtido por instruções VACUUM, ANALYZE, CREATE INDEX CONCURRENTLY.
- Share: Adquirido por CREATE INDEX.
- Share Row Exclusive: Este tipo de lock não é adquirido automaticamente
por qualquer comando PostgreSQL.
- Exclusive: Este tipo de lock não é adquirido automaticamente sobre uma tabela por qualquer comando PostgreSQL.
- Access Exclusive: Este comando é adquirido por ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER e VACUUM FULL. Este é também o modo por defeito do LOCK TABLE se nenhum modo for especificado.

Locks ao Nível de um Tuplo

O PostgreSQL também permite a utilização de locks associados a um tuplo de uma determinada tabela. Estes locks podem ser exclusivos ou partilhados. Um lock exclusivo, atribuído a um determinado tuplo, significa que o lock é adquirido sempre que são feitas operações de update ou delete sobre esse tuplo. O lock permanece ativo até a transacção fazer commit ou rolls back, exatamente como os locks ao nível da tabela. Estes locks bloqueiam apenas as escritas sobre o tuplo a que estão associados, permitindo, desta forma, queries concorrentes realizarem leituras desse mesmo tuplo. Para se adquirir um lock exclusivo sobre um determinado tuplo basta fazer-se um SELECT FOR UPDATE. Para se associar um lock partilhado a um determinado tuplo basta fazer-se um SELECT FOR SHARE. Desta forma nenhuma transacção pode realizar operações de update ou delete sobre um tuplo que está a ser utilizado noutra transacção. Qualquer tentativa deste tipo ira levar ao bloqueio até o lock ser libertado.

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

where ***lockmode*** is one of:

ACCESS SHARE I ROW SHARE I ROW EXCLUSIVE
I SHARE UPDATE EXCLUSIVE I SHARE I SHARE ROW EXCLUSIVE
I EXCLUSIVE I ACCESS EXCLUSIVE

Forma de obter um lock no PostgreSQL

Deadlocks

Um deadlock é uma situação na qual duas acções concorrentes se encontram à espera que uma e outra terminem, o qual não acontece (exemplo abaixo). O uso de um lock explicito pode aumentar a possibilidade de ocorrência de deadlocks. O PostgreSQL detecta automaticamente situações de deadlocks e resolve-os abortando uma das transacções que estão envolvidas, permitindo desta forma que a outra transacção possa concluir o seu processamento. A decisão de qual das transacções será abortada não é determinista.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Exemplo de deadlock

No exemplo acima apresentado o deadlock é verificado quando a transacção quatro realiza o lock-S(B), ficando bloqueada, visto que a transacção três ainda não libertou o lock sobre o B. A transacção três fica igualmente bloqueada quando realiza o lock-X(A), tendo este ainda não ter sido libertado pela transacção quatro, ocorrendo, desta forma, um deadlock.

5.4 - Atomicidade e Durabilidade

Como qualquer sistema de base de dados os PosgreSQL também necessita de ter um mecanismo de recuperação quando acontece alguma falha do sistema, como falhas de corrente ou problemas de hardware ou software. O postgreSQL implementa um sistema designado *Write-Ahead-Logging* (WAL). Este sistema é o método standard para manter a consistência da informação. A ideia base deste conceito é que todas as alterações feitas à base de dados apenas são escritas depois de terem sido escritas no log. Desta forma, não é necessário escrever imediatamente todas as alterações feitas quando uma transacção faz commit, pois sabemos que caso haja alguma falha o sistema tem a capacidade de recuperar todas as alterações feitas até então através do log. Para além disso, o WAL também acrescenta melhorias significativas relativamente ao número de escritas em disco, reduzindo este número significativamente (escrita sequencial em disco).

6 – Suporte para base de dados distribuídas

O core do PostgreSQL não tem suporte para base de dados distribuídas, não sendo possível replicar os dados por diferentes computadores. Para colmatar esta deficiência, existem alguns projectos open source sendo os mais conhecidos de Rubyrep e PostgreSQL-XC, que para além de terem o core do PostgreSQL também fornece as funcionalidades de *multi-master replication*. Sendo o trabalho apenas sobre o PostgreSQL não iremos abordar esta temática.

7 – Outras características

7.1 – Comando VACUUM

Em qualquer sistema de gestão de base de dados existe a constante preocupação de fazer a “limpeza” de dados que já não estejam a ser utilizados. Aqui entra a chamada **Routine Vacuuming** implementada pelo Postgres.

No caso do Postgres as alterações feitas nos tuplos ou a tabelas inteiras (UPDATE, DELETE, DROP, etc.) não removem imediatamente as suas versões antigas, para que estes estejam visíveis em transações concorrentes.

Mas mais tarde ou mais cedo esses dados não estarão a ser usados levando o mecanismo a remover versões de tuplos em tabelas e índices, marcando o espaço como disponível internamente à base de dados e não para o sistema operativo (a não ser que a libertação de espaço seja significante). Existe ainda o modelo VACUUM FULL que opta por rescrever tabelas inteiras para não deixar espaços vazios.

Este mecanismo está ativado e será realizado de acordo com o agendamento estipulado por defeito. No entanto, para melhorar a sua eficácia e utilidade, a sua execução é ativada em caso de ser chamado diretamente (VACUUM), através do comando ANALYSE ou em outro tipo de comando como CREATE INDEX, pois estes últimos dois enunciados podem alterar significativamente o espaço utilizado. Apesar de ser aconselhável a sua utilização, esta pode ser desativada, passando a sua execução a ser opcional por parte dos administradores.

7.2 – Tipos de dados

O Postgres abrange um vasto leque de tipos de dados para armazenamento de informação destacando-se alguns tais como formas geométricas, vetores ou tipos compostos (estruturas de dados). Mas nesta secção será dado o destaque aos formatos de representação de dados **XML** e **JSON**, pois estes são dos mais usados e muito relevantes nas trocas de dados entre domínios (e.g. cliente-servidor).

Assim, para que a manipulação e interpretação deste formato no estado interno da base de dados fosse possível, foram desenvolvidas as ferramentas e mecanismos necessárias para que tal aconteça. No entanto existem algumas limitações pois este tipo de dados não suporta operações de comparação (=, <=, >=) nem permite criar índices sobre este tipo de atributo.

O suporte para atributos com tipo de dados XML é exemplificado de seguida. No entanto estes documentos em XML têm que estar bem formatados, caso contrário o Postgres rejeita a informação fornecida.

```
CREATE TABLE example (
                id integer,
                    data xml
)
```

De modo a facilitar a forma como estes dados são manuseados existem duas funções essenciais que fazem conversões de dados:

- XMLPARSE – traduz um documento ou conteúdo textual numa árvore em formato XML
- XMLSERIALIZE – traduz os dados XML guardados para um formato textual

Temos os seguintes exemplos para demonstração:

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

```
XMLSERIALIZE (DOCUMENT query_to_xml('select * from t) as text)
```

Existem também um conjunto de funções para a construção "de raiz" de um documento XML, sendo alguns dos mais relevantes:

- XMLELEMENT – cria um nó na árvore XML
- XMLPI – permite introduzir expressões de outros tipos de linguagens de programação ou representativas
- XMLCONCAT- junta dois excertos de conteúdo XML
- ...

Dando um exemplo:

```
XMLROOT (
        XMLELEMENT (
            NAME elem,
            XMLATTRIBUTES (
                    'mojo' AS name,
                10+1 AS num
            ),
            XMLELEMENT (
                    NAME foo,
                bar
            )
        ),
    VERSION '1.0',
    STANDALONE YES
)
```

E respetivo resultado:

```
<?xml version='1.0' standalone='yes' ?>
  <elem name='mojo' num='11'>
    <foo>bar</foo>
  </elem>
```

A implementação deste tipo de dados não estaria completa se não existisse uma maneira de procurar valores/nós específicos num atributo XML, por isso, existe um mecanismo XQuery para efetuar perguntas em formato XPath sobre o XML. Estas funções são:

- XPATH (xpath, xml) – a função "xpath" é avaliada sobre os dados "xml" e devolve um vetor com os elementos encontrados.

- XPATH_EXISTS (xpath, xml) – faz o mesmo tipo de avaliação mas apenas devolve se existe o caminho "xpath" (num booleano verdadeiro ou falso)

Referentemente ao formato JSON, o Postgres oferece um tipo de dados para que se possa guardar a sua informação com validação dos valores inseridos. Tal como no tipo de dados XML também existem funções que permitem operações tais como procura e extração de valores ou conversão de/para o tipo vetor:

- array_to_json – transforma um vetor num vetor de elementos JSON

- row_to_json – transforma um tuplo em JSON

- to_json – transforma um elemento fornecido em JSON. Se não conseguir converte para um formato textual

- json_each – recebe um elemento JSON e é mais usado para procuras (select * from t1 json_each(json))

- ...

Existem mais funções do género do json_each, no entanto não serão aqui mencionadas por ser uma lista extensa e com sintaxes mais complexas, servido o json_each como representante do propósito dessas operações (pesquisas, seleções, filtragens, etc.).

7.3 – Serviços de conexão com base ODBC e JDBC

Para que seja possível a comunicação com as base de dados relacionais o Postgres oferece dois conjuntos distintos de API para que tal seja possível.

A primeira é a **JDBC** (Java Database Connectivity) que tem o seu conjunto de classes e interfaces escritas em Java para efetuar a conexão com as bases de dados SQL. Na definição geral de JDBC podem existir quatro tipos de driver, no entanto o JDBC do Postgres só fornece um deles, o de tipo 4. Este tipo de driver (também chamado de driver nativo) é escrito totalmente em Java o que faz com que seja independente da plataforma onde é usado pois uma vez compilado o driver corre em qualquer sistema. De seguida são ilustrados alguns exemplos básicos de como se utiliza a API da JDBC.

Para aceder à base de dados basta efetuar o seguinte código:

```
Connection conn = Driver.getConnection(url,username,password);
```

Com a conexão validada é possível efetuar *queries* como no seguinte exemplo:

Mas um dos objetivos deste modelo de conexão com base de dados é efetuar *queries* mais complexas e que também dependam de parâmetros escolhidos pelos utilizadores, como no seguinte exemplo:

```
int pop = 500000;
PreparedStatement st =
conn.prepareStatement("SELECT * FROM t1
                  WHERE population >= ?");

st.setInt(1, pop);

ResultSet rs = st.executeQuery();

while (rs.next())
{

System.out.print("Attribute 1 is: ");

System.out.println(rs.getString(1));
}
rs.close();
st.close();
```

Mas como é sabido, o Java não é a única linguagem de programação utilizada, por isso tem que existir maneira de utilizar outras linguagens para conectar com as base de dados.

Assim, para que o acesso a base de dados seja mais homogéneo existe o **ODBC** (Open Database Connectivity), sendo que no Postgres o driver ODBC oficial é chamado de **psqlOBDC**. Este modelo define um conjunto de interfaces para várias linguagens (Visual Basic, Visual C++, Delphi, etc.) de modo a garantir o acesso a diferentes base de dados (MySQL, SQL Server, etc.) sem necessidade de codificar métodos especializados ou de alterar significativamente camadas de dados. Ou seja, o OBDC é independente da base de dados porque o respetivo driver trabalha como uma camada de tradução entre as aplicações e os sistemas de gestão de base de dados.