Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Mestrado Integrado em Engenharia Informática

Sistemas de Bases de Dados

Estudo sobre o SGBD PosgreSQL 9.3

Realizado por:

Grupo 20:

António Cambeiro
João Cambeiro
Vladislav Pinzhuro

1 de Junho de 2014

Índice

1 Introdução **3**

1.1 Introdução histórica 3

2 Estrutura de ficheiros **4**

2.1 Organização dos ficheiros das tabelas e índices 6

2.2 *Visibility* e *Free Space Maps* 7

2.3 TOAST 8

2.4 Clustering de tabelas 9

2.5 Partição de tabelas 9

2.6 Buffer Management 10

3 Indexação e Hashing **12**

3.1 Uso de Tabelas de Hash 14

3.2 Estruturas Temporariamente Inconsistentes 15

4 Processamento e Optimização de Perguntas **16**

4.1 O percurso duma pergunta no sistema 16

4.2 Estabelecimento de Conexão 17

4.3 Parsing 17

4.4 Reescrita 18

4.5 Planeamento/Optimização 19

4.5.1 Pesquisa exaustiva 21

4.5.2 Algoritmo genético 26
4.6 Execução 27

5 Gestão de Transacções e Controlo de Concorrência 28

5.1 Níveis de Isolamento 28
5.2 Locking explícito 29
5.2.1 Row-level locks 32
5.2.2 Advisory Locks 32
5.2.3 Locking e índices 32
5.2.4 Recuperação PITR (Point-in-time Recovery) 33

6 Suporte para bases de dados distribuídas 34

6.1 Log Shipping/Warm Standby 34
6.2 Streaming Replication 34
6.3 Cascading Replication 34
6.4 Synchronous Replication 35
6.5 Comparação com Oracle 11g 35

7 Outras características 36

7.1 Suporte para XML 36
7.2 Segurança 37

8 Bibliografia 39

9 Anexos 40

1 Introdução

Este trabalho foi realizado no âmbito da cadeira de Sistemas de Bases de Dados e tem como objectivo o estudo do PostgreSQL 9.3. Cobre uma análise do sistema sobre as vertentes leccionadas da cadeira e faz uma comparação com o Oracle 11g ao longo do documento.

1.1 Introdução histórica

O PostgreSQL surgiu de um projecto académico chamado Ingres, da universidade de Berkeley. Em 1982 o líder do projecto, prof. Michael Stonebraker, abandonou Berkeley com o objectivo de criar uma versão proprietária do Ingres. A tentativa falhou devido a problemas/limitações do sistema e trouxe Michael de volta(1985) a Berkeley com o intuito de retomar o projecto (desta feita com o nome Postgres) e melhorá-lo.

Em 1986 foram revelados documentos descritivos do sistema e em 1988 foi apresentado um protótipo funcional do mesmo. Foram lançadas versões até 1993, altura em que o projecto foi descontinuado.

Em 1994 os estudantes Andrew Yu e Jolly Chen retomaram o projecto e substituíram o interpretador do Postgres por um baseado em SQL; surge então o Postgre95 e é disponibilizado na web em *Open Source*.

Em 1996 o projecto é renomeado para PostgreSQL por forma a reflectir a nova linguagem que suportava o sistema:SQL. Apartir daqui, as novas versões que têm "saído" tem tido o contributo de developers voluntários espalhados pelo mundo e tem tido o apoio de algumas empresas.

2 Estrutura de ficheiros

Normalmente os ficheiros de configuração e ficheiros de dados do cluster da base de dados, estão guardados dentro da pasta de dados do cluster. Esta localização é definida pela variável de sistema `PGDATA`. É possível existirem vários clusters na mesma máquina desde que geridos por instâncias diferentes de servidores. Dentro da diretoria `PGDATA/` encontramos os ficheiros de configuração `postgresql.conf`, `pg_hba.conf` e `pg_ident_conf` e as subdirectorias e ficheiros de controlos apresentados na tabela 1.

Cada tabela e índice são guardados em ficheiros separados em disco. Estes ficheiros persistentes têm o nome do *filenode* da tabela ou índice. Para relações temporárias o nome do ficheiro é da forma `BBB_FFF` em que BBB é o ID do *backend* que criou o ficheiro e FFF é o número do *filenode*.

Em tabelas ou índices com um tamanho superior a 1GB, a tabela ou índice é dividida em segmentos de 1GB em que o nome do primeiro é *filenode* e nome dos segmentos seguintes é filenode.1, filenode.2, etc. Este tamanho definido por defeito pode ser alterado usando a opção de configuração `--with-segsize` na fase de *build* do PostgreSQL.

Os ficheiros temporários necessários a operações que não caibam em memória, são criados na pasta `PGDATA/base/pgsql_tmp`.

O PostgreeSQL utiliza o sistema de ficheiros oferecido pelo sistema operativo onde o SGBD foi instalado. Desta forma existe desta forma uma menor optimização das operações necessárias ao funcionamento do PostgreSQL.

No SGBD da Oracle não é utilizado o sistema de ficheiros do sistema operativo onde o sistema foi instalado, neste caso é usado um sistema proprietário da Oracle que faz a gestão dos ficheiros utilizados pelo SGBD. Esta abordagem introduz uma maior optimização, uma vez que, é apenas necessário suportar as operações necessárias ao funcionamento do SGBD e a implementação destas operações é feita levando em conta a especificidade destas mesmas operações.

Item	Descrição
`PG_Version`	Ficheiro que contém a versão do PostgreSQL
`base`	Cada subdiretoria pertence a cada uma das bases de dados instaladas no SBD
`global`	As tabelas partilhadas por todas as bases de dados pertencentes ao *cluster*
`pc_clog`	Subdiretorias que contém a informação sobre o estado de *commit* das transações
`pg_multixact`	Subdiretoria que contém a informação sobre o estado de multi-transações. Esta informação é utilizada para *locks* sobre tuplos partilhados
`pg_notify`	Subdiretoria que contém a informação sobre as notificações do sistema
`pg_serial`	Subdiretoria que contém a informação sobre as transações serializáveis em estado *commited*
`pg_snapshots`	Subdiretoria que contém a informação sobre os *snapshots* exportados
`pg_stat_tmp`	Subdiretoria em que são guardados ficheiros temporários utilizados pelo subsistema responsável pelas estatísticas do SGDB
`pg_subtrans`	Subdiretoria onde são guardados os dados das subtransações
`pg_tblspc`	Subdiretoria que contém os link simbólicos para os *tablespaces*
`pg_twophase`	Subdiretoria que contém os ficheiros relativos ao estado das transações que utilizam o *two-phase commit*
`pg_xlog`	Ficheiros que contem os ficheiros WAL (*Write Ahead Log*)
`postmaster.opts`	Um ficheiro que guarda as opções iniciais relativas à linha de comandos

Tabela 1: Estrutura de ficheiros PosgreSQL

Item	Descrição
`PageHeaderData`	Com 24 bytes. Possui apontadores para o espaço livre e informação sobre a página
`ItemIdData`	Um *array (offset, length)* apontando para os itens da página. 4 bytes por item
`Freespace`	Espaço livre
`Items`	Os itens contidos na página
`Special Space`	Informação varia consoante a técnica de indexação utilizada. Vazio em caso de tabelas normais

Tabela 2: Slotted Page

2.1 Organização dos ficheiros das tabelas e índices

No PostgreSQL uma tabela ou um índice é guardado num *array* de páginas com um tamanho fixo de 8kB, não havendo suporte para a existência de diferentes tamanhos na mesma instalação. Este tamanho fixo pode ser alterado na altura de compilação do servidor alterando o campo `BLCKSZ` no ficheiro `pc_config_manual.h`. Depois da alteração é necessário recompilar o código do servidor.

No PostgreSQL os registos podem ter tamanho variável, mas não podem ocupar mais de uma página.

Numa tabela todas as páginas são logicamente equivalentes e um tuplo pode ser guardado em qualquer página, mas no caso de índices, a primeira página é reservada como uma *metapage* e nela consta informação de controlo. No caso dos índices também podem existir tipos diferentes tipos de páginas consoante o tipo de indexação utilizado.

A especificação duma *slotted page* encontra-se na tabela 2.

Os itens são inseridos a partir do fim do espaço livre e a sua estrutura varia consoante o tipo de informação guardada na tabela. Tabelas e sequências usam a estrutura *HeapTupleHeaderData*. Como referência apresentamos a descrição desta estrutura na tabela 3. O Oracle também utiliza *slotted pages*, suportando desta forma registos de tamanho variável.

Field	Type	Length	Description
`t_xmin`	TransactionId	4 bytes	insert XID stamp
`t_xmax`	TransactionId	4 bytes	delete XID stamp
`t_cid`	CommandId	4 bytes	insert and/or delete CID stamp (overlays with `t_xvac`)
`t_xvac`	TransactionId	4 bytes	XID for VACUUM operation moving a row version
`t_ctid`	ItemPointerData	6 bytes	current TID of this or newer row version
`t_infomask2`	uint16	2 bytes	number of attributes, plus various flag bits
`t_infomask`	uint16	2 bytes	various flag bits
`t_hoff`	uint8	1 byte	offset to user data

Tabela 3: Slotted Page

2.2 *Visibility* e *Free Space Maps*

Cada ficheiro que contém uma relação ou um índice (excepto para o caso de um índice hash) tem associado um *Visibility Map* e um *Free Space Map*.

Um *Visibility Map* é caracterizado por ser o local onde é guardada a informação das páginas que contêm tuplos visíveis para todas as transações activas. Esta informação é guardada num ficheiro, com o mesmo nome da relação acrescido do sufixo `_vm`.

Num *Free Space Map*, é guardada a informação sobre o espaço livre da relação. O mapa é guardado num ficheiro, com o mesmo nome da relação acrescido do sufixo `_fsm`.

2.3 TOAST

Como o PostgreSQL usa um tamanho de página fixo (8kB) e não permite que um tuplo ocupe mais de 1 página, houve que encontrar uma forma de guardar campos com tamanho grande.

A técnica TOAST (*The Oversized-Attribute Storage Technique*) consiste em comprimir ou dividir em várias linhas os valores de campos com tamanho grande.

Um tipo de dados para suportar a técnica TOAST deve ter um tamanho variável, em que a primeira palavra de 32 bits contém o tamanho total em bytes. Se um ou mais atributos de uma tabela forem passíveis de aplicar a técnica TOAST, será criada uma tabela TOAST e o seu OID guardado no ficheiro `pg_class.reltoastrelid` da tabela.

Existem quatro técnicas para guardar valores de colunas TOAST:

- PLAIN – não é utilizada a compressão ou a divisão dos dados. É a única estratégia disponível para tipos de dados, a que não é possível aplicar a técnica TOAST.
- EXTENDED – É utilizada a compressão e a divisão de dados. Caso a compressão não seja suficiente os dados são divididos. É a opção *default* do SGBD.
- EXTERNAL – Usa a divisão dos dados mas não a compressão. Para alguns tipos de dados aumenta a performance de operações de subelementos, mas aumenta as necessidades ao nível de espaço.
- MAIN – Usa a compressão como método preferencial. A divisão só será utilizada caso a compressão não seja possível.

Como grande parte das operações funcionam sobre comparações de pequenas chaves, apenas o registo principal dos tuplos será analisado. Isto permite que os valores TOAST apenas sejam recuperados quando têm de ser apresentados ao utilizador. Também se obtêm benefícios em operações de ordenação porque aumenta a probabilidade da tabela principal caber em memória, uma vez que não é necessário carregar a tabela TOAST.

2.4 Clustering de tabelas

No PostgreSQL não é suportado *Multitable Clustering*, contrariamente ao SGBD Oracle. Desta forma, operações frequentes, que operem sobre informação que resulta de junções de tabelas, são mais eficientes no SGBD da Oracle. Apenas é suportado *table clustering* que permite reordenar fisicamente o ficheiro da tabela com a ordem determinada por um índice. Durante a operação de *clustering* um ACCESS EXCLUSIVE lock é adquirido, não permitindo operações de leitura ou de escrita na tabela. Caso operações subsequentes à operação de *clustering* alterem a tabela, é necessário voltar a executar a operação.

Apresentamos os seguintes exemplos:

Operação de *cluster* sobre a tabela *employees* usando o índice `employees_ind`.

```
CLUSTER employees USING employees_ind;
```

Operação de *cluster* sobre a tabela *employees* utilizando o índice previamente definido.

```
CLUSTER employees;
```

Operação de *cluster* sobre todas as tabelas que tenham sido alvo de uma operação de *cluster* anteriormente.

```
CLUSTER;
```

2.5 Partição de tabelas

O PostgreSQL suporta particionamento de tabelas em tabelas de menor dimensão. Esta técnica permite uma melhor performance quando os tuplos mais acedidos se encontram na mesma partição ou num número pequeno de partições. Em operações de consulta ou atualizações em que seja necessário operar sobre um elevado número de tuplos de uma partição, é mais favorável realizar uma busca sequencial sobre a partição do que realizar uma busca sobre a tabela completa, mesmo utilizando um índice ou acessos aleatórios.

O PostgreSQL suporta dois tipos de partição de tabelas:

- *Range Partitioning* – A tabela é particionada por em intervalos de valores. Estes intervalos são definidos em função de uma chave ou de um conjunto de colunas, não existindo interseção de valores entre diferentes intervalos.
- *List Partitioning* - A tabela é particionada e é definido que chaves compõem cada uma das partições.

Para criar partições a partir de uma tabela é necessário criar as tabelas que servirão para representar as partições, depois é necessário adicionar as restrições para cada uma das partições de forma a que não exista *overlap* entre valores de diferentes partições e por fim a criação de *triggers* para os dados sejam inseridos nas partições corretas.

O SGBD Oracle adiciona aos tipos de particionamento suportados pelo PostgreSQL o *hash partitioning*. Esta técnica de *hash* tem como objectivo uma distribuição mais equitativa dos elementos entre as diferentes partições.

2.6 Buffer Management

O sistema de *buffer management* tem como objectivo reduzir o número de operações sobre os discos rígidos dos servidores. Este tipo de operações tem um impacto negativo elevado na performance. O *buffer* é implementado num array circular de buffers e cada entrada na cache tem 8kB. O PostgreSQL implementa um algoritmo de *clock sweep*. Neste tipo de algoritmo é executada uma navegação sequencial e quando é atingido o final do array de *buffers* o próximo passo da navegação é a posição inicial do *array*. Como o PostGreSQL utiliza o sistema de ficheiros do sistema operativo do servidor onde se encontra instalado, em muitos casos os blocos presentes na cache do sistema operativo relacionados com o PostgreSQL são os mesmos presentes no *array* de *buffers* do SGBD. Desta forma não é necessário alocar uma quantidade de memória grande ao SGBD, mas é preciso ter em atenção não atribuir memória a menos, porque os sistemas de cache utilizam o menos eficiente LRU invés do mecanismo *clock sweep*.

Os *buffers* na cache podem estar marcados da seguinte forma:

- PINNED – Significa que um processo adquiriu um lock sobre o *buffer* e nenhum outro processo lhe pode aceder até o processo libertar o lock.
- DIRTY – Significa que o *buffer* foi alterado desde que foi lido a partir do disco.

Quando é lido um bloco a partir do disco, o *buffer management* seleciona o *buffer* a ser ocupado utilizando o algoritmo clock sweep e o primeiro bloco com `USAGE_COUNT` = 0 é despejado. Caso este *buffer* esteja marcado como dirty é primeiro escrito em disco. O sistema de buffer management também está presente no SGBD Oracle.

3 Indexação e Hashing

O PostgreSQL suporta os seguintes tipos de indexação:

- BTree – Este tipo de índice funciona bem em consultas sobre igualdades ou intervalos sobre dados que podem ser ordenados. É o índice criado por defeito, através do comando `CREATE INDEX`
- Hash – São apenas utilizados em operações de comparação (operador =). O seu uso é desencorajado devido a problemas relacionados com falhas do sistema. Podem ser criados através do comando:
  `CREATE INDEX name ON table USING hash (column)`
- GiST - Mais do que um tipo de índice, o Gist (*Generalized Search Tree*) é uma infraestrutura que permite criar vários esquemas de indexação. Uma das principais vantagens é possibilitar a criação de novos tipos de dados e as melhores estratégias de acesso, por parte de um especialista do domínio dos dados, em vez de um especialista em base de dados
- SP-GiST – Suporta árvores de pesquisa particionadas, que facilitam o desenvolvimento de estruturas de dados não equilibradas como quad-trees, k-d trees e radix trees. As pesquisas que respeitem as regras de particionamento destas estruturas são muito eficientes
- GIN – Este tipo de índices suportam valores com mais de uma chave, pex. Arrays. A distribuição do PostgreSQL define classes de operadores que dão suporte a operações sobre arrays unidimensionais
- Multicolumn – Um índice pode ser ter como base várias colunas, facilitando *queries* que actuem num subconjunto das colunas. Podem ser criados sobre índices B-tree, GIN, Gist ou SP-Gist. Podem ser criados com o comando `CREATE INDEX test ON test2 (major,minor)`
- Índices Únicos – São utilizados para garantir que um valor numa coluna é único ou o valor combinado de várias colunas é único. Apenas são suportados em índices B-tree.Podem ser criados utilizando o comando:
  `CREATE UNIQUE INDEX name ON table (column [, ...])`

- Índices que usam expressões – O valor utilizado na indexação resulta da aplicação de uma função ou expressão escalar numa ou mais colunas da tabela. O valores resultantes da computação apenas têm de ser calculados quando são inseridos ou quando são actualizados e não de cada vez que se efetua uma busca utilizando o índice. Pode ser criado através do comando: `CREATE INDEX test1 ON table1 (lower(column1))`

- Índices Parciais – O índice é construído sobre um subconjunto da tabela. O subconjunto é definido através de um predicado. O subconjunto tem apenas os tuplos que cumprem esse predicado. Uma das principais vantagens deste tipo de índices é evitar a repetição dos valores indexados., diminuindo o número de valores indexados, aumentando a velocidade tanto de pesquisas como de acessos. É possível a criação de índices parciais únicos. Um índice parcial pode ser criado com o comando:
  ```
  CREATE INDEX bank_transactions_ix ON transactions (client_number)
  where NOT (client_number >100 AND client_number < 150)
  ```

- Índices múltiplos – Vários ficheiros de índices podem ser combinados com o fim de acelerar pesquisas em que as cláusulas AND ou OR sejam utilizadas. Neste caso são construídos *bitmaps* em memória e estes são preenchidos conforme respeitem as condições do índice. Por fim estes *bitmaps* são *ANDed* ou *ORed* (conforme as cláusulas). Por fim os tuplos da relação são revisitados para ser apresentado o resultado ao utilizador. O Oracle não suporta múltiplos índices embora através de *Hints* se possa sugerir o uso de *bitmaps*.

`Nota:` O esquema e outra informação adicional sobre a base de dados usada nos exemplos posteriores encontra-se na secção 9.

Neste exemplo é possível ver que o optimizador de *queries* utiliza o índice criado na altura da definição da chave primária da tabela customers. Por defeito estes índices são do tipo B-tree.

```
EXPLAIN ANALYZE SELECT * FROM customers WHERE customerid=1000;
QUERY PLAN
----------
Index Scan using customers_pkey on customers (cost=0.00..8.27
   rows=1
width=268) (actual time=0.029..0.033 rows=1 loops=1)
Index Cond: (customerid = 1000)
Total runtime: 0.102 ms
```

Neste exemplo é possível verificar que o optimizador de queries decidiu utilizar os dois índices disponíveis e depois, com a ajuda de um bitmap, realizar um AND nas duas condições da *query*.

```
EXPLAIN ANALYZE SELECT customerid,username FROM customers WHERE
customerid<10000 AND username<'user100';
QUERY PLAN
----------
Bitmap Heap Scan on customers (cost=5.71..370.28 rows=95 width
   =13)
(actual time=0.036..0.043 rows=2 loops=1)
Recheck Cond: ((username)::text < 'user100'::text)
Filter: (customerid < 10000)
-> Bitmap Index Scan on ix_cust_username (cost=0.00..5.69 rows
   =191
width=0) (actual time=0.019..0.019 rows=2 loops=1)
Index Cond: ((username)::text < 'user100'::text)
Total runtime: 0.099 ms
```

3.1 Uso de Tabelas de Hash

O PostgreSQL usa tabelas de hash para implementar índices hash. O tipo de hash utilizado é o hash dinâmico, uma vez que é previsível que a maioria das tabelas cresça com o tempo. Como foi referido anteriormente, o uso de índices do tipo hash não é aconselhado visto poderem surgir erros

quando o sistema recupera de falhas. Além disso os ganhos obtidos ao nível da performance não são significativos em relação ao uso de índices B+. No SGBD Oracle não existem índices *hash* mas existem *Hash Clusters* onde várias tabelas podem ser adicionadas ao cluster e usando uma *cluster key*, os tuplos são acedidos. Um índice *hash* do PostgreSQL pode ser adaptado ao Oracle, criando uma *Single-Table Hash Cluster* e utilizando como *cluster key* os atributos pretendidos. Neste último caso tem de haver uma relação um para um entre a *cluster key* e um tuplo da tabela.

3.2 Estruturas Temporariamente Inconsistentes

Uma das operações que compõem uma transação pode originar um estado inconsistente temporário da base de dados. No PostGreSQL é possível definir o nível de verificação das restrições que atuam sobre uma base de dados durante uma transação. Nas restrições criadas com `DEFERRABLE INNITIALY IMEDIATE`, em cada operação que compõem a transação a consistência da base de dados é verificada. Nas restrições criadas com `DEFERRABLE INITIALLY DEFERRED` o estado de consistência da base de dados apenas é verificado na altura de commit da transação.

O PostgreSQL não segue o SQL Standard para restrições NOT NULL e CHECK e para restrições non-deferrable do tipo UNIQUE imediatamente e não no final da operação, como é sugerido pelo standard.

Para criar uma restrição pode ser utilizada a seguinte sinopse:
SET CONSTRAINTS ALL — name [, ...] DEFERRED — IMEDIATE

No SGBD Oracle também são suportadas estruturas temporariamenteinconsistentes.

4 Processamento e Optimização de Perguntas

Esta secção descreve como o sistema processa e optimiza perguntas SQL. A secção 4.1 apresenta uma vista global sobre as fases pelas quais o sistema passa a processar e optimizar uma pergunta. Nas secções posteriores, descreve-se em pormenor todas as fases envolvidas neste processo.

4.1 O percurso duma pergunta no sistema

Depois de ser estabelecida uma *conexão* (secção 4.2) entre a aplicação cliente e o servidor, e depois de ser recebida, com sucesso, uma pergunta feita pelo cliente no servidor, este começa por passar a mesma ao *parser* (secção 4.3), cujo papel consiste em analisar perguntas perante existência de erros sintácticos. Em caso de ausência dos mesmos, a pergunta é transformada numa árvore de query *(query tree)*, sendo esta uma representação formal e não ambígua da pergunta em causa. Após ser criada, a árvore passa por uma fase de *rewrite* (secção 4.4), durante a qual o sistema aplica as regras que tenha guardadas nos seus catálogos *(system catalogs)*. Por exemplo, é nesta fase que as vistas são expandidas para as expressões originais correspondentes. De seguida, a árvore é passada ao *planner/optimiser* (secção 4.5), cujo papel consiste em percorrer a mesma e criar um plano de execução para a pergunta em causa. Nesta fase, são criadas todas as possíveis variantes de execução da pergunta, às quais também é associado uma estimativa de custo de execução. Por exemplo, são geradas variantes diferentes consoante a existência e disponibilidade de índices, ou, em caso duma pergunta que envolva junções, são consideradas todas as maneiras diferentes de as executar. No fim, o melhor plano, i.e., o plano com a menor estimativa de custo, é passado ao *executor* (secção 4.6), que executa o plano e devolve a resposta à aplicação cliente.

4.2 Estabelecimento de Conexão

O sistema utiliza um modelo simples de cliente/servidor, segundo o qual um processo de cliente pode estar ligado a exactamente um processo no servidor. Como não se sabe com antecedência quantas conexões serão eventualmente estabelecidas, é utilizado um processo mestre *(master)*, chamado `postgres`, cujo objectivo consiste em escutar uma dada porta TCP e criar um novo processo no servidor cada vez que é recebido um novo pedido de conexão. O novo processo, pela sua vez, executa autenticação do pedido e, em caso de sucesso, passa a ser o servidor *(backend)* para a respectiva conexão. Todos os processos no servidor utilizam semáforos e memória partilhada de modo a garantir a integridade e consistência dos dados durante acessos concorrentes. É de notar que *multithreading* não é suportado pelo sistema PosgreSQL. Desta maneira, é garantido que qualquer erro ou bloqueio num processo backend, causado pelas bibliotecas de autenticação não-multithreaded como SSL ou PAM, não cause erros a clientes que estejam a ser tratados por outros backends.

A partir do momento em que uma conexão é estabelecida com sucesso, o cliente pode enviar perguntas ao servidor. Todas as perguntas são transmitidas em pleno texto, sem qualquer parsing feito no cliente. Após receber uma pergunta, a mesma passa pelo processo de parsing, que é descrito em promenor na próxima secção.

4.3 Parsing

O processo de parsing em si envolve duas fases distintas:

1. **Parser**

   Verifica se a pergunta está de acordo com a gramática definida, ou seja, se está sintacticamente correcta. Utiliza um *lexer*, que é responsável por reconhecer todos os identificadores e as palavras reservadas do SQL e gerar tokens apropriados. Se a sintaxe da pergunta for válida, o parser constrói uma árvore de parsing *(parse tree)*.

2. ***Transformation Process***

   Recebe a árvore de *parsing* gerada na fase 1 e faz uma interpretação semântica sobre a árvore. Esta análise é necessária para saber que tabelas, funções e operadores são referenciadas pela pergunta em causa. Para representar esta informação, é construída uma árvore de query *(query tree)*, sendo esta mais específica em comparação com a árvore de parsing. Por exemplo, a árvore de query inclui informação sobre os tipos de dados dos atributos e dos resultados das expressões. Outro exemplo são as chamadas de funções, onde, numa árvore de parsing, algo que, do ponto de vista sintáctico, pareça uma chamada de função, é representado por um nó designado *FuncCall*. Por outro lado, numa árvore de query, o mesmo nó será transformado em nó *FuncExpr* ou *Aggref*, dependendo se, após a análise semântica, a expressão designar, respectivamente, uma função simples, ou uma função de agregação.

É necessário existir esta separação entre as duas fases porque os catálogos de sistema *(system catalogs)* podem ser acedidos apenas no âmbito duma transacção, e as transacções não começam assim que uma pergunta é recebida, pois não se sabe à priori se o corpo da pergunta apresenta uma sintaxe válida e se contém comandos SQL. A primeira fase de parsing é suficiente para conseguir identificar todos os comandos de controlo de transacções, tais como `BEGIN` e `ROLLBACK`, que vão poder ser executados sem qualquer análise posterior. Assim que o parser confirmar a validade sintáctica da pergunta e perceber que está a lidar com uma pergunta que contém comandos SQL, como `SELECT` ou `UPDATE`, uma transacção pode ser iniciada. É apenas nesta altura que o processo de transformação *(transformation process)*, descrito na fase 2, pode começar.

4.4 Reescrita

Esta fase é responsável por traduzir todas as regras *rules* definidas pelo utilizador. Os utilizadores podem criar regras que disparam perante alguns comandos, tais como `UPDATE`, `INSERT` ou `SELECT`. Uma vista (`VIEW`) é implementada pelo sistema através do sistema das regras, convertendo a definição da vista numa regra de selecção. Assim, sempre que é recebida uma pergunta

que envolva uma selecção numa vista definida anteriormente, a respectiva regra é aplicada, e a pergunta é reescrita usando a definição da vista.

A fase de reescrita começa por processar todas as comandos `UPDATE`, `DELETE` e `INSERT`, aplicando a cada um as regras apropriadas. Posteriormente, todas as regras que envolvam a cláusula `SELECT` são aplicadas. Como a aplicação duma regra pode causar reescrita da pergunta para uma forma intermédia que possa requerer a aplicação de outra regra (por exemplo, uma vista cuja definição referencia outra vista), todas as regras são verificadas perante a necessidade de serem aplicadas em cada forma intermédia da pergunta, até chegar a um ponto onde não haja possibilidade de aplicar mais regras.

A maior vantagem de implementar as vistas usando o sistema de regras consiste em fornecimento de informação adicional ao planeador (secção 4.5). Desta maneira, o planeador terá toda a informação sobre que tabelas terão de ser analisadas, que relações entre essas tabelas existem e que restrições impostas pelas vistas existem. Assim, como o objectivo do planeador é encontrar o melhor plano de execução, enquanto mais informação houver melhor será a sua decisão sobre o melhor plano.

Uma regra pode ser registada utilizando o comando `CREATE RULE`. Informação sobre todas as regras fica guardada no catálogo do sistema chamado `pg_rewrite`, sendo este usado posteriormente durante a fase de reescrita para poderem ser expandidas todas as regras aplicáveis à pergunta em causa. Nota-se que não existem regras por omissão, sendo todas as regras criadas explicitamente pelos utilizadores ou implicitamente durante a definição das vistas.

4.5 Planeamento/Optimização

Depois de ser reescrita, uma pergunta entra na fase de planeamento. Nesta fase, cada bloco estrutural da pergunta é tratado separadamente e um plano de execução é gerado para cada um. A ordem pela qual os blocos são considerados é de dentro para fora, i.e., primeiro, são considerados os blocos interiores, prosseguindo para os mais exteriores. O optimizador associa um custo estimado a cada plano de execução. Este custo inclui as

operações I/O de acesso sequencial e aleatório ao disco, e o tempo que o CPU leva para processar tuplos, predicados e índices, sendo as operações de I/O mais pesadas. Para permitir uma melhor estimativa do custo, é importante que existam boas estimativas do número dos tuplos que vão ser processados durante a execução do plano. Este número é obtido graças a informação estatística que é mantida pelo sistema para cada relação. Esta informação indica o número total de tuplos para cada relação, junto com alguma informação específica sobre cada atributo da relação, como a respectiva cardinalidade, uma lista de valores comuns e o número de ocorrências dos mesmo, e uma histograma *equi-depth* que divide cada valor dum atributo em grupos de quantidades iguais. Além disso, também é mantida uma correlação estatística entre as ordenações físicas e lógicas dos valores dum atributo, indicando assim o custo duma pesquisa com índice para devolver os tuplos que satisfaçam os predicados nesse atributo. A informação estatística não é actualizada constantemente de modo a não danificar o desempenho do sistema e, por isso, encontra-se desactualizada frequentemente. No entanto, o utilizador pode forçar a actualização de informação estatística a qualquer momento, recorrendo aos comandos `VACUUM` e `ANALYZE`. Alguns comandos DLL, como `CREATE INDEX`, também provocam o mesmo resultado.

Se for computacionalmente viável, o sistema opta por testar exaustivamente todos os possíveis planos de execução, o que resulta no melhor plano ser eventualmente encontrado e escolhido (secção 4.5.1). Caso contrário, o sistema recorre à utilização dum optimizador genético *(Genetic Query Optimizer)*, o qual permite encontrar um plano bom, não garantindo a optimalidade do mesmo (secção 4.5.2). A decisão se é computacionalmente viável testar exaustivamente todos os planos é tomada de acordo com o número de junções envolvidas na pergunta em causa. Para tomar esta decisão, é consultada a variável do sistema `geqo_threshold`. Se o número de junções for superior ao valor indicado pela variável, sendo este 12 por omissão, o sistema opta por utilizar o optimizador genético. Caso contrário, o sistema decide testar exaustivamente todos os planos possíveis.

`Nota:` Durante o processo de planeamento, o optimizador trabalha com representações simplificadas dos planos, chamadas *paths*, que omitem a informação que não vai ser necessária durante o planeamento. Entretanto, depois de determinar o *path* mais barato, o plano de execução completo é reconstruído e retornado como resultado.

O comando `EXPLAIN` fornece informação detalhada acerca dos planos gerados pelo optimizador para qualquer pergunta.

```
EXPLAIN SELECT * FROM customers;

QUERY PLAN
──────────────────────────────────────────────────────────────
 Seq Scan on customers  (cost=0.00..676.00 rows=20000 width=268)
(1 row)
```

4.5.1 Pesquisa exaustiva

No início duma pesquisa exaustiva, o optimizador começa por gerar planos para cada relação individual envolvida na pergunta. Os planos possíveis para uma relação individual incluem a pesquisa sequencial e a pesquisa com índices. Visto que é possível efectuar a pesquisa sequencial em qualquer situação, um plano com a mesma é criado sempre. Por outro lado, se existir um índice definido numa relação envolvida e este for aplicável na pergunta em causa, é criado outro plano de execução baseado na utilização desse índice. Se houver outros índices aplicáveis, é criado um plano para cada um deles. Este procedimento de planeamento foi usado pela primeira vez no *System R*—o primeiro sistema relacional desenvolvido pela IBM nos anos 1970.

Exemplo duma pergunta para a qual o optimizador opta por fazer uma pesquisa sequencial:

```
EXPLAIN ANALYZE SELECT * FROM customers;

QUERY PLAN
──────────────────────────────────────────────────────────────
 Seq Scan on customers  (cost=0.00..676.00 rows=20000 width=268)
     (actual time=0.010..12.654 rows=20000 loops=1)
 Total runtime: 20.575 ms
(2 rows)
```

Exemplo duma pergunta para a qual o optimizador utiliza um índice B-tree e Bitmap:

```
EXPLAIN ANALYZE SELECT * FROM customers
WHERE customerid=1000;

QUERY PLAN
------------------------------------------------------------------------
 Index Scan using customers_pkey on customers  (cost=0.00..8.27
    rows=1 width=268) (actual time=0.053..0.055 rows=1 loops=1)
   Index Cond: (customerid = 1000)
 Total runtime: 0.123 ms
(3 rows)
```

```
EXPLAIN ANALYZE SELECT customerid,username FROM customers
WHERE customerid<10000 AND username<'user100';

QUERY PLAN
------------------------------------------------------------------------
 Bitmap Heap Scan on customers  (cost=5.70..368.87 rows=95 width
    =13) (actual time=0.049..0.052 rows=2 loops=1)
   Recheck Cond: ((username)::text < 'user100'::text)
   Filter: (customerid < 10000)
   ->  Bitmap Index Scan on ix_cust_username  (cost=0.00..5.68
      rows=190 width=0) (actual time=0.020..0.020 rows=2 loops
      =1)
         Index Cond: ((username)::text < 'user100'::text)
 Total runtime: 0.103 ms
(6 rows)
```

Se uma pergunta envolver junções, são considerados vários planos de junções depois de encontrar os melhores planos para as relações individuais. Há três maneiras disponíveis de efectuar as junções: *nested loop join*, *merge join* e *hash join*. Caso uma pergunta envolver mais do que duas junções, o optimizador considera todas as possíveis sequências de junções para encontrar a mais barata. Neste caso, é preferida uma junção entre duas relações para as quais exista uma cláusula de junção correspondente no corpo de `WHERE`, i.e., se existir uma restrição do tipo A.x = B.y. Os pares de junções que não satisfaçam este critério são considerados apenas se existir uma relação que não esteja envolvida numa junção com alguma outra relação. As vezes, perante uma parte de pergunta aninhada dentro duma selecção ou dentro duma parte de junção, o optimizador pode preferir materializar essa parte

da pergunta.

Exemplo duma pergunta para a qual o optimizador utiliza o *nested loop join*:

```
EXPLAIN ANALYZE SELECT * FROM orders,inventory;

QUERY PLAN
------------------------------------------------------------------------
 Nested Loop  (cost=0.00..1500389.00 rows=120000000 width=42) (
    actual time=0.040..94457.719 rows=120000000 loops=1)
   ->  Seq Scan on orders  (cost=0.00..214.00 rows=12000 width
      =30) (actual time=0.010..9.441 rows=12000 loops=1)
   ->  Materialize  (cost=0.00..200.00 rows=10000 width=12) (
      actual time=0.000..2.476 rows=10000 loops=12000)
         ->  Seq Scan on inventory  (cost=0.00..150.00 rows
            =10000 width=12) (actual time=0.016..3.938 rows
            =10000 loops=1)
 Total runtime: 115577.425 ms
(5 rows)
```

Exemplo duma pergunta para a qual o optimizador utiliza o *index-nested loop join*:

```
EXPLAIN ANALYZE SELECT * FROM orders,orderlines
WHERE orders.totalamount=100 AND orders.orderid=orderlines.
   orderid;

QUERY PLAN
------------------------------------------------------------------------
 Nested Loop  (cost=0.00..259.41 rows=5 width=48) (actual time
    =9.011..9.011 rows=0 loops=1)
   ->  Seq Scan on orders  (cost=0.00..244.00 rows=1 width=30) (
      actual time=9.008..9.008 rows=0 loops=1)
         Filter: (totalamount = 100::numeric)
   ->  Index Scan using ix_orderlines_orderid on orderlines  (
      cost=0.00..15.34 rows=5 width=18) (never executed)
         Index Cond: (orderid = orders.orderid)
 Total runtime: 9.083 ms
(6 rows)
```

Exemplo duma pergunta para a qual o optimizador utiliza o *merge join*:

```
EXPLAIN ANALYZE SELECT C.customerid,sum(netamount) FROM
   customers C,orders O
WHERE C.customerid=O.customerid GROUP BY C.customerid;

QUERY PLAN
------------------------------------------------------------------------
 GroupAggregate  (cost=0.05..2015.49 rows=12000 width=10) (
    actual time=0.076..53.417 rows=8996 loops=1)
   ->  Merge Join  (cost=0.05..1835.49 rows=12000 width=10) (
      actual time=0.057..39.958 rows=12000 loops=1)
         Merge Cond: (c.customerid = o.customerid)
         ->  Index Scan using customers_pkey on customers c  (
             cost=0.00..963.25 rows=20000 width=4) (actual time
             =0.024..11.830 rows=20000 loops=1)
         ->  Index Scan using ix_order_custid on orders o  (cost
             =0.00..672.24 rows=12000 width=10) (actual time
             =0.022..11.102 rows=12000 loops=1)
 Total runtime: 55.477 ms
(6 rows)
```

Exemplo duma pergunta para a qual o optimizador utiliza o *hash join*:

```
EXPLAIN ANALYZE SELECT prod_id,title FROM products p
WHERE EXISTS (SELECT 1 FROM orderlines ol WHERE ol.prod_id=p.
   prod_id);

QUERY PLAN
------------------------------------------------------------------------
 Hash Join  (cost=1329.79..2281.28 rows=9796 width=19) (actual
    time=59.510..71.835 rows=9973 loops=1)
   Hash Cond: (p.prod_id = ol.prod_id)
   ->  Seq Scan on products p  (cost=0.00..198.00 rows=10000
      width=19) (actual time=0.008..2.923 rows=10000 loops=1)
   ->  Hash  (cost=1207.34..1207.34 rows=9796 width=4) (actual
      time=59.482..59.482 rows=9973 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 234kB
         ->  HashAggregate  (cost=1109.38..1207.34 rows=9796
             width=4) (actual time=51.315..55.383 rows=9973 loops
             =1)
               ->  Seq Scan on orderlines ol  (cost=0.00..958.50
                    rows=60350 width=4) (actual time
                   =0.008..18.635 rows=60350 loops=1)
 Total runtime: 73.765 ms
(8 rows)
```

O plano final (*plan tree*) contém toda a informação sobre como este deve ser executado, incluindo o tipo de pesquisa (sequencial ou com um índice) para as relações individuais, o algoritmo de junção aplicado às junções, e todos os passos auxiliares que possam ser necessários, tais como ordenação ou cálculo de funções de agregação. Também é indicado em que alturas da execução devem ser feitas as operações de selecção e projecção.

Exemplo duma pergunta com ordenação para a qual o optimizador utiliza o *external sort-merge*:

```
EXPLAIN ANALYZE SELECT customerid FROM customers ORDER BY zip;

QUERY PLAN
------------------------------------------------------------------------
 Sort  (cost=2104.77..2154.77 rows=20000 width=8) (actual time
    =40.183..47.777 rows=20000 loops=1)
   Sort Key: zip
   Sort Method: external sort  Disk: 352kB
   ->  Seq Scan on customers  (cost=0.00..676.00 rows=20000
      width=8) (actual time=0.012..13.932 rows=20000 loops=1)
 Total runtime: 52.703 ms
(5 rows)
```

Exemplo duma pergunta com ordenação para a qual o optimizador utiliza o *quicksort*, visto que o valor de `work_mem` foi aumentado de modo a permitir com que a pergunta fosse executa na memória:

```
SET work_mem='2MB';
SET EXPLAIN ANALYZE SELECT customerid FROM customers ORDER BY
   zip;

QUERY PLAN
------------------------------------------------------------------------
 Sort  (cost=2104.77..2154.77 rows=20000 width=8) (actual time
    =29.110..34.192 rows=20000 loops=1)
   Sort Key: zip
   Sort Method: quicksort  Memory: 1294kB
   ->  Seq Scan on customers  (cost=0.00..676.00 rows=20000
      width=8) (actual time=0.012..13.675 rows=20000 loops=1)
 Total runtime: 39.101 ms
(5 rows)
```

4.5.2 Algoritmo genético

Se o número de tabelas num bloco de pergunta for muito grande, o algoritmo de pesquisa exaustiva torna-se muito caro. Perante este problema, ao contrário de muitos SGBDs comerciais, como o Oracle, que utilizam técnicas baseadas em regras e algortimos *greedy*, o PostgreSQL utiliza um algoritmo genético que foi originalmente desenvolvido para resolver os problemas do caixeiro viajante *(traveling-salesman problems)*. Há afirmações (não provadas) da utilização sucedida deste algoritmo perante perguntas com cerca de 45 tabelas envolvidas. Na execução deste algoritmo, todos os planos possíveis são codificados em cadeias de inteiros. Cada cadeia representa uma ordem específica de efectuar as junções. Por exemplo, 1-2-3-4 significa juntar primeiro as relações 1 e 2, depois juntar o resultado com 3, e finalmente com 4. Nota-se que os planos para as relações individuais continuam a ser gerados recorrendo ao algoritmo *System R*.

Na fase inicial do algoritmo genético, são geradas aleatoriamente algumas sequências de junções. Para cada sequência gerada, é invocado o algoritmo *System R* para estimar o custo de executar a pergunta usando essa sequência. As sequências de baixo custo são consideradas 'mais fortes' do que as de custo mais alto. O algoritmo genético descarta os candidatos 'mais fracos' e gera novas sequências combinando genes dos candidatos 'mais fortes', ou seja, utiliza porções aleatórias das sequências de baixo custo para gerar outras sequências. Este processo é repetido até ter sido considerado um número predefinido de sequências, e a melhor sequência encontrada até esse momento é usada para gerar o plano final. Nota-se que, muitas vezes, o plano gerado não é optimal, o que apresenta um desafio para a equipa de desenvolvimento em encontrar compromisso entre o tempo de computação e a optimalidade do plano final.

É de notar que o PostgreSQL não implementa o sistema de *hints*, que é usado por algums SGBDs, como o Oracle, de modo a permitir optimização adiconal dos planos.

4.6 Execução

O executor recebe o plano criado pelo optimizador e processa-o recursivamente, extraindo no final todos os tuplos que devem aparecer no resultado final. Quando um nó da árvore é seleccionado, este devolve um tuplo ou sinaliza que não tem mais tuplos para devolver, aplicando assim *demand-pull (demand-driven) pipeline*. O sistema aplica este mecanismo mesmo perante uma pergunta complexa definida por uma árvore de alta profundidade. Cada nó deve aplicar todas as expressões de selecção ou projecção que lhe tenham sido atribuídas na fase anterior pelo optimizador.

O executor é utilizado para avaliar os quatro tipos de perguntas SQL básicas, nomeadamente `SELECT`, `INSERT`, `UPDATE` e `DELETE`. Em caso de `SELECT`, o executor apenas precisa de devolver cada tuplo retornado pela árvore como resultado. Em caso de `INSERT`, cada tuplo retornado também precisa de ser inserido na tabela indicada. Isto é feito recorrendo a um nó especial chamado *ModifyTable*. Em caso de `UPDATE`, cada tuplo do plano contém os valores actualizados dos atributos solicitados junto com um *tuple-ID* do tuplo original. Estes dados são usados pelo nó *ModifyTable* para criar os tuplos actualizados e marcar os antigos como apagados. Para `DELETE`, a única coluna que é devolvida pelo plano é o tuple ID, que é usado pelo *ModifyTable* para marcar todos os tuplos correspondentes aos IDs devolvidos como apagados. Os operadores importantes do executor incluem a pesquisa sequencial e com índices; os métodos de junção *sorted merge join*, emphnested-loop join (incluindo a sua variante *index-nested loop*) e *hybrid hash join*; a ordenação *external sort-merge*; e a agregação baseada em ordenação ou hashing.

Ao contrário dalguns sistemas comerciais, triggers e restrições de integridade, como asserções, não são implementadas na fase de reescrita, sendo da responsabilidade do executor. Quando triggers ou restrições são introduzidos pelo utilizador, os detalhes sobre os mesmos são associados com a informação guardada nos catálogos sobre todas as relações e índices relevantes. Assim, durante qualquer modificação de tuplos, o executor identifica, impõe e invoca todos os triggers e outras restrições aplicáveis às relações envolvidas.

5 Gestão de Transacções e Controlo de Concorrência

No que diz respeito a gestão de transacções, o PostgreSQL utiliza duas técnicas distintas, nomeadamente *snapshot isolation* e *two-phase locking*. Qual delas é usada depende do tipo do comando que estiver a ser executado. Para os comandos DML (qualquer comando que actualiza ou lê dados duma tabela, por exemplo `SELECT`, `INSERT` e `UPDATE`), é utilizada a técnica de *snapshot isolation*, designada por *multiversion concurrency control (MVCC)* pelo PostgreSQL. Por outro lado, o controlo de concorrência para os comandos DLL (os comandos que afectam tabelas inteiras, podendo apagar uma tabela ou mudar o respectivo esquema), é baseado em *two-phase locking*. É de notar que, ao contrário do Oracle, o PostgreSQL não implementa transacções anónimas, estando de momento na lista de extensões possíveis para as futuras versões do mesmo.

Na secção seguinte descrevem-se os níveis de isolamento fornecidos pelo PostgreSQL.

5.1 Níveis de Isolamento

O standard SQL define quatro níveis de isolamento: *read uncommited*, *read commited*, *repeatable read* e *serializable*, sendo os três primeiros níveis mais relaxados, permitindo mais concorrência para as aplicações que não necessitam da serialização rígida. Os diferentes níveis de isolamento são definidos em termos de três fenómenos violadores de serialização:

1. **dirty reads**

   Uma transacção lê valores escritos por outra transacção que ainda não fez commit;

2. **non-repeatable reads**

   Uma transacção lê diferentes valores acedendo ao mesmo objecto pela segunda vez sem o ter modificado, i.e, houve uma outra transacção que modificou o mesmo objecto e fez commit;

3. **phantom reads**

   Uma transacção executa pela segunda vez uma pergunta, retornando um conjunto de tuplos que satisfazem o predicado, e observa um conjunto diferente do da primeira execução, indicando que houve uma outra transacção que introduziu modificações e fez commit.

Os níveis de isolamento definidos por SQL encontram-se na tabela seguinte:

Nível de Isolamento	Dirty Read	Non-repeatable Read	Phantom Read
Read Uncommited	Possível	Possível	Possível
Read Commited	Impossível	Possível	Possível
Repeatable Read	Impossível	Impossível	Possível
Serializable	Impossível	Impossível	Impossível

Em PostgreSQL, o utilizador pode solicitar qualquer um destes níveis de isolamento, recorrendo ao comando `SET TRANSACTION`. No entanto, internamente existem apenas três níveis distintos, sendo estes *read commited*, *repeatable read* e *serializable*. O nível *read uncommited* equivale a *repeatable read* na implementação. O PostgreSQL permite apenas estes três níveis de modo a permitir uma implementação mais eficiente de MVCC.

5.2 Locking explícito

O sistema fornece vários modos de locks para controlar acesso concorrente aos dados. Estes modos são usados em situações onde MVCC não apresenta o comportamento desejado. A maioria dos comandos PostgreSQL adquirem locks automaticamente em modos apropriados para garantir a integridade das tabelas referenciadas.

A figura 1 apresenta todos os modos de locks disponíveis, incluindo os contextos nos quais os mesmos são obtidos automaticamente e os modos em conflito. O utilizador pode requerer explicitamente um lock recorrendo ao comando `LOCK`. Todos estes modos são aplicados a tabelas inteiras. Uma vez adquirido, o lock é mantido até o fim da respectiva transacção. No entanto, se um lock for adquirido depois de ter estabelecido um *savepoint*, o lock é libertado imediatamente caso haja um `ROLLBACK` para esse *savepoint*.

A utilização dos locks explícitos geralmente aumenta a probabilidade de ocorrência de *deadlocks*. O sistema detecta todas as situações de *deadlocks* e resolve os mesmos efectuando a operação `ABORT` numa das transacções envolvidas, permitindo assim o prosseguimento das outras transacções. Na detecção de *deadlocks*, é utilizado um grafo direccionado, designado *wait-for graph*, cujos nós representam transacções e cujos arcos representam as relações em termos de locks existentes entre essas transacções. Por exemplo, existe um arco de A para B se A estiver à espera dum lock que foi adquirido por B anteriormente em modo de lock incompatível. Segundo este esquema, existe um deadlock apenas se existir um ciclo no grafo. O sistema percorre este grafo periodicamente de modo a detectar todas as ocorrências de *deadlocks*.

Lock name	*Conflicts with*	*Acquired by*
ACCESS SHARE	ACCESS EXCLUSIVE	**select** query
ROW SHARE	EXCLUSIVE ACCESS EXCLUSIVE	**select for update** query **select for share** query
ROW EXCLUSIVE	SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	**update** **delete** **insert** queries
SHARE UPDATE EXCLUSIVE	SHARE UPDATE EXCLUSIVE SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	**vacuum** **analyze** **create index concurrently**
SHARE	ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	**create index**
SHARE ROW EXCLUSIVE	ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	---
EXCLUSIVE	All except ACCESS SHARE	---
ACCESS EXCLUSIVE	All modes	**drop table** **alter table** **vaccum full**

Figura 1: Modos de locks

5.2.1 Row-level locks

Para além de locks que afectam tabelas inteiras, existe uma outra classe de locks que afectam tuplos. Estes podem ser obtidos em dois modos, nomeadamente *exclusive* ou *shared*. Os locks em modo *exclusive* são obtidos automaticamente quando um tuplo é actualizado ou apagado, e é mantido até a respectiva transacção fizer `COMMIT` ou `ROLLBACK`. O utilizador pode recorrer ao comando `SELECT FOR UPDATE` para obter um *exclusive row-level lock* nos tuplos retornados pelo comando. Para obter um *shared row-level block*, usase o comando `SELECT FOR SHARE`. Apesar de não existir o limite do número dos tuplos que podem ser bloqueados, solicitação dum lock pode provocar uma escrita no disco. Por exemplo, o comando `SELECT FOR UPDATE` modifica os tuplos seleccionados para os marcar como bloqueados, o que resulta num escrita no disco.

5.2.2 Advisory Locks

O sistema também permite criar *advisory locks*, que são definidos no contexto da própria aplicação e não são forçados pelo sistema. Estes podem ser adquiridos a *nível de sessão* ou *nível de transacção*, e livrados assim que uma sessão ou transacção terminar. Assim sendo, é possível adquirir um lock a iniciar uma sessão e larga-lo apenas quando esta terminar, garantindo assim que não haja concorrência durante a execução da sessão. É de notar que estes locks permanecem mantidos mesmo se a transacção durante a qual tenham sido adquiridos tiver acabar com um `ROLLBACK`. No entanto, os locks adquiridos a *nível de transacção* apresentam o comportamento normal neste caso, sendo libertados automaticamente no final da transacção e não possibilitando a libertação explícita dos mesmos.

5.2.3 Locking e índices

Todos os tipos de índices permitem acesso concorrente recorrendo a utilização de *page-level locks (block-level locks)*, o que permite acesso paralelo a uma índice para várias transacções, caso estas não solicitarem locks em conflito para o mesmo bloco. Estes locks são normalmente mantidos durante

algum período de tempo de modo a evitar deadlocks, exceptuando o caso dos índices hash, para os quais um bloqueio contínuo pode provocar um deadlock.

5.2.4 Recuperação PITR (Point-in-time Recovery)

O mecanismo de recuperação *PITR* do PostgreSQL é baseado em *WAL (Write-Ahead Logging)*, e é semelhante às técnicas de recuperação como *ARIES*, sendo simplificado nalguns aspectos devido ao protocolo MVCC. Segundo RITP, O sistema mantém um log, chamado *write ahead log (WAL)*, no directório `pg_xlog/`, e guarda no mesmo todas as alterações feitas aos ficheiros da base de dados. Se ocorrer uma falha no sistema, a base de dados é recuperada executando todas as entradas introduzidas no log após o último *checkpoint*.

6 Suporte para bases de dados distribuídas

O PostgreSQL contém um mecanismo de replicação de dados baseado na interacção de um servidor primário (*master*) com um ou vários servidores secundários (*slaves*), preparados para assumir o papel de servidor primário em caso de falha deste último.

6.1 Log Shipping/Warm Standby

O mecanismo de replicação é implementado através da técnica de *Log Shipping/Warm Standy*. Esta técnica baseia-se no envio periódico do WAL (*Write-Ahead Logs*), por segmentos (ficheiros) de transacção com tamanho de 16MB, do servidor primário para os servidores secundários. Este mecanismo é assíncrono o que representa uma grande desvantagem já que se podem perder transacções em caso de falha do servidor primário. Outra desvantagem está no facto de todos os servidores envolvidos na interacção terem de ter a mesma arquitectura. Ainda assim, é um mecanismo fácil de implementar e que resulta no aliviar da carga de trabalho do servidor principal, já que os servidores secundários (neste caso, do tipo *Hot Standby*) podem efectuar operações de leitura.

6.2 Streaming Replication

Este tipo de replicação permite que os servidores secundários estejam o mais actualizados possível. Desta feita, os segmentos do WAL são enviados à medida que são gerados.

6.3 Cascading Replication

Este tipo de replicação permite que servidores secundários aceitem registos WAL de outros servidores secundários. Isto permite aliviar a carga do master server já que algumas ligações a este se tornam desnecessárias. No caso de um servidor secundário "subir" à posição de servidor primário, os

servidores que faziam *stream* deste mantém a ligação se a instrução `recovery_target_timeline` estiver avaliada em 'latest'.

6.4 Synchronous Replication

Com replicação síncrona, o commit de uma transacção espera até que seja recebida uma notificação de transacção escrita no log por parte de um servidor secundário. Desta feita, garante-se que uma actualização do log do servidor primário se reflectiu em um ou mais servidores. O ponto negativo está no tempo que uma actualização pode demorar.

6.5 Comparação com Oracle 11g

O Oracle suporta replicação de dados de um servidor *master* sob a forma de *materialized views* sendo que estas poderão conter parte da informação da *master data*. Esta é uma das diferenças para o PostgreSQL já que este não permite fragmentação de dados. Outra das diferenças está na possibilidade de o Oracle poder ter mais que um servidor primário.

7 Outras características

Das características suportadas pelo PostgreSQL serão abordadas as de `suporte para XML` (secção 7.1) e `segurança` (secção 7.2).

7.1 Suporte para XML

O PostgreSQL dispõe de muitos mecanismos de manipulação/interpretação de XML.

É possível a utilização de XML como tipo de dados de uma tabela:

```
CREATE TABLE "Bar"
(
"ID" integer NOT NULL,
"Bar" xml,
CONSTRAINT "Bar_pkey" PRIMARY KEY ("ID")
)
```

O Sistema dispõe de muitas funções para gerar conteúdo XML (ou a partir de objectos XML) dos quais se destacam:

- `XMLPARSE`: Criação de objectos XML utilizando o comando:

```
XMLPARSE ( \{ DOCUMENT | CONTENT \} value )

Exemplo:
XMLPARSE (DOCUMENT '<?xmlversion="1.0"?><Bar><city>lisboa</
   city></BAR>')
```

- `XMLSERIALIZE`: Geração de texto a partir de um objecto XML utilizando o comando:

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

Exemplo:
XMLSERIALIZE (DOCUMENT ('<Bar><city>lisboa</city></Bar>') AS
    text)
```

- `query_to_xml`: Criação de conteúdo XML a partir do resultado de uma query utilizando o comando:

```
query_to_xml(query text, nulls boolean, tableforest boolean,
    targetns text)

Exemplo:
select *
from query_to_xml('SELECT * FROM public.actor',false,false
   ,'');
```

A limitação no uso deste tipo de dados está no facto de ser impossível comparar os mesmos. Isto representa uma limitação nas perguntas, operando sobre este tipo de dados, que se podem fazer. A criação de índices sobre colunas com este tipo de dados é impossível.

7.2 Segurança

O sistema dispõe dos seguintes mecanismos de autenticação:

- Trust
- Password
- GSSAPI
- SSPI
- Kerberos
- Ident
- Peer
- LDAP
- RADIUS
- Certificate

- PAM

O ficheiro `pg hba.conf` mantém informação relativa aos utilizadores e suas permissões. Existe uma linha para cada utilizador; por exemplo:

```
host replication postgres 127.0.0.1 md5
```

Neste caso o utilizador *postgres* do tipo *host* com morada *127.0.0.1* tem acesso à base dados *replication* utilizando *md5* para se autenticar. Note-se que por ser md5, a password é cifrada antes de ser enviada. Por defeito, o sistema atribui o método de md5 quando regista utilizadores.

8 Bibliografia

[1] Documentação oficial do PostgreSQL http://www.postgresql.org/docs/9.3

[2] A. Silberschatz, Henry F. Korth, S.Sudarshan, *Database System Concepts, 6th*, McGraw-Hill, New York, NY, 2011.

[3] Wiki do PosgreSQL http://wiki.postgresql.org/wiki/Main_Page

[4] PostgreSQL http://en.wikipedia.org/wiki/PostgreSQL

[5] G. Smith, *PostgreSQL 9 High Performance*, Packt Publishing, Birmingham, UK, 2010.

[6] The PGCon 2010 Conference http://www.pgcon.org/2010

[7] Oracle Database Online Documentation 11g Release 1 (11.1) http://docs.oracle.com/cd/B28359_01/index.htm

[8] Replication http://www.themagicnumber.es/replication-in-postgresql-i?lang=en

[9] XML in PostgreSQL http://www.scribd.com/doc/2402063/SQL-XML-For-Postgres-Developers

[10] Read XML from file http://dba.stackexchange.com/questions/8172/sql-to-read-xml-from-file-into-postgresql-database

9 Anexos

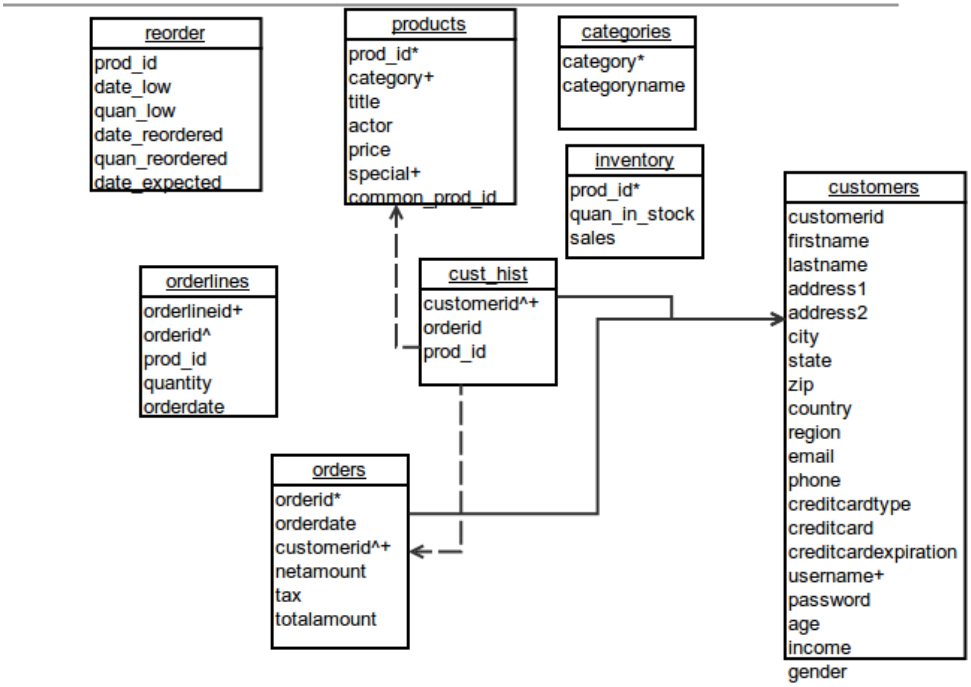

Figura 2: Esquema da base de dados usada nos exemplos

O número de tuplos contidos em cada tabela:

Tabela	Tuplos
`customers`	20000
`inventory`	10000
`products`	10000
`orders`	12000
`categories`	16
`cust_hist`	60350
`orderlines`	60350
`reorder`	1510

table	size
public.customers	3808 kB
public.orderlines	2840 kB
public.cust_hist	2368 kB
public.products	808 kB
public.orders	800 kB
public.inventory	400 kB

index	size
public.ix_orderlines_orderid	1336 kB
public.ix_cust_hist_customerid	1080 kB
public.ix_cust_username	544 kB
public.customers_pkey	368 kB
public.ix_order_custid	232 kB
public.orders_pkey	232 kB
public.ix_prod_special	192 kB
public.ix_prod_category	192 kB
public.products_pkey	192 kB
public.inventory_pkey	192 kB

Figura 3: Tamanhos das tabelas e índices principais