4
Implementação e Resultados Experimentais

Neste capítulo, abordamos a implementação da arquitetura com um único agente apresentada no Capítulo 3. Criamos um protótipo da arquitetura utilizando o SGBD de código fonte aberto PostgreSQL e realizamos uma implementação em C++ do *framework* apresentado em [35]. Focamos a nossa atenção nas heurísticas necessárias para a criação de novos índices, uma vez que isto já permite que utilizemos a arquitetura para gerenciar automaticamente o projeto de índices de uma base de dados que inicialmente não apresente nenhum índice criado. A implementação permite, entretanto, que a lógica relativa ao trabalho de remoção de índices seja posteriormente adicionada.

Iniciamos nossa apresentação descrevendo, na seção 4.1, alguns conceitos da arquitetura do PostgreSQL. Mostramos, na seção 4.2, as cargas de trabalho e bases de dados que foram utilizadas para nossos testes de desempenho e de qualidade. Apresentamos o ambiente e as ferramentas utilizadas para o desenvolvimento e os testes na seção 4.3. Discutimos, na seção 4.4, a implementação de índices hipotéticos no SGBD PostgreSQL. A implementação e os resultados obtidos com a arquitetura com um único agente são debatidos na seção 4.5. Por fim, apresentamos alguns comentários sobre o trabalho de implementação realizado na seção 4.6.

4.1
Arquitetura do PostgreSQL

Nesta seção, abordamos os principais conceitos da arquitetura do PostgreSQL que serão necessários para a compreensão da nossa implementação. As informações contidas nesta dissertação foram obtidas a partir do manual do produto [46], da documentação oferecida pelos seus desenvolvedores [47] e da análise do código do SGBD, aqui efetivada.

Iremos abordar, na próxima subseção, o modelo utilizado para comunicação pelo sistema e as suas implicações sobre os processos criados no

servidor. Em seguida, mostraremos os principais módulos envolvidos com o processamento de consultas.

Modelo de Comunicação

O SGBD PostgreSQL segue um modelo de comunicação cliente-servidor. Cada cliente se conecta a exatamente um processo servidor. Como a quantidade de conexões que serão realizadas não é conhecida no momento em que o sistema é inicializado, um processo mestre, denominado `postmaster`, é criado para gerenciar a chegada de novos clientes. Este processo também é responsável por iniciar a área compartilhada de memória que será utilizada por todos os processos do SGBD no servidor e por gerenciar tarefas especiais, como inicializações, paradas e *checkpoints*.

Cada conexão ao sistema é tratada por um processo no servidor, denominado `postgres`. Estes processos `postgres` recebem os comandos enviados pelos clientes e se comunicam uns com os outros utilizando semáforos e memória compartilhada. Assim, conseguem garantir a integridade dos dados mesmo trabalhando simultaneamente em comandos enviados por clientes distintos.

A Figura 4.1 mostra os processos do lado servidor do sistema e os módulos mais importantes envolvidos no processamento de consultas.

Processamento de Consultas

Quando um processo `postgres` recebe um comando SQL, ele invoca o seu analisador sintático (*parser*, veja a Figura 4.1). Este módulo é responsável por garantir que a sintaxe do comando está correta e por gerar uma árvore que o representa. Comandos são de dois tipos principais: utilitários e complexos.

Comandos utilitários são executados imediatamente, sem passar por transformações adicionais. Exemplos deste tipo de comando são os comandos da linguagem de definição de dados, como `create index`, `create table`, entre outros.

Já comandos complexos exigem outras transformações antes da execução. São aplicados os passos de reescrita de consultas e de otimização. Comandos complexos incluem os comandos `select`, `insert`, `delete`, `update`, `declare cursor` e `execute`.

A reescrita de consultas transforma a árvore original proveniente do analisador sintático em uma árvore semanticamente validada, com tipos

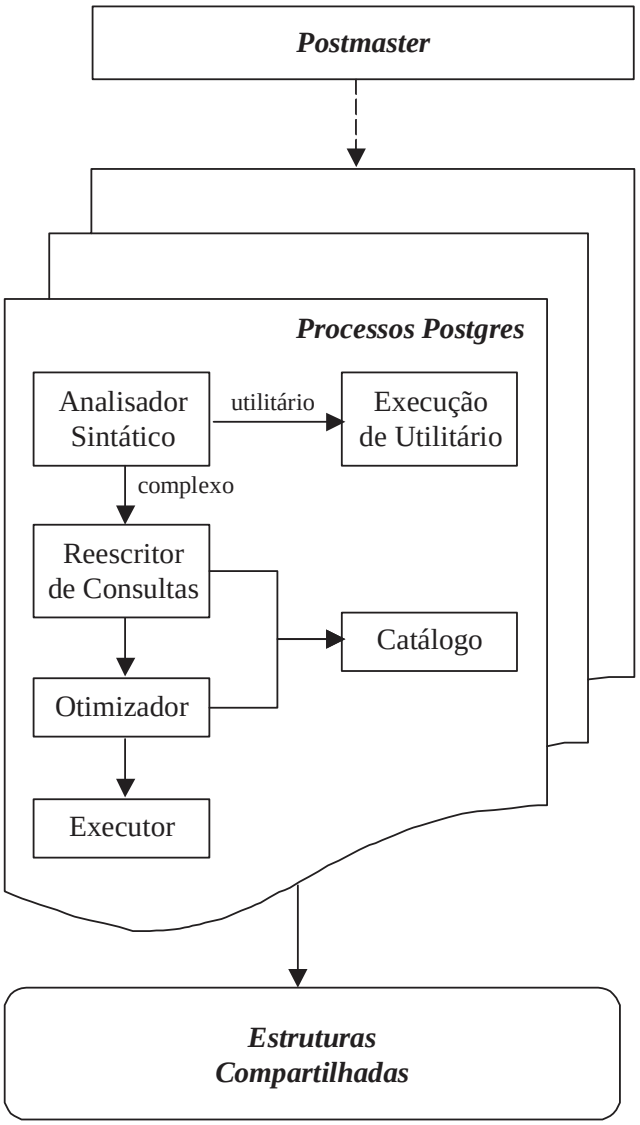

Figura 4.1: Processos e módulos do PostgreSQL

checados e regras de reescrita aplicadas. Estas regras de reescrita e restrições de tipos são armazenadas no catálogo do sistema. Um exemplo comum de reescrita é a expansão de visões na árvore [28].

A árvore validada é enviada para o otimizador, que busca pelas melhores combinações de métodos de acesso e operadores físicos que podem ser utilizados para responder a consulta. Para consultas com um número aceitável de junções [1], o sistema utiliza um algoritmo de busca exaustiva. Caso o número de junções seja maior do que o limite aceitável, é utilizada uma heurística de busca baseada em algoritmos genéticos. O otimizador baseia suas decisões em custos estimados a partir de estatísticas coletadas sobre as tabelas da base de dados. Estas estatísticas são armazenadas no

catálogo e consultadas pelo otimizador.

Após a etapa de otimização é obtido um plano de execução para a consulta, juntamente com estimativas de custos para as operações. Este plano é dado como entrada para o módulo executor do PostgreSQL, que se encarrega de interpretar as operações estipuladas e acessar módulos de mais baixo nível no SGBD, como as interfaces dos métodos de acesso e do gerente de armazenamento. As tuplas obtidas pelo executor são enviadas de volta ao cliente que originou o comando.

4.2 Cargas de Trabalho e Bases de Dados de Teste

Descrevemos, nesta seção, as bases de dados e cargas de trabalho utilizadas para testes da implementação realizada. Fizemos testes separados para a implementação de índices hipotéticos e para a implementação da arquitetura com um único agente para auto-sintonia de índices apresentada na seção 3.2. Apresentamos os objetivos e motivações de cada teste realizado nas próximas subseções, bem como as bases de dados e aplicações utilizadas.

Esquema e Dados para Testes de Índices Hipotéticos

Para os testes de índices hipotéticos, estamos interessados em utilizar uma base de dados simples, com poucas tabelas, mas com possibilidade de crescimento do número de registros armazenados. A motivação é a de permitir que comparemos o tempo de criação de índices reais e hipotéticos e também que verifiquemos se as estimativas geradas pelo otimizador são consistentes entre os dois tipos de índice.

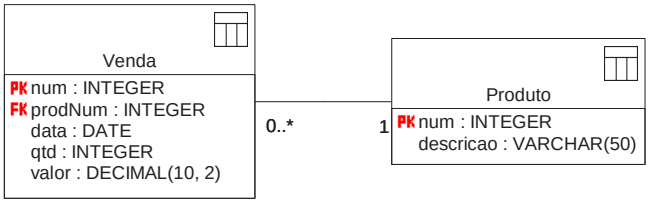

Figura 4.2: Esquema do banco de dados simples de vendas

Criamos programas para gerar dados para o esquema de uma base de dados simples de vendas mostrado na Figura 4.2. O esquema é composto de

[1]Este número é configurado como uma variável do sistema chamada *geqo_threshold*, que possui valor padrão igual a 11.

duas tabelas: `Produto` e `Venda`. A tabela `Produto` registra os produtos que podem ser vendidos. Ela é populada com um conjunto fixo de produtos.

A tabela `Venda` registra as vendas realizadas dos produtos disponíveis desde uma data no passado. O seu conteúdo é criado por um programa gerador. O programa gera vendas para uma quantidade de dias passada como entrada. Cada dia possui 1000 vendas. Para cada venda, escolhemos aleatoriamente um produto do intervalo de produtos disponível, uma quantidade de um determinado intervalo (arbitrariamente estabelecido como sendo entre 1000 e 200000) e um valor proporcional à quantidade escolhida. O valor é obtido pela multiplicação da quantidade por um preço unitário aleatoriamente sorteado de outro intervalo pré-definido (entre 6 e 11).

Para os testes de índices hipotéticos, variamos o tamanho da base de vendas e fizemos medições de desempenho e de qualidade. Os resultados destes testes podem ser encontrados na seção 4.4.

Carga de Trabalho para Testes da Arquitetura com um Único Agente

Para os testes da arquitetura com um único agente é necessário o uso de uma base de dados mais complexa do que a mostrada na subseção anterior e de uma carga de trabalho associada a esta nova base de dados. É nosso desejo avaliar como o agente pode conseguir adaptar dinamicamente o projeto de índices de uma base de dados à medida que o sistema processa uma carga de trabalho submetida por múltiplos usuários. Como a arquitetura com um único agente tende a ser mais aplicável a cargas de trabalho relativamente estáveis, pesquisamos a existência de cargas transacionais (OLTP, *On-Line Transaction Processing*) que pudessem ser utilizadas com o SGBD PostgreSQL. Nosso objetivo com essa busca foi o de encontrar uma carga de trabalho que pudesse simular um uso típico que uma aplicação de bancos de dados faz do SGBD.

A organização *Open Source Development Labs* (OSDL) [43] mantém um conjunto de testes de desempenho para avaliar o comportamento de soluções de *software* livre, em especial o sistema operacional Linux. Existem testes voltados especificamente para sistemas de bancos de dados e inspirados nos *benchmarks* do *Transaction Processing Performance Council* (TPC) [60].

Utilizamos para os nossos testes o *toolkit Database Test 2* (DBT-2) provido pela OSDL. Este *toolkit* simula uma carga de trabalho que representa um fornecedor de varejo operando uma quantidade determinada de armazéns e seus distritos de vendas associados. A carga de trabalho

envolve transações de um conjunto de operadores de terminais em um ambiente de entrada de pedidos. A origem da base de dados utilizada e das transações implementadas é o *benchmark* TPC-C [61].

Apesar de utilizar o mesmo esquema, forma de geração de dados e transações do *benchmark* TPC-C, os resultados obtidos com o DBT-2 não devem ser comparados com resultados oficiais publicados pelo TPC. Os resultados oficiais do TPC aderem a um conjunto de práticas de auditoria que requerem, entre outras informações, a publicação de dados sobre preços e configuração de todos os produtos utilizados para obtenção do resultado. O objetivo do DBT-2 é oferecer para seus usuários a capacidade de gerar uma carga de trabalho OLTP comparável às encontradas em aplicações reais.

Na Figura 4.3, mostramos a arquitetura do DBT-2. Existem três grandes componentes no sistema que se comunicam através de conexões de rede (TCP/IP): os terminais, os concentradores de terminais e o SGBD. Vários terminais podem se conectar a alguns concentradores de terminais que por sua vez se conectam a um único SGBD.

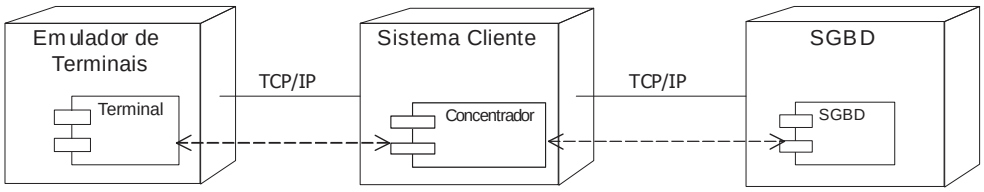

Figura 4.3: Arquitetura do *toolkit* DBT-2

No SGBD são criadas nove tabelas com procedimentos armazenados que dão suporte a cinco tipos de transações. A base de dados pode ser dimensionada para um número arbitrário de armazéns de itens de varejo, possibilitando a simulação de operações de organizações de diferentes tamanhos.

A figura 4.4 mostra o esquema utilizado pelo *benchmark* TPC-C. Cada armazém cobre 10 distritos, cada distrito serve 3000 compradores, e cada armazém mantém estoque completo para 100000 itens. Assim, se criarmos uma base com 3 armazéns, a maior tabela, a tabela de estoque, terá 300000 tuplas. Além disto, os compradores fazem pedidos de diversos dos itens mantidos em estoque. Cada item associado a um pedido gera uma nova linha de pedido. Até que os pedidos feitos pelos compradores sejam processados eles são classificados como novos pedidos. A organização mantém, ainda, informações de histórico de compras dos seus compradores.

As transações que podem ser executadas no SGBD são:

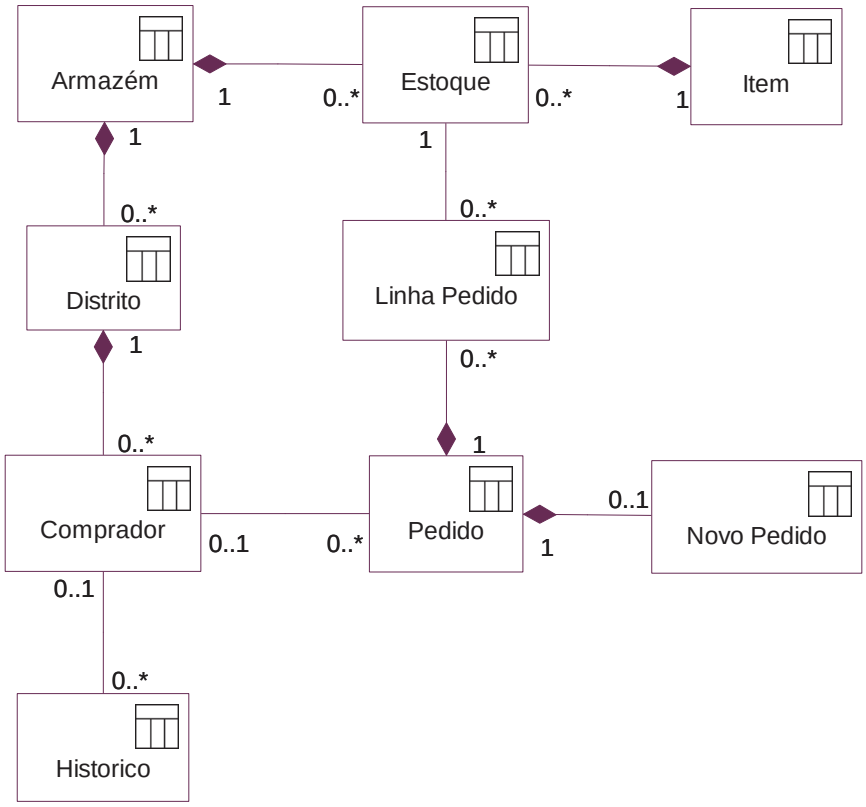

Figura 4.4: Esquema do *benchmark* TPC-C

- Novo Pedido (*New-Order*): transação de peso médio que simula atividade *on-line* tipicamente encontrada em ambientes de produção. A vazão reportada como resultado de um teste executado com o *toolkit* é o número deste tipo de transação executado por minuto.
- Pagamento (*Payment*): transação leve que atualiza o saldo de um comprador e reflete o pagamento feito nas informações estatísticas mantidas para o distrito e o armazém correspondentes.
- Estado de Pedido (*Order-Status*): transação de peso médio que verifica o estado do pedido mais recente de um comprador.
- Entrega (*Delivery*): transação que processa a entrega de até dez pedidos.
- Nível de Estoque (*Stock-Level*): transação pesada apenas de leitura que determina o número de itens recentemente vendidos que têm um nível de estoque abaixo de um dado limite.

Cada transação possui uma freqüência de execução. Estas freqüências são mostradas na Tabela 4.1 e representam qual é a parcela esperada para cada transação no número total de transações executadas pelo *toolkit*.

Nome da Transação	Freqüência de Execução
Novo Pedido	45%
Pagamento	43%
Estado de Pedido	4%
Entrega	4%
Nível de Estoque	4%

Tabela 4.1: Freqüências das transações executadas pelo *toolkit*

Cada terminal simula as atividades de uma pessoa que acessa o sistema e executa uma das transações acima. O terminal é implementado como um processo leve(*thread*[2]) num programa emulador de terminais. Para cada armazém da base de dados são criados 10 terminais.

Os terminais se conectam a concentradores de terminais para acessar o banco de dados em um modelo em três camadas. Os concentradores mapeiam diversos terminais para uma única conexão ao SGBD, permitindo compartilhamento de conexões. Cada terminal conectado em um concentrador causa a criação de uma *thread* de tratamento do terminal no programa concentrador.

Uma especificação mais detalhada do esquema e das transações empregadas pode ser encontrada na descrição do *benchmark* TPC-C [61]. Os resultados dos testes por nós conduzidos com esta carga de trabalho podem ser vistos na seção 4.5.2.

4.3 Ambiente de Implementação

Utilizamos em nossa implementação o SGBD de código fonte aberto PostgreSQL versão 7.4 beta 3 [46]. Foram feitas alterações no código deste SGBD para introduzir a noção de índices hipotéticos e para acoplar o *Agente de Benefícios*. O código adicionado ao SGBD foi escrito em C, linguagem em que o mesmo está implementado. Já o código do agente foi escrito em C++, permitindo a utilização de orientação a objetos. Fizemos uso do *framework* CppUnit 1.8.0 [17] para a construção de testes unitários para as classes do agente e do software Doxygen 1.2.14 [20] para geração de documentação. Testes de desempenho foram em grande parte automatizados através do uso de *shell scripts* para o ambiente Bash.

[2]Nesta dissertação, utilizaremos o termo *thread* pois este está mais próximo do contexto da implementação realizada.

A modelagem do sistema foi feita inicialmente utilizando a ferramenta Together [63] e depois migrada para Rational Rose 2002 [51]. Na codificação empregamos o editor GNU Emacs 21.2.1 [30]. Utilizamos, ainda, o depurador gdb [29] e a biblioteca Electric Fence [21] para a detecção de erros de alocação dinâmica de memória.

Os testes foram conduzidos em dois ambientes distintos. Para os testes de índices hipotéticos utilizamos uma máquina Pentium III 1.1 GHz com 256 MB de memória RAM e com um disco padrão IDE. O sistema operacional instalado é o Red Hat Linux 8, kernel 2.4.18-14, com ambiente gráfico GNOME [48]. A compilação do sistema foi feita com os programas gcc e g++, versão 3.2 [29].

Já para os testes da arquitetura com um único agente, utilizamos uma máquina mais poderosa para conseguir mais escalabilidade em termos do número de armazéns simulados. Empregamos um computador Pentium IV 2.0 GHz com 512 MB de memória RAM e com disco padrão IDE. O sistema operacional instalado é o Red Hat Linux 9, kernel 2.4.20-30.9, com ambiente gráfico GNOME. A compilação foi feita com os programas gcc e g++, versão 3.2.2. A carga transacional descrita na seção 4.2 foi gerada utilizando o *toolkit* OSDL DBT-2 versão 0.21.1 [44].

4.4 Implementação de Índices Hipotéticos

Para realizar a implementação de índices hipotéticos descrita na seção 3.1 foi necessário disponibilizar três modificações importantes no SGBD PostgreSQL:

1. Permitir a criação e remoção de índices hipotéticos no catálogo do sistema.

2. Alterar o otimizador para permitir que planos de execução sejam gerados tanto considerando quanto ignorando a existência de índices hipotéticos.

3. Modificar outros componentes do SGBD que devem ignorar a presença de índices hipotéticos.

Para implementarmos a primeira alteração, estendemos a relação que registra índices no catálogo para que esta tivesse um novo atributo para diferenciar índices reais de índices hipotéticos. Este atributo foi também adicionado às estruturas de dados correspondentes no sistema. Alteramos,

ainda, o analisador sintático (*parser*) para que reconhecesse os comandos `create hypothetical index` e `drop hypothetical index`. Por fim, implementamos rotinas chamadas a partir destes comandos para criar (ou remover) índices no catálogo do sistema sem invocar os procedimentos de construção física dos mesmos. Para a criação e remoção de índices hipotéticos, pudemos obter bloqueios menos restritivos sobre a tabela base do que os bloqueios obtidos pelo sistema para as operações sobre índices reais. Durante as operações sobre índices hipotéticos, somente são impedidas outras operações da linguagem de definição de dados sobre a tabela base, sendo possível processar consultas e atualizações normalmente.

Já para conseguirmos fazer com que o otimizador tanto pudesse considerar como ignorar os índices hipotéticos registrados no catálogo, observamos que o otimizador possui rotinas específicas para consulta ao catálogo e criação de estruturas de dados com informações estatísticas sobre tabelas e índices. Nosso ponto focal de alteração foram estas rotinas. Incluímos um modo de operação para que o otimizador pudesse decidir se quer ou não obter informações sobre índices hipotéticos. Alteramos as rotinas, ainda, para estimar os parâmetros estatísticos de índices hipotéticos conforme descrito na seção 3.1. Adicionamos, então, o comando `explain hypothetical` ao analisador sintático. Quando este comando é invocado, utilizamos o otimizador no modo que reconhece a existência de estruturas hipotéticas. Para todos os demais comandos, o otimizador opera como se as estruturas hipotéticas não estivessem presentes no catálogo.

Por fim, fizemos modificações nos outros componentes do SGBD que realizam operações sobre índices reais para que os índices hipotéticos fossem ignorados. Por exemplo, o comando `vacuum` do PostgreSQL, entre outras funções, obtém informações estatísticas sobre a quantidade real de tuplas e páginas de cada índice materializado no sistema. Alteramos o código deste e de alguns outros módulos para que não tentassem executar ações que seriam inconsistentes sobre índices hipotéticos.

É interessante observar que as mudanças realizadas para incluir o conceito de índices hipotéticos no sistema foram localizadas e de pouca complexidade. A maior parte do tempo investido na implementação se concentrou na análise do código do sistema e dos impactos que a nova funcionalidade traria. Um fator importante considerado na implementação é o de que caso o usuário do SGBD não queira utilizar a nova funcionalidade de índices hipotéticos este poderá trabalhar com o sistema como se a funcionalidade não existisse. Assim, não há impacto sobre as aplicações já escritas para o SGBD. A seguir, mostramos como os índices hipotéticos

podem ser utilizados no sistema.

Uso de Índices Hipotéticos

Para ilustrar o comportamento dos novos comandos adicionados, apresentamos um exemplo com o PostgreSQL modificado. Suponha que estejamos escrevendo uma consulta sobre a base de dados simples de vendas descrita na seção 4.2. Podemos ver o plano que o SGBD escolheria para a consulta utilizando o comando `explain`:

```
simple=# explain
simple-# select prodNum, data, sum(valor) as total
simple-# from   venda
simple-# where  valor > 1500000 and
simple-#        data between '20040101' and '20040131'
simple-# group by prodNum, data;
                          QUERY PLAN

-------------------------------------------------------------
 HashAggregate  (cost=25388.79..25388.83 rows=12 width=21)
   ->  Seq Scan on venda  (cost=0.00..25367.00 rows=2906
                           width=21)
         Filter: ((valor > 1500000::numeric) AND
                  (data >= '2004-01-01'::date) AND
                  (data <= '2004-01-31'::date))
(3 rows)
```

Podemos perceber que o SGBD fará uma varredura seqüencial sobre a tabela de vendas e aplicará os predicados sobre os atributos de valor e de data. A quantidade de tuplas esperada como resultado é de 2906 e o custo de processamento da varredura será de 25367.00. Após a varredura, um operador de agregação será aplicado para processar a cláusula `group by` do comando. O custo total esperado é de 25388.83.

Gostaríamos de verificar a utilidade de um determinado índice sobre a tabela de vendas para nossa consulta. Podemos criar um índice hipotético nos atributos importantes para os predicados conforme mostrado abaixo:

```
simple=# create hypothetical index hi_venda_valor_data
simple-# on venda (valor, data);
```

Repare que, ao fazermos isto, o índice não é materializado e, assim, o processamento de outras transações no sistema continua inafetado. Podemos, agora, verificar que plano seria escolhido pelo SGBD se este índice fosse materializado na base. Utilizamos o comando `explain hypothetical`:

```
simple=# explain hypothetical
simple-# select prodNum, data, sum(valor) as total
simple-# from   venda
simple-# where  valor > 1500000 and
simple-#        data between '20040101' and '20040131'
simple-# group by prodNum, data;
                          QUERY PLAN

-------------------------------------------------------------
 HashAggregate  (cost=10514.63..10514.66 rows=12 width=21)
   ->  Index Scan using hi_venda_valor_data on venda
         (cost=0.00..10492.83 rows= 2906 width=21)
         Index Cond: ((valor > 1500000::numeric) AND
                      (data >= '2004-01-01'::date) AND
                      (data <= '2004-01-31'::date))
(3 rows)
```

Podemos perceber que, caso existisse, o índice sobre as colunas de valor e data seria escolhido para o processamento da consulta. A quantidade esperada de tuplas a ser retornada pela varredura indexada seria de 2906 a um custo de 10492.83. Após a varredura indexada, seria aplicado o operador de agregação e o plano como um todo teria um custo de 10514.66, que é 58,6% inferior ao custo do plano que se utiliza de uma varredura seqüencial.

O novo índice traria benefícios à consulta que pretendemos executar. Caso esta consulta seja freqüente, devemos materializar o índice para acelerá-la. Devemos remover o índice hipotético e, então, criar o índice real utilizando o comando:

```
simple=# create index i_venda_valor_data on venda ( valor, data );
```

Uma vez criado o índice real, podemos verificar o plano que o SGBD escolheria para a consulta através do comando `explain`:

```
simple=# explain
simple-# select prodNum, data, sum(valor) as total
simple-# from   venda
```

```
simple-# where  valor > 1500000 and
simple-#        data between '20040101' and '20040131'
simple-# group by prodNum, data;
                          QUERY PLAN

-------------------------------------------------------------
 HashAggregate  (cost=10504.63..10504.66 rows=12 width=21)
   ->  Index Scan using i_venda_valor_data on venda
       (cost=0.00..10482.83 rows=2906 width=21)
         Index Cond: ((valor > 1500000::numeric) AND
                      (data >= '2004-01-01'::date) AND
                      (data <= '2004-01-31'::date))
(3 rows)
```

Como esperado, o plano de execução encontrado pelo SGBD é o mesmo que foi estimado quando criamos o índice hipotético. Entretanto, podemos reparar que espera-se que a varredura indexada retorne o mesmo número de tuplas, mas a um custo ligeiramente inferior ao anteriormente calculado. Esta diferença também torna o custo do plano como um todo inferior ao custo determinado quando consideramos o índice hipotético.

A diferença de custo pode ser explicada pelas estimativas realizadas para as estatísticas dos índices hipotéticos (veja a discussão na seção 3.1). Em especial, o número de páginas do índice é aproximado pelo número de páginas da tabela subjacente, o que gera estimativas conservadoras para os custos de acesso quando avaliamos um índice hipotético. Nas próximas subseções, aprofundaremos esta análise sobre a qualidade das estimativas geradas para índices hipotéticos e também abordaremos as diferenças de desempenho experimentadas na criação de índices hipotéticos e reais.

Desempenho de Índices Hipotéticos

Conduzimos diversos experimentos para avaliar as características de desempenho e de qualidade de nossa implementação de índices hipotéticos no SGBD PostgreSQL. Em nosso primeiro experimento investigamos a diferença de tempo de criação existente entre índices reais e índices hipotéticos. Esta diferença justifica, em grande parte, o uso de índices hipotéticos para simular configurações de índices em grandes bases de dados.

Populamos nossa base simples de vendas (ver seção 4.2) com quantidades de tuplas crescentes na tabela de vendas em incrementos de 50 mil. A cada tamanho da tabela de vendas, fizemos dez medidas dos tempos de

criação de um índice real e de um índice hipotético sobre as colunas de valor e data. Cada ponto mostrado no gráfico da Figura 4.5 é a média dos tempos obtidos nestas dez medições. A medição do tempo de execução dos comandos foi feita através da opção de *timing* do cliente `psql` para o SGBD PostgreSQL [46]. Tanto cliente quanto servidor foram executados na mesma máquina, o que implica que os tempos obtidos não foram afetados pelo tempo de comunicação via rede.

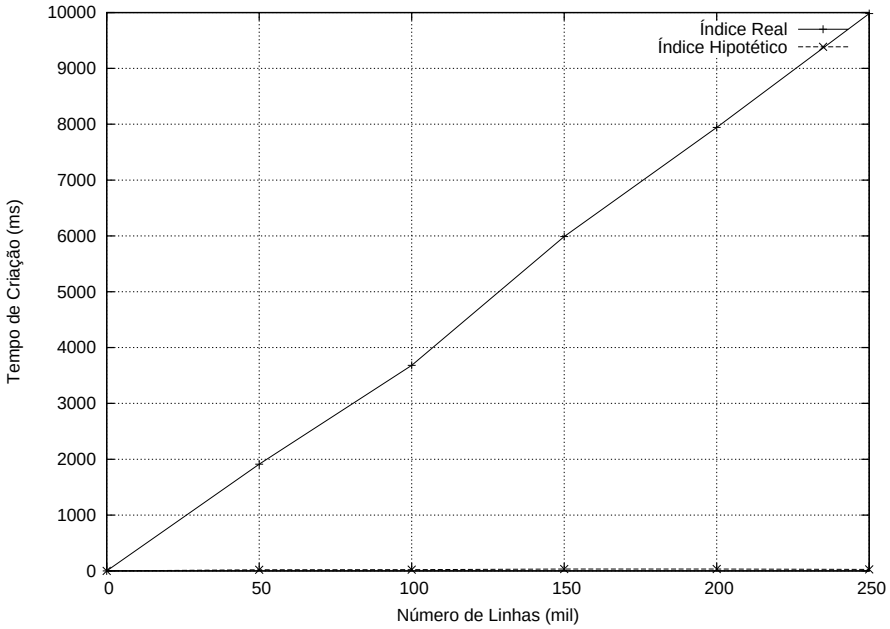

Figura 4.5: Tempo de Criação de Índices Reais e Hipotéticos

A Figura 4.5 mostra, portanto, o comportamento do tempo de criação de índices reais e hipotéticos à medida que aumentamos a quantidade de tuplas da tabela subjacente. Podemos perceber que o tempo de criação de índices reais cresce de forma aproximadamente linear à medida que a tabela cresce de tamanho. Índices hipotéticos possuem um tempo de criação praticamente constante e, em nosso ambiente, igual a poucos centésimos de segundo.

Um usuário poderia, portanto, experimentar índices hipotéticos em grandes bases de dados trazendo pouquíssimo impacto de desempenho e disponibilidade sobre as transações sendo concorrentemente executadas no sistema. Cabe observar que no PostgreSQL não é possível criar índices reais *on-line* (veja a seção 3.1) e, assim, seria necessário interromper as operações concorrentes de atualização sobre a tabela durante a criação de índices. Ainda que o sistema oferecesse este tipo de funcionalidade, o impacto de

desempenho de materializar um novo índice e concorrentemente processar transações sempre estaria presente.

Qualidade de Índices Hipotéticos

Para obtermos estes benefícios do uso de índices hipotéticos, é necessário que a qualidade de estimativa de custos e de planos de execução com configurações hipotéticas seja aceitável. Ao criar o índice real correspondente a algum índice hipotético escolhido, o usuário deve ter a certeza de que o otimizador levará em conta o novo índice na construção de planos de execução da mesma forma que foi simulada com a configuração hipotética.

Montamos alguns experimentos adicionais para procurar validar a qualidade das estatísticas criadas para índices hipotéticos. Tomamos alguns tipos de consultas que fazem usos distintos dos índices presentes no sistema (para uma classificação, ver Shasha e Bonnet [53]). Estas consultas podem ser vistas na Tabela 4.2.

Tipo de Consulta	Texto da Consulta
Consulta por intervalo e agregação	select prodNum, data, sum(valor) as total from venda where valor > 1500000 and data between '20040101' and '20040131' group by prodNum, data;
Consulta pontual	select * from venda where num = 100;
Consulta de ordenação	select * from venda order by num;

Tabela 4.2: Algumas consultas utilizadas para avaliação de estatísticas para índices hipotéticos

Nosso critério foi o de selecionar consultas que fazem acessos a diferentes quantidades de páginas dos índices utilizados. A primeira consulta faz uma restrição de intervalos e, posteriormente, uma agregação. Um índice sobre valor e data pode acelerar o processamento da restrição. Espera-se que um subconjunto das tuplas da tabela seja retornado pela consulta.

A segunda consulta é uma consulta pontual, ou seja, acessa uma única linha da tabela. A coluna de número da venda é um atributo seqüencial que identifica a ordem de entrada das vendas no sistema. Um índice sobre este

atributo poderia acelerar esta consulta. Por fim, a terceira consulta traz todas as tuplas da tabela ordenadas pelo atributo de número da venda. Há uma correlação muito alta entre a ordem deste atributo e a ordem física das tuplas presentes na tabela de vendas. Por conta desta propriedade, o sistema poderia utilizar um índice sobre o atributo de número para recuperar todas as tuplas da tabela. A vantagem seria a eliminação de uma operação de ordenação para garantir a ordem desejada na cláusula `order by` do comando[3].

Para cada uma destas consultas criamos os índices discutidos acima como índices hipotéticos e índices reais. Obtivemos, então, os planos de execução das consultas e comparamos os custos de processamento das varreduras indexadas e dos planos como um todo. Repetimos este procedimento para diversos tamanhos da tabela de vendas, em incrementos de 200 mil tuplas.

Número de Tuplas (mil)	**Dif. Custo de Varredura**	**Dif. Custo Total**
200	0,00%	0,00%
400	0,12%	0,12%
600	0,11%	0,11%
800	0,10%	0,10%
1000	0,10%	0,10%

Tabela 4.3: Qualidade de índice hipotético em consulta por intervalo e agregação

A Tabela 4.3 mostra os resultados obtidos para a consulta por intervalo. São mostrados o número de tuplas presente na tabela de vendas, a diferença percentual entre o custo da varredura indexada no plano de execução hipotético e no plano de execução real e a diferença percentual entre o custo total do plano de execução hipotético e do plano de execução real.

Para a consulta de intervalos, as diferenças entre os custos estimados com índices hipotéticos e com índices reais permaneceu baixa e em torno de 0,11%. Conforme discutido na seção 3.1, para a classe de índices que consideramos no nosso trabalho, o único parâmetro que é superestimado na avaliação de custos de índices hipotéticos é o número de páginas do índice. Este é aproximado pelo número de páginas da tabela. Assim, em

[3]Em alguns SGBDs, como o Microsoft SQL Server [41], índices somente são usados para eliminar ordenações quando estes são definidos como índices primários. Como não é possível criar índices primários no PostgreSQL, o sistema possui a funcionalidade de se aproveitar de índices secundários para ordenações quando estes índices possuem com alto grau de correlação com a tabela subjacente.

uma consulta que retorna um subconjunto das tuplas da tabela, o otimizador estima custos de acesso com índices hipotéticos ligeiramente superiores aos custos estimados quando o índice é de fato materializado.

Repare que não tivemos qualquer diferença de estimativas quando a tabela possuía 200 mil tuplas. Isto ocorre pois não existiam tuplas que satisfizessem o predicado da consulta quando a tabela era deste tamanho. Outra conseqüência interessante da sistemática de geração de tuplas para a tabela de vendas (veja a seção 4.2) é a pouca variação nas diferenças de custos estimados. Isto se deve ao fato de sempre gerarmos valores de forma aleatória (respeitando uma faixa de valores válidos) e de termos apenas campos de tamanho fixo na tabela, ocasionando uma seletividade proporcional para diferentes tamanhos de tabela. É relevante observar que esta proporcionalidade se reflete nas diferenças de custos, mesmo quando temos tabelas de tamanhos bastante diferentes.

Para a consulta pontual, as estimativas de custos realizadas com índices hipotéticos foram idênticas às realizadas com índices reais. No caso desta consulta, o número de páginas do índice tem pouca influência no cálculo de custos uma vez que somente uma página da tabela será acessada. Desta forma, as diferenças de estimativas de custo foram nulas.

Em alguns outros testes também observamos excelente qualidade de estimativa de custo de varreduras indexadas em consultas que realizam junções de *loops* aninhados e que se utilizam de um índice seletivo para acessar a tabela interna.

Número de Tuplas (mil)	**Dif. Custo de Varredura**	**Dif. Custo Total**
200	26,88%	26,88%
400	26,92%	26,92%
600	26,93%	26,93%
800	26,94%	26,94%
1000	26,94%	26,94%

Tabela 4.4: Qualidade de índice hipotético em consulta de ordenação

Já para a consulta de ordenação, obtivemos os resultados mostrados na Tabela 4.4, com as diferenças de custos maiores e em torno de 27%. Neste tipo de consulta percorremos todas as páginas do índice e, assim, a estimativa feita para a quantidade de páginas é de maior relevância. Como o plano de execução possui apenas a varredura indexada, este também apresenta o mesmo erro de estimativa. Novamente devido à sistemática de geração de tuplas para a tabela de vendas, as diferenças percentuais entre os custos hipotéticos e reais permaneceram praticamente constantes. Podemos

dizer que, para a nossa aproximação conservadora do número de páginas de índices hipotéticos, este é o pior caso em termos de estimativas de custos.

Também reparamos em outros testes que, em consultas que realizam junções por ordenação e fusão (*merge join*) e em que o índice é varrido para obter todas as tuplas da tabela em ordem, houve degradação de qualidade nas estimativas de custo geradas para índices hipotéticos. Novamente, neste tipo de consulta, o número de páginas do índice que serão acessadas é superestimado.

Em resumo, observamos que, em uma grande variedade de situações, as estimativas que realizamos para calcular as estatísticas de índices hipotéticos apresentam resultados de custo muito próximos dos observados para os índices reais equivalentes. A degradação de qualidade das estimativas ocorre apenas em consultas em que uma grande parcela do índice é percorrida. Isto se deve ao fato de que aproximamos o número de páginas do índice pelo número de páginas da tabela subjacente. Não houve tempo hábil para investigar outras funções de estimativa que pudessem gerar resultados mais precisos nestes casos específicos. Acreditamos, entretanto, que, como houve consistência nas escolhas dos planos de execução, a aproximação conservadora não será prejudicial para o usuário do sistema.

4.5 Implementação da Arquitetura com um Único Agente

Nesta seção, abordaremos a implementação da arquitetura com um único agente proposta no Capítulo 3 utilizando o SGBD PostgreSQL. Primeiramente, na seção 4.5.1, apresentamos um projeto detalhado da arquitetura, discutindo a forma de integração entre o agente e o PostgreSQL, a heurística utilizada para escolha de índices candidatos e as classes, padrões de projeto e interações empregados em cada camada do *framework* de construção de agentes de Kendall et al. [35]. Em seguida, na seção 4.5.2, debatemos os resultados experimentais obtidos com o uso do agente quando o SGBD processa a carga transacional do teste DBT-2, descrito na seção 4.2.

4.5.1 Projeto do Agente

O *framework* abstrato de [35] em que nos baseamos para a construção de nosso agente foi implementado por seus autores na linguagem Java. Inici-

amos nossa implementação a partir de uma revisão do porte do *framework* para C++ conduzido em [40]. Esta revisão, e a posterior integração do agente criado com o SGBD PostgreSQL, foram conduzidas em conjunto com o trabalho de Milanés [42]. O resultado desta cooperação é uma infra-estrutura que possibilita embutirmos um sistema com múltiplos agentes no SGBD. Nesta dissertação, utilizamos esta infra-estrutura com apenas um agente criado para a escolha automática de índices.

Para a construção do *Agente de Benefícios*, instanciamos o *framework* portado para C++ utilizando as camadas indicadas na seção 3.2. A seguir, comentamos as dificuldades encontradas na integração do agente com o PostgreSQL, apresentamos uma heurística adaptada da literatura para escolha de índices candidatos, detalhamos como ocorreu a extensão do *framework* em cada camada implementada no agente e mostramos a forma de cálculo de estimativas feitas para variáveis empregadas na heurística de benefícios.

Integração entre Agente e PostgreSQL

Uma primeira dificuldade para a integração entre o sistema de agentes e o PostgreSQL veio do fato de que o primeiro é escrito em C++, enquanto o segundo é escrito em C e usa, internamente, variáveis cujos nomes coincidem com palavras reservadas da linguagem C++. Seguimos as recomendações de [14] sobre compilação, ligação e uso de diretivas de código para integrar código escrito nestas duas linguagens.

Isolamos todas as rotinas de interface entre o nosso agente e o SGBD em um módulo específico. Todas as rotinas utilizadas como efetuadores das ações do agente sobre o SGBD ficaram concentradas neste módulo. Esta decisão permitiu o isolamento do código dependente do PostgreSQL. Assim, caso desejemos acoplar o agente proposto a outro SGBD, as alterações no código estariam concentradas neste módulo de interface.

Outra questão importante a ser decidida foi sobre a forma de encaixar o agente no modelo de processos implementado pelo PostgreSQL. Conforme visto na seção 4.1, o SGBD cria um processo no servidor para cada conexão realizada por um cliente. Estas conexões são aceitas por um processo mestre.

Uma primeira alternativa investigada para a execução do agente foi o uso de *threads* criadas a partir do processo mestre. Como as *threads* compartilham o espaço de endereçamento do processo mestre, enfrentamos dificuldades para a execução de ações por parte do agente. As ações

modificavam o estado das variáveis do processo mestre, atrapalhando o seu correto funcionamento.

Assim, para a execução do agente, preferimos instanciar um novo processo a partir do processo mestre (`postmaster`) assim que o sistema é inicializado. Estabelecemos como variável global, herdada por todos os demais processos do servidor, a chave de uma fila de mensagens do sistema operacional [62] que permite a comunicação com o agente. Todo processo que deseja enviar notificações para o agente utiliza rotinas de passagem de mensagens para alcançar seu propósito. Esta disposição dos processos é mostrada na Figura 4.6.

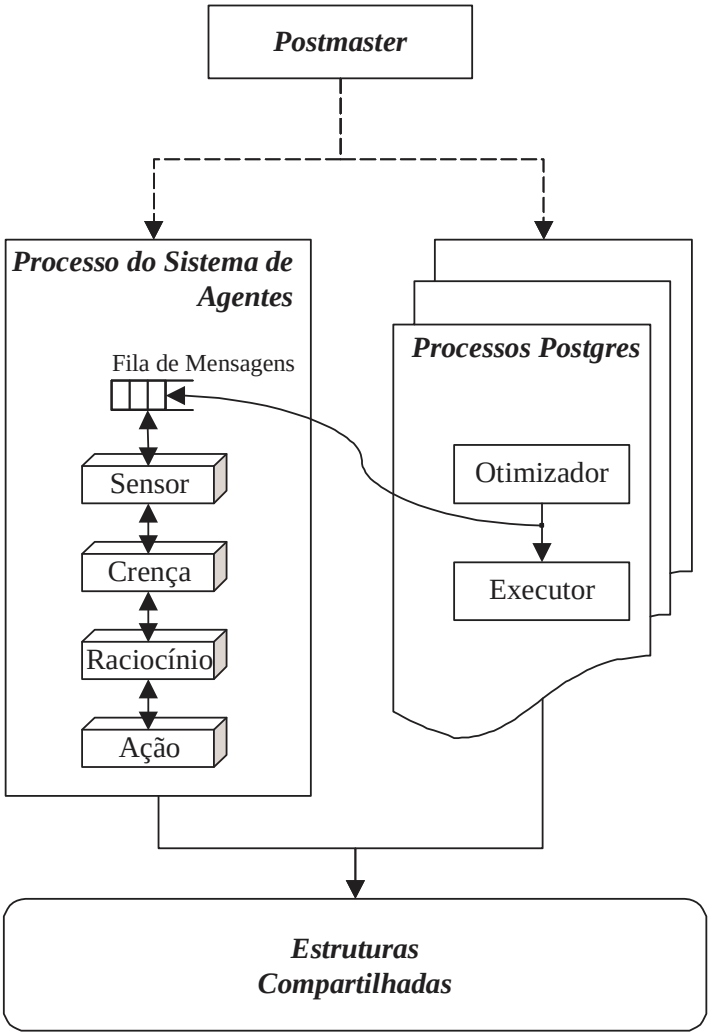

Figura 4.6: Integração entre o agente e o PostgreSQL

Conforme discutido no Capítulo 3, é interessante instrumentar o SGBD para que este envie para o agente os comandos SQL processados juntamente com informações sobre estimativas de custos. Assim, alteramos o

código do sistema para que cada processo `postgres` enviasse uma mensagem com estas informações para a fila de mensagens do agente. A camada *Sensor* processa as mensagens que são depositadas na fila, procedendo, então, com a atualização das crenças do agente.

Além da integração necessária para recuperar informações sobre os comandos sendo processados pelo SGBD, também foi necessário permitir que o agente realizasse ações sobre o SGBD. Entre estas ações estão a simulação de configurações hipotéticas e a criação de índices. Logo após instanciarmos o processo para execução do agente a partir do processo mestre, executamos uma rotina especial de inicialização. A rotina faz todas as chamadas necessárias aos módulos do SGBD para fazer com que o processo do agente tenha as mesmas características de execução de comandos de um processo `postgres` convencional.

As principais diferenças existentes entre um processo `postgres` e o processo do agente são o fato de que o processo do agente não está atrelado a um canal de comunicação aberto com um cliente e o de que apenas este processo de fato executa o código do agente. Lembramos que o agente possui acesso aos módulos do SGBD (e vice-versa) pois, seguindo os preceitos da arquitetura embutida (veja seção 2.2), o código dos dois sistemas é ligado em um único executável.

Heurística de Escolha de Índices Candidatos

Conforme debatido na seção 3.2, podemos utilizar uma heurística já proposta na literatura para a escolha de índices candidatos no agente. Estudamos diversas heurísticas, entre elas [25, 24, 12, 37], e procuramos por trabalhos que tivessem implementações junto a SGBDs existentes.

Escolhemos para nossa implementação a heurística de Lohman et al. [37], que se destaca por sua simplicidade e eficiência. Ela foi implementada no sistema IBM DB2 UDB [34] e permite que a escolha de índices candidatos seja realizada para um dado comando SQL com a execução de apenas uma chamada ao otimizador do SGBD. A heurística analisa predicados e cláusulas presentes no comando SQL submetido para encontrar colunas que poderiam ser indexadas. Os usos interessantes de colunas são classificados e combinados para formar índices hipotéticos. Todos os índices hipotéticos enumerados são criados no sistema e, então, uma chamada é feita ao otimizador. Os índices hipotéticos escolhidos pelo otimizador no plano de execução gerado são recomendados como índices candidatos para o comando.

A heurística procura encontrar no comando SQL cinco tipos de usos interessantes de colunas:

1. EQ: colunas envolvidas em predicados de igualdade;
2. O: colunas envolvidas em cláusulas ORDER BY, GROUP BY e predicados de junção;
3. RANGE: colunas que aparecem em restrições de intervalos;
4. SARG: colunas que aparecem em outros predicados indexáveis (por exemplo, `like`). Este uso de colunas remete ao conceito de predicados de busca (*sargable predicates*) discutido em [52];
5. REF: demais colunas referenciadas no comando SQL.

Em nossa implementação da heurística, identificamos, ainda, mais um grupo de usos interessantes de colunas, denominado *BAD*. Este grupo é formado por colunas afetadas por comandos de atualização. Em comandos do tipo `insert` e `delete`, são todas as colunas das tabelas atualizadas; em comandos do tipo `update`, são as colunas referenciadas na cláusula `SET`. Este grupo de usos de colunas será interessante para determinarmos, posteriormente, quais índices são prejudicados pelo comando.

A heurística, então, combina os grupos de colunas das seguintes formas para formar índices hipotéticos com múltiplas colunas, em ordem, eliminando colunas duplicadas:

- EQ + O
- EQ + O + RANGE
- EQ + O + RANGE + SARG
- EQ + O + RANGE + REF
- O + EQ
- O + EQ + RANGE
- O + EQ + RANGE + SARG
- O + EQ + RANGE + REF

Estas combinações procuram refletir usos típicos de índices para acelerar consultas [37]. Para identificar índices mais simples, por exemplo com uma coluna, a heurística original propõe, após a realização das combinações

acima, que a lista de índices seja complementada com uma etapa de enumeração exaustiva de índices envolvendo as colunas selecionadas. Esta enumeração seria interrompida por um limite de tempo.

Em nosso cenário de um agente executado de forma embutida no servidor, o uso deste tipo de estratégia exaustiva de enumeração pode representar um consumo de recursos e um atraso na escolha de índices para o comando indesejáveis. Assim, limitamos a busca executada pela heurística à enumeração dos índices possíveis de uma coluna envolvendo os grupos de colunas EQ, O, RANGE e SARG. Estes grupos representam os usos de colunas em predicados que admitem indexação.

A heurística aqui descrita foi implementada na camada de *Raciocínio* do agente. A seguir, iremos detalhar como foi realizada a implementação de cada camada empregada.

Configuração das Camadas

Nesta subseção e nas próximas, iremos detalhar as classes e padrões de projeto empregados na construção do agente utilizando notação UML [6]. Como uma descrição do *framework* que utilizamos já está disponível em [35], iremos focar nossa atenção em como fizemos a extensão dos *hot spots* disponíveis para implementar o *Agente de Benefícios*.

A Figura 4.7 mostra as classes envolvidas na configuração das camadas do agente. Quando o SGBD é iniciado com o sistema de agentes ativado, é criada uma instância da classe *IndexAgent*, que representa o *Agente de Benefícios*. Em nossa implementação, a ativação do agente é opcional e pode ser escolhida com o uso de um parâmetro de inicialização do SGBD.

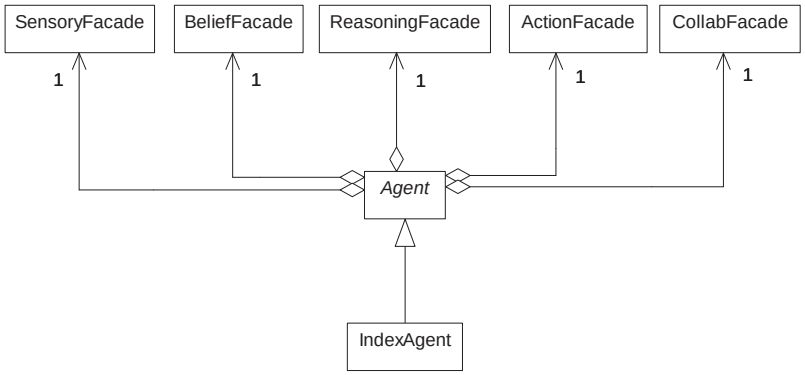

Figura 4.7: Diagrama de classes envolvidas na configuração das camadas

Como a classe *IndexAgent* estende a classe abstrata *Agent* do *framework*, sua instância está relacionada com as instâncias das classes que

representam as camadas do agente (*SensoryFacade*, *BeliefFacade*, *ReasoningFacade*, *ActionFacade* e *CollabFacade*). Cada camada é representada por uma classe que implementa o padrão de projeto *Facade* [26], o que cria um ponto de acesso único para cada camada.

A instância de *IndexAgent* fornece para cada camada os sensores, crenças, planos e ações que serão utilizados nesta implementação do agente. Em seguida, inicia o processo de recebimento de notificações provenientes do SGBD acionando a camada *Sensor*.

Camada Sensor

A camada *Sensor* é responsável por obter as informações provenientes do ambiente em que está inserido o agente. A Figura 4.8 mostra as classes presentes nesta camada. As classes *BeliefFacade* e *ABelief* pertencem à camada de *Crença* e são mostradas no diagrama para melhor compreensão. A classe *SQLInformation* é geral e não pertence à camada *Sensor*. Ela encapsula as informações associadas ao processamento de um comando SQL vindas do SGBD e, assim, será utilizada em diversas outras camadas.

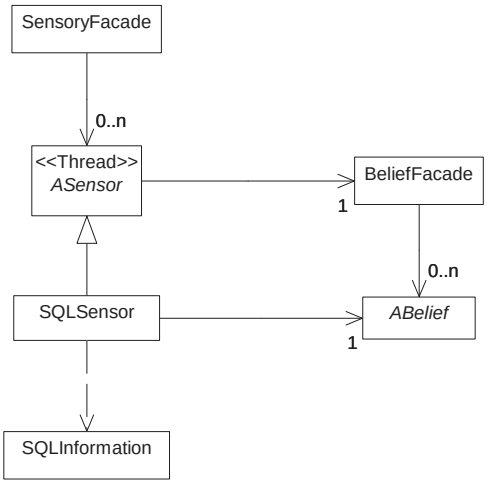

Figura 4.8: Diagrama de classes da camada *Sensor*

A principal extensão do *framework* realizada na camada *Sensor* é a implementação da classe *SQLSensor*. Uma instância desta classe é criada na inicialização do agente e inicia uma nova *thread* que lê a fila de mensagens criada para integração do agente com o SGBD. É nessa *thread* que todo o processamento subseqüente do agente ocorre.

Repare que a classe *SQLSensor* é uma extensão de *ASensor*. Apesar de termos em nossa implementação criado apenas um sensor para recuperar

as informações de comandos SQL, juntamente com seus custos e estatísticas, é possível criar diversos sensores na camada. Por exemplo, poderíamos criar um sensor para medir o consumo de recursos do sistema, como CPU e memória. Caso o consumo se tornasse excessivo, a camada *Raciocínio* poderia tomar uma decisão de parar o agente. Outro sensor poderia realizar a medição do espaço livre em disco no sistema. Esta informação seria utilizada para evitar criações de índices que comprometessem a disponibilidade do SGBD. É interessante observar que podemos derivar comportamentos mais inteligentes de sintonia se cruzarmos informações sensoriadas em diversos pontos distintos do SGBD. Este tipo de estratégia é empregada em trabalhos de auto-sintonia global, como por exemplo [42].

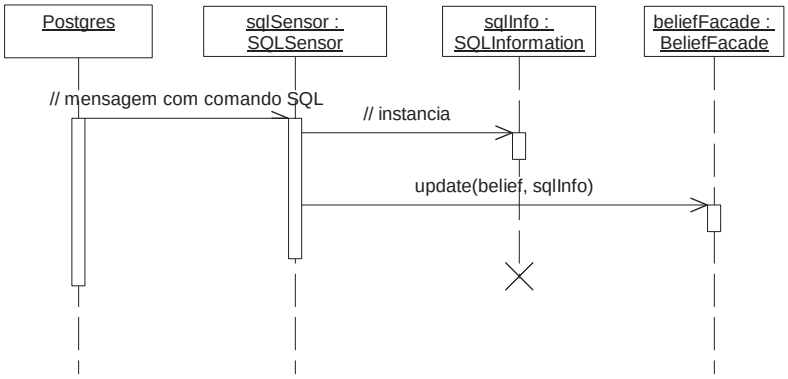

Figura 4.9: Uma interação típica na camada *Sensor*

A Figura 4.9 mostra uma interação típica envolvendo instâncias das classes da camada *Sensor*. O SGBD envia assincronamente uma mensagem para o sensor do agente. Quando o sensor recupera a mensagem, ele instancia a classe *SQLInformation* para encapsular as informações recebidas. Esta instância e a crença associada ao sensor são informadas à fachada da camada *Crença*. Nesta camada, a atualização da crença do agente com as novas informações irá ocorrer.

Camada Crença

A camada *Crença* é responsável por representar o conhecimento obtido pelo agente sobre o seu ambiente e por armazenar métricas derivadas a partir deste conhecimento. Na Figura 4.10, vemos as classes desta camada. A classe *ReasoningFacade* pertence à camada *Raciocínio* e as classes *Observer* e *Subject* são uma implementação geral do padrão de projeto *Observer* [26]. Este padrão de projeto permite implementar um mecanismo de notificação.

Toda vez que uma instância da classe *Subject* sofre alguma modificação, todas as instâncias registradas da classe *Observer* são avisadas da ocorrência de mudanças.

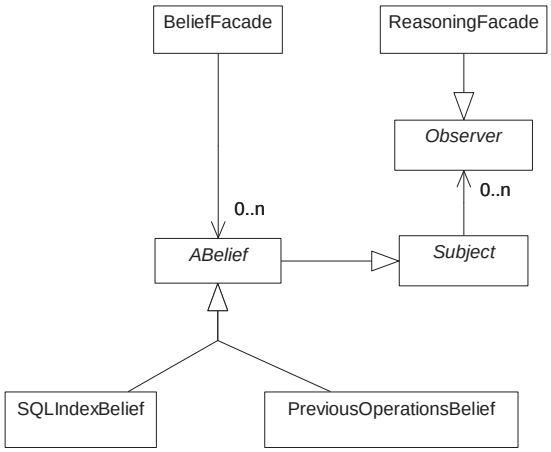

Figura 4.10: Diagrama de classes da camada *Crença*

As classes *SQLIndexBelief* e *PreviousOperationsBelief* representam as crenças que o agente possui, sendo, assim, extensões da classe abstrata *ABelief*. A primeira representa o comando SQL recebido do SGBD, juntamente com suas informações associadas de custos e estatísticas. Já a classe *PreviousOperationsBelief* armazena as informações sobre benefícios acumulados para cada índice candidato. Esta classe dá suporte para a implementação da heurística de benefícios (veja a seção 3.2).

Nesta camada, encontramos um uso do padrão de projeto *Observer* [26]. Este padrão de projeto é utilizado para implementar um mecanismo de notificação de mudanças. Toda vez que ocorre uma mudança em alguma crença do agente (*ABelief*) é feita uma chamada à instância da classe *ReasoningFacade*, que representa a camada de *Raciocínio*. Uma conseqüência do uso do padrão de projeto *Observer* é o desacoplamento entre as camadas *Crença* e *Raciocínio*. O fluxo de informação pode ser melhor compreendido na Figura 4.11.

Quando a camada *Sensor* detecta que um novo comando SQL foi submetido ao SGBD, ela envia esta informação para a fachada da camada de *Crenca*. A instância de *BeliefFacade* irá, então, atualizar a crença fornecida pelo sensor com o novo valor. A crença notifica todos os seus observadores. No caso, a fachada *ReasoningFacade*, que observa todas as crenças, será atualizada com as novas informações vindas do SGBD. Na camada *Raciocínio*, o novo comando SQL será analisado e será decidido se devemos materializar algum índice.

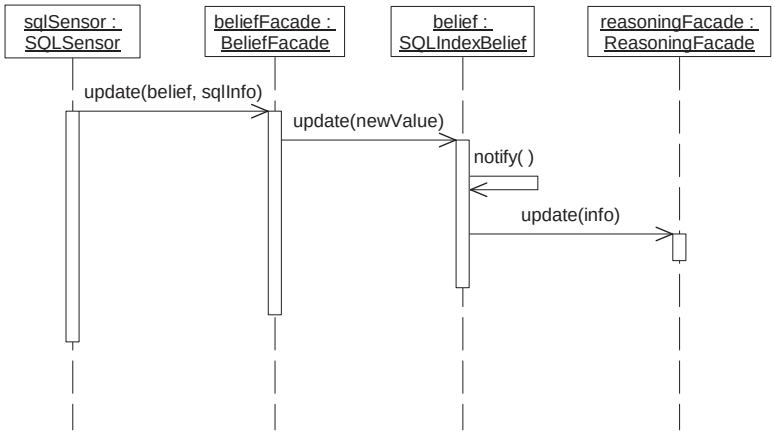

Figura 4.11: Uma interação típica na camada *Crença*

Camada Raciocínio

A camada de *Raciocínio* é responsável por decidir como o agente irá interagir com o seu ambiente de forma a alcançar seus objetivos. Em nossa implementação, incluímos explicitamente as heurísticas para escolha de candidatos e de benefícios como planos nesta camada. Alguns trabalhos da literatura de agentes adotam, na implementação desta camada, o uso de mecanismos genéricos de inferência para que o agente calcule suas próximas ações [66]. Não seguimos esta abordagem pois já existem soluções algorítmicas desenvolvidas para a seleção de índices [25, 24, 12, 37]. As classes utilizadas podem ser vistas na Figura 4.12.

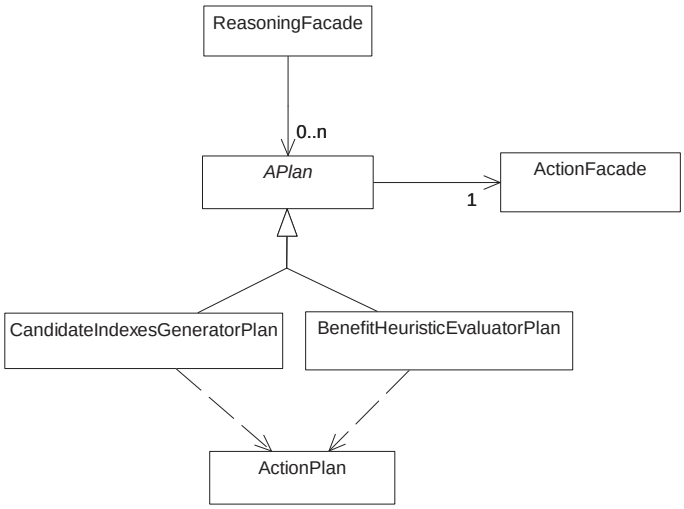

Figura 4.12: Diagrama de classes da camada *Raciocínio*

As subclasses de *APlan* representam os planos que podem ser aplicados às crenças do agente. A primeira subclasse, *CandidateIndexesGenera-*

torPlan, representa nossa implementação da heurística de escolha de candidatos descrita anteriormente.

Quando a crença sobre os comandos SQL processados pelo sistema é atualizada, a fachada da camada de *Raciocínio* delega para todos os seus planos a responsibilidade de decidir se devem agir sobre a crença atualizada. Esta é uma aplicação do padrão de projeto *Chain of Responsibility* [26], em que cada objeto de uma lista pode decidir se deve ou não tratar uma requisição. O plano instanciado a partir da classe *CandidateIndexesGeneratorPlan* decide processar esta crença. São enumerados, então, os índices que potencialmente trazem benefícios para a consulta. Em seguida, estes índices hipotéticos são criados no SGBD e é feita uma chamada ao otimizador do sistema para obter um plano de execução que os leva em consideração. Estas ações de interação com o SGBD são criadas pela instância da classe *CandidateIndexesGeneratorPlan* como elementos de um plano de ações (*ActionPlan*). Este plano de ações é informado para a fachada da camada *Ação*, que o executa interagindo com o SGBD através de efetuadores. Este fluxo de informações pode ser visto na Figura 4.13.

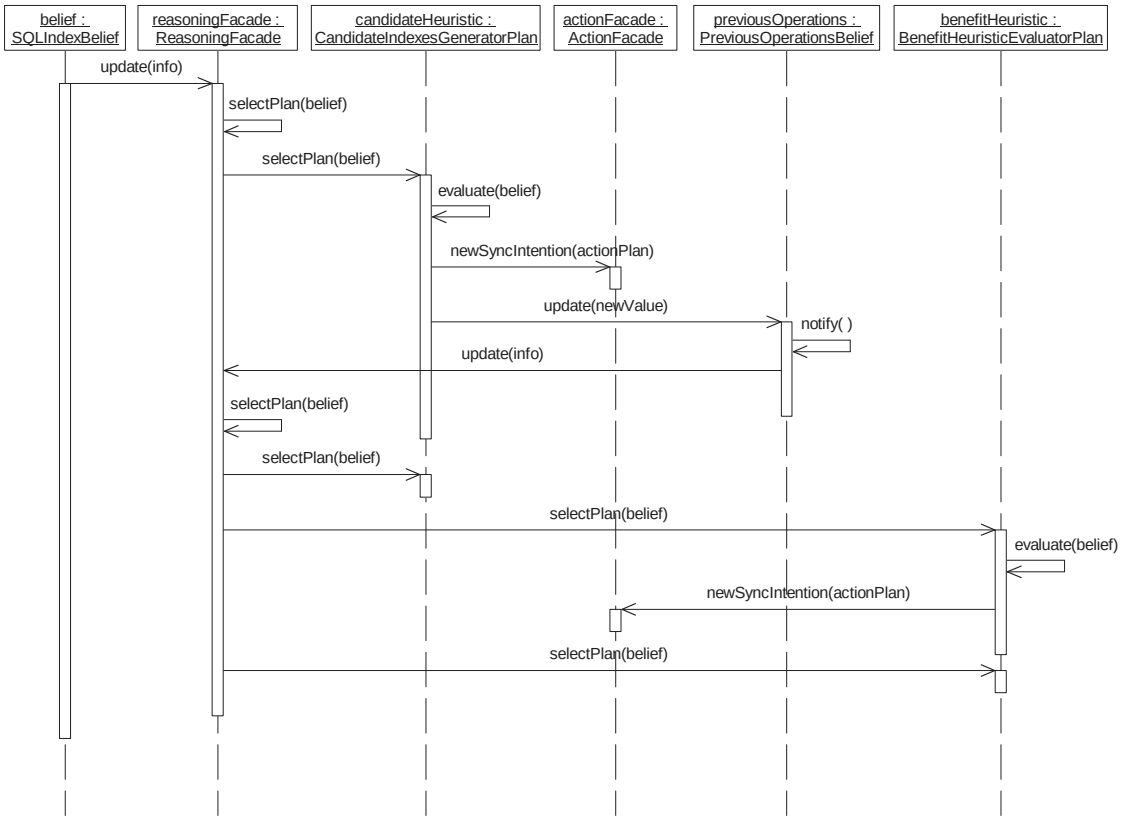

Figura 4.13: Uma interação típica na camada *Raciocínio*

Assim que a heurística de escolha de candidatos obtém o retorno de

suas ações sobre o SGBD, ela atualiza a crença que armazena benefícios acumulados (*PreviousOperationsBelief*) com os índices candidatos encontrados. A atualização desta crença gera uma notificação para a fachada da camada *Raciocínio*. Isto é implementado através do padrão de projeto *Observer*, explicado na subseção anterior.

A fachada de *Raciocínio* dispara, então, a avaliação do plano representado pela segunda subclasse da classe *APlan*, a classe *BenefitHeuristicEvaluatorPlan*. Esta última representa a heurística de benefícios apresentada na seção 3.2.

Em nossa implementação, tratamos da parte da heurística de benefícios relativa a índices candidatos e, portanto, à criação de novos índices. Priorizamos a obtenção de uma implementação consistente para a criação automática de índices, uma vez que isto já permite que utilizemos o agente em bases de dados em que nenhum trabalho manual de seleção de índices tenha sido realizado. Conforme veremos na seção 4.5.2, a ausência da remoção automática de índices não prejudicou nossos testes com o agente.

Caso a heurística de benefícios decida que novos índices devem ser criados na base de dados, será feita uma chamada à fachada da camada de *Ação* com um plano de ações adequado. A camada de *Ação* oferece serviços para a interação com o SGBD.

Camada Ação

A camada de *Ação* é responsável pela execução das decisões tomadas na camada *Raciocínio*. Estas decisões são codificadas em planos de ação e executadas através de intenções. As classes utilizadas podem ser vistas na Figura 4.14. No *framework* de [35], as intenções podem ser de dois tipos: reações e colaborações. Em nossa implementação, como temos um único agente, apenas reações são empregadas (classe *ReactionIntention*).

Uma característica interessante de nossa implementação do *framework* de [35] para C++ é a de que reações podem ser executadas de forma síncrona ou assíncrona. No formato síncrono, a intenção é executada na própria *thread* corrente; já no formato assíncrono, é criada uma nova *thread* para execução da intenção. No *framework* original, somente estavam disponíveis reações assíncronas.

Conforme vimos anteriormente, o agente é executado em um processo que possui características similares a de um processo `postgres`. É uma característica do SGBD a execução de apenas um comando por vez dentro

do contexto de um processo deste tipo. Assim, para manter esta consistência, todas as reações executadas pelo agente são iniciadas de forma síncrona.

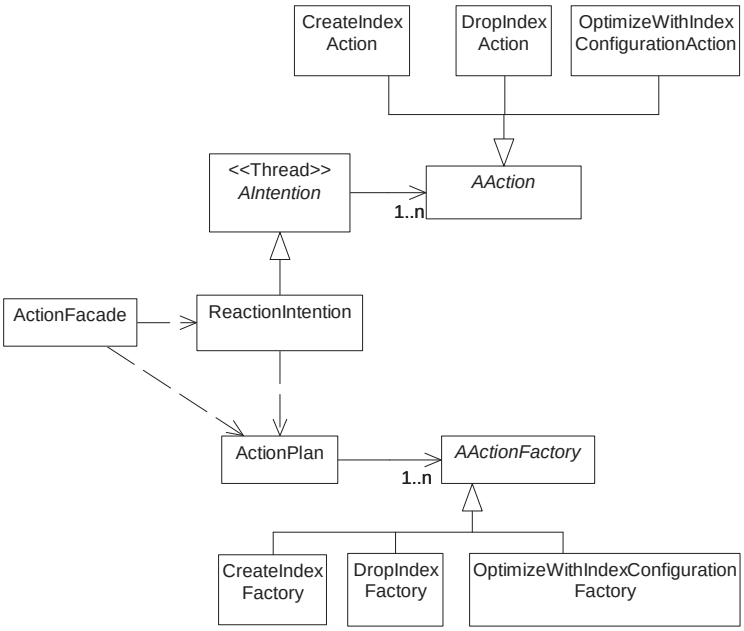

Figura 4.14: Diagrama de classes da camada *Ação*

Um plano de ação possui um conjunto de fábricas de ações. Utilizamos o padrão de projeto *Abstract Factory* [26] para permitir que apenas fábricas sejam associadas aos planos de ação, deixando a instanciação das ações propriamente ditas para o momento de sua execução. As fábricas possuem a capacidade de construir o objeto do tipo correto no momento em que a instanciação é requerida. As etapas para a execução de uma ação de criação de um índice são mostradas na Figura 4.15.

Inicialmente, a fachada da camada *Ação* recebe o pedido de execução de uma intenção de forma síncrona. A fachada repassa o plano de ações para uma reação, que trata de recuperar as ações adequadas. As instanciações das ações partem das fábricas de ações configuradas no plano de ações. Na seqüência, as ações são executadas dentro do contexto da reação. Cada ação interage com o SGBD através de efetuadores implementados num módulo de interface.

No cenário da Figura 4.15, temos apenas uma ação de criação de um índice. Esta não é a única ação disponível. As ações implementadas nesta camada são:

- criação de índices reais e hipotéticos (classe *CreateIndexAction*);
- remoção de índices reais e hipotéticos (classe *DropIndexAction*);

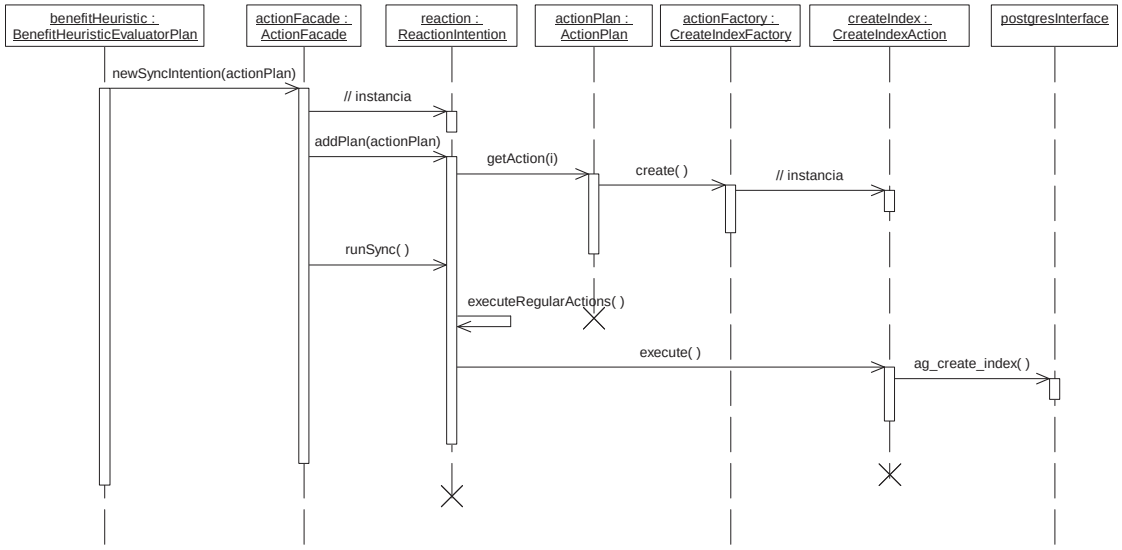

Figura 4.15: Uma interação típica na camada *Ação*

– otimização de um comando SQL sobre a atual configuração de índices reais e hipotéticos (classe *OptimizeWithIndexConfigurationAction*).

Estimativas de Custos de Atualização e Criação de Índices

Um ponto importante a lembrarmos a respeito da heurística de benefícios é o de que existem algumas grandezas, como o custo de atualização de um índice e o custo de criação de um índice, que devem ser estimadas segundo informações do modelo de custos do SGBD. Após um estudo do código de cálculo de custos do PostgreSQL, definimos a seguinte aproximação para o custo de atualização de um índice:

$$C_A = 2 \left\lceil \frac{r}{R} \right\rceil P + cr$$

Na fórmula, r é o número de tuplas atualizadas pelo comando, R é o número de tuplas da tabela, P é o número de páginas da tabela e c é um coeficiente que relaciona, percentualmente, o custo de uma operação de E/S com o custo de CPU para processar uma tupla que esteja em memória. No PostgreSQL, cada E/S tem custo estimado igual a um e o coeficiente c tem um valor padrão de 1%.

O primeiro termo da fórmula calcula o custo de E/S da atualização. Estimamos que iremos atualizar uma quantidade de páginas de índice proporcional à fração de registros da tabela que foram atualizados multiplicada pelo tamanho da tabela em páginas. Lembramos que, para índices

hipotéticos, estimamos que o tamanho em páginas do índice é idêntico ao tamanho em páginas da tabela. Multiplicamos o resultado por dois pois, no pior caso, precisamos ler e escrever cada página atualizada. Além do custo de E/S, ainda somamos o custo estimado de CPU de processar o número de registros que foram atualizados pelo comando.

Já para o custo de criação de um índice I, definimos a seguinte aproximação:

$$CC_I = 2P + cR \log R$$

Os parâmetros possuem o mesmo significado definido para a fórmula de estimativa do custo de atualização de um índice. Supomos uma política de criação de índices em que todas as páginas da tabela são lidas, ordenadas e então o índice é criado de uma forma *bottom-up*. O primeiro termo da fórmula leva em conta o custo de E/S de ler todas as páginas da tabela e de escrever todas as páginas do índice (para índices hipotéticos, a mesma quantidade estimada). Já o segundo termo estima o custo de ordenar todas as tuplas da tabela em memória.

É importante frisar que estas equações para calcular os custos de atualização e de criação de índices podem ser refinadas. Isto pode exigir um estudo mais detalhado do modelo de custos do SGBD e, até mesmo, a sua extensão. Não houve tempo hábil para investigar o impacto do uso de diferentes funções de estimativa dos custos de atualização e de criação de índices.

4.5.2 Resultados Experimentais

Discutimos, nesta seção, alguns resultados obtidos com a implementação do *Agente de Benefícios* no SGBD de código aberto PostgreSQL. Para simularmos uma situação realista de uso do SGBD, utilizamos em todos os testes o *toolkit* DBT-2 (veja a sua descrição na seção 4.2).

O SGBD possui diversos parâmetros de inicialização que regulam as suas características de alocação de memória, execução de *checkpoints*, coleta de estatísticas, entre outros. O objetivo de nosso trabalho não é fazer a sintonia destes parâmetros de inicialização, mas sim verificar como o agente implementado consegue melhorar o projeto de índices do banco de dados. Assim, fizemos uma configuração padrão para os parâmetros de inicialização do sistema e utilizamos esta mesma configuração em todos os

testes realizados. A Tabela 4.5 mostra os parâmetros aos quais atribuímos valores distintos dos valores sugeridos pela instalação do SGBD. É importante notar que o uso de valores fixos para, por exemplo, o tamanho da área de memória utilizada pelo sistema e seu intervalo de *checkpoints* tem uma conseqüência desejável sobre os experimentos. À medida que submetemos o sistema ao processamento de bases maiores de dados e de mais transações, este efetivamente passa a ter de lidar com uma carga mais intensa, dado que mantemos os recursos disponíveis fixos. Assim, podemos observar como o agente se comporta quando a intensidade da carga sobre o sistema varia.

Nome do Parâmetro	**Valor**	**Significado**
tcpip_socket	true	Permitidas conexões TCP/IP
max_connections	50	50 conexões no máximo ao SGBD
shared_buffers	30720	240MB de *buffers* compartilhados
wal_buffers	32	256KB de *buffers* para *log*
checkpoint_segments	30	480MB de disco para *checkpoints*
checkpoint_timeout	2100	Forçar *checkpoint* após 35 min
checkpoint_warning	1800	Avisar sobre *checkpoints* em menos de 30 min
log_min_messages	warning	Mensagens que vão para o *log* de depuração
log_pid	true	Depurar números de processos
log_statement	true	Depurar comandos submetidos
log_parser_stats	true	Depurar estatísticas do *parser*
log_planner_stats	true	Depurar estatísticas do otimizador
log_executor_stats	true	Depurar estatísticas do executor
stats_command_string	true	Colher estatísticas sobre comandos
stats_block_level	true	Colher estatísticas sobre blocos
stats_row_level	true	Colher estatísticas sobre tuplas

Tabela 4.5: Valores dos parâmetros de inicialização do PostgreSQL

Conforme vimos na seção 4.2, os testes realizados com o *toolkit* DBT-2 simulam a execução de uma carga transacional com múltiplos usuários no SGBD. O *toolkit* já vem com um conjunto de índices sugeridos para o melhor processamento da carga de trabalho. Alguns destes índices são criados como conseqüência da criação de chaves primárias. Para avaliarmos a contribuição trazida pelo nosso agente à vazão observada nos testes, estabelecemos três cenários:

1. Execução da carga de trabalho em uma base sem índices e com o agente desligado: esperamos, a menos de pequenas variações devidas à freqüência de submissão das transações, que este seja o cenário com a menor vazão observada, uma vez que todas as consultas serão

processadas através de varreduras seqüenciais nas tabelas subjacentes. Removemos todos os índices, inclusive os referentes a chaves primárias.

2. Execução da carga de trabalho em uma base sem índices e com o agente ligado: neste cenário observamos a eficácia do agente em encontrar e materializar automaticamente índices que podem trazer benefícios para a execução da carga de trabalho. Conforme vimos na seção 4.5.1, não tomamos ações de remoção de índices reais na nossa implementação.

3. Execução da carga de trabalho em uma base com os índices sugeridos pelo *toolkit* e com o agente desligado: esperamos, a menos de pequenas variações devidas à freqüência de submissão das transações, que este seja o cenário com a maior vazão observada, uma vez que estamos utilizando os índices recomendados pelos implementadores do *toolkit* e o sistema não executa o agente.

É importante observar que quanto mais próxima a vazão do segundo cenário ficar da observada no terceiro cenário, tanto melhor terá sido o trabalho executado pelo agente.

Existem, ainda, dois parâmetros que modificam as características de cada teste: a quantidade de armazéns utilizados e o tempo de duração. Lembramos que a cada armazém utilizado são criados dez novos terminais. Estes terminais são *threads* executadas no servidor. Além dos terminais, também são necessárias *threads* para gerenciar as conexões com o SGBD, que podem ser especificadas separadamente. Após alguma experimentação com os parâmetros de número de conexões e de quantidade de armazéns, a maior escalabilidade que conseguimos no ambiente disponível foi de até quatro armazéns utilizando dez conexões com o SGBD. Quando utilizamos cinco armazéns ou um número maior de conexões, a quantidade de processos simultâneos no servidor se torna tão grande que o próprio *toolkit* apresenta falhas irrecuperáveis em suas conexões entre processos. É interessante reparar que uma base com quatro armazéns oferece uma quantidade de dados próxima da de diversas aplicações reais (ver seção 4.2).

Nos testes que realizamos, variamos a quantidade de armazéns utilizados e mantivemos o parâmetro de número de conexões ao SGBD fixo. Conseguimos, desta forma, simular um aumento real da carga canalizada pelas dez conexões utilizadas.

Variamos, ainda, a duração dos testes, realizando experimentos com trinta, sessenta e noventa minutos. Esta variação é interessante para observarmos o tempo necessário para que o agente aprenda sobre a carga de

trabalho e crie um bom projeto de índices para o sistema. Discutimos a seguir as vazões encontradas para estes tempos de duração quando variamos o número de armazéns utilizados e os cenários de testes.

Relação entre Vazão e Tempo de Duração dos Testes

A Figura 4.16 mostra o gráfico de vazão por quantidade de armazéns para os testes com tempo de duração igual a 30 minutos. A vazão é medida em termos de transações do tipo *Novo Pedido* (*New-Order*) executadas por minuto. Apresentamos as vazões observadas para os três cenários de teste discutidos anteriormente.

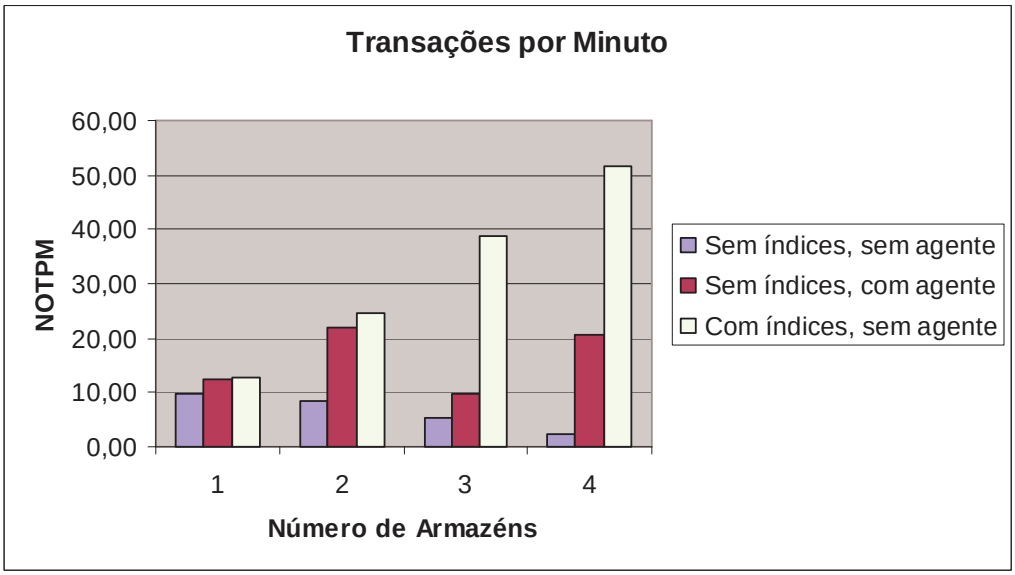

Figura 4.16: Vazão observada com testes de 30 minutos

É interessante observar que o primeiro cenário traçado, de execução do sistema em uma base sem índices, tem uma escalabilidade ruim. À medida que aumentamos o tamanho da base de dados, incluindo mais armazéns, a vazão observada diminui sensivelmente. Isto ocorre pois todas as consultas são processadas com varreduras seqüenciais, o que implica que efetivamente mais tempo é gasto para processar cada consulta à medida que a base cresce. Com uma quantidade maior de terminais submetendo consultas, começamos a ter problemas de contenção de dados.

Já o terceiro cenário traçado, em que criamos os índices sugeridos pelo *toolkit*, apresenta boas características de escalabilidade. À medida que a base cresce e temos mais terminais interessados em acessar os dados, mais transações são executadas por minuto. Para diversas consultas, o

uso de índices permite que o tempo de resposta permaneça praticamente constante mesmo quando ocorre crescimento da base. Quando se associa isto ao aumento no número de terminais (e, portanto, na quantidade de consultas submetidas por unidade de tempo), temos um aumento na vazão observada.

Repare que, como esperado, a vazão obtida a partir de uma base sem índices e com o agente ligado se situou entre as vazões obtidas para os outros dois cenários. O sistema passa, nestes testes, por duas grandes fases. Na primeira fase, o agente observa os comandos SQL que são submetidos ao SGBD e escolhe índices adequados. Durante esta fase, a vazão se aproxima da obtida em nosso primeiro cenário. Já na segunda fase, o agente consegue materializar índices que aceleram o processamento das consultas. Com estes índices presentes na base, há uma melhora considerável do tempo de resposta das transações e há uma tendência da vazão se situar em patamares mais próximos do terceiro cenário.

Tivemos, entretanto, um comportamento menos do que ideal na escalabilidade observada para o cenário com o agente. Para três e quatro armazéns, foi possível verificar um problema de bloqueios que diminuiu a agilidade do agente na criação de novos índices. No PostgreSQL, não há a funcionalidade de criação de índices *on-line* (isto é possível em alguns SGBDs, como o Oracle por exemplo [45]). Assim, a criação de um novo índice exige a obtenção de um bloqueio sobre a tabela que impede o seu acesso concorrente por transações de atualização.

As transações que realizam atualizações, no PostgreSQL, obtêm um bloqueio intencional de escrita sobre a tabela e bloqueios exclusivos sobre as tuplas atualizadas. É possível, portanto, termos mais de um escritor atualizando a mesma tabela simultaneamente, desde que estejam interessados em tuplas distintas. Isto implica que novos escritores podem obter bloqueios intencionais sobre a tabela mesmo que escritores que já a estavam atualizando não tenham ainda liberado os seus bloqueios intencionais. Este comportamento pode fazer com que uma transação interessada em criar um índice passe por uma longa espera até um momento em que nenhum escritor possua um bloqueio intencional sobre a tabela.

Esta situação ocorreu nos testes com o agente ligado tanto com três como com quatro armazéns. O agente sempre passou por esperas antes de conseguir criar algum índice real. Estas esperas foram causadas por conflitos de bloqueios sobre as tabelas em que novos índices iam ser criados. Isto significou que o sistema ficou por menos tempo com os índices adequados para o seu funcionamento criados, o que afetou negativamente a vazão.

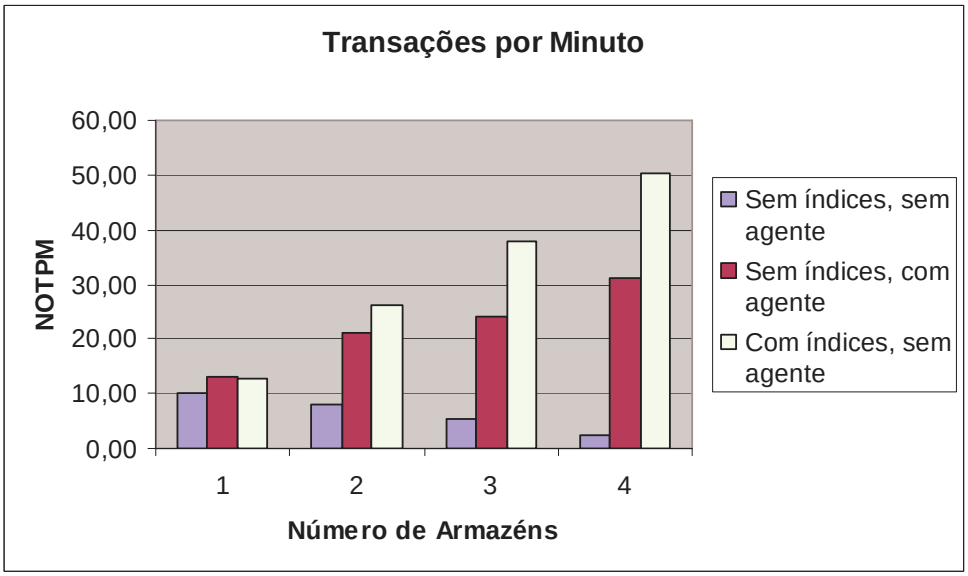

Figura 4.17: Vazão observada com testes de 60 minutos

Para validarmos este comportamento, podemos observar as vazões encontradas nos testes com durações maiores, de 60 e de 90 minutos, exibidas nas Figuras 4.17 e 4.18. Nestes testes observamos maior escalabilidade para a vazão que encontramos no cenário sem índices e com o agente ligado do que nos testes com duração de 30 minutos. A vazão do cenário com o agente sempre se situa, como esperado, entre das vazões do cenário sem índices e do cenário com os índices do *toolkit*. O agente consegue superar a fase de criação de índices e o sistema opera durante algum tempo com índices adequados. Isto faz com que a vazão seja tanto mais próxima da observada para o terceiro cenário (com índices do *toolkit*) quanto maior foi o tempo de duração do teste e, portanto, o tempo de operação do sistema com um conjunto de índices satisfatório. Esta tendência pode ser observada nas Figuras pela maior proximidade entre as vazões do segundo e do terceiro cenários no experimento de 90 minutos do que no experimento de 60 minutos.

Uma conclusão que podemos derivar deste experimento é a de que decisões de projeto físico, como a criação de índices, tendem a trazer um maior benefício no médio e longo prazos. No nosso experimento, o uso do agente se revelou mais interessante nos testes de duração mais longa, levando a vazão para valores crescentemente mais próximos da vazão observada quando utilizamos um conjunto de índices previamente criado e sugerido pelos fornecedores da aplicação.

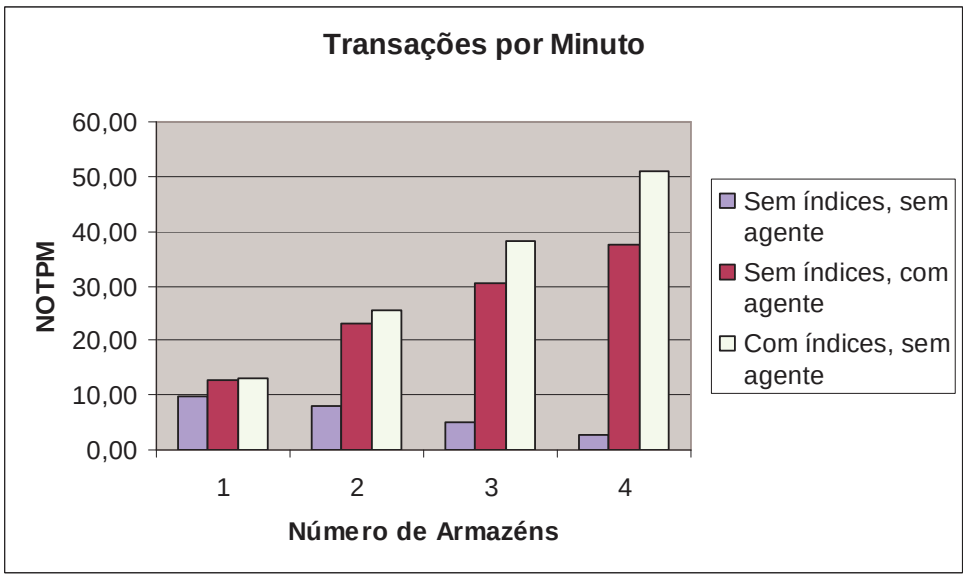

Figura 4.18: Vazão observada com testes de 90 minutos

Qualidade dos Índices Selecionados

Fazemos nesta subseção uma avaliação da qualidade dos índices escolhidos pelo agente nos experimentos anteriores. Em primeiro lugar, foi possível verificar que há uma relativa estabilidade nos índices escolhidos pelo agente, mesmo quando variamos o tamanho da base de dados. Há um conjunto de dez índices que foram criados pelo agente repetidamente nos experimentos realizados.

Em alguns experimentos, além destes dez índices básicos, foram criados um ou dois outros índices. Estes índices adicionais foram escolhidos por variações nas características de custeamento dos índices por parte do otimizador quando variamos o tamanho da base. É importante notar que os comandos da carga de trabalho foram mantidos idênticos entre os experimentos, bem como a heurística de enumeração de índices empregada pelo agente.

A Tabela 4.6 mostra os dez índices repetidamente escolhidos pelo agente nos experimentos e os compara com os índices sugeridos pelo *toolkit* DBT-2 para o processamento da carga de trabalho. Repare que muitos dos índices são extremamente semelhantes, tendo como diferença apenas a ordem empregada para as colunas. Verificamos que isto se deve ao fato de que, dentro de cada grupo de colunas detectado pelo agente (por exemplo, EQ ou RANGE; ver seção 4.5.1), a ordem das colunas é estabelecida de acordo com a posição em que as colunas aparecem na tabela base. Já

[4]Estes índices foram escolhidos somente para bases com mais do que um armazém.

Índices criados pelo Agente	**Índices criados pelo *toolkit***
customer(c_id, c_d_id, c_w_id)	customer(c_w_id, c_d_id, c_id)
customer(c_d_id, c_w_id, c_last, c_first)	customer(c_w_id, c_d_id, c_last, c_first, c_id)
district(d_w_id)	district(d_w_id, d_id)
item(i_id)	item(i_id)
new_order(no_o_id, no_d_id, no_w_id)	new_order(no_w_id, no_d_id, no_o_id)
new_order(no_w_id)[4]	não há índice correspondente
order_line(ol_o_id, ol_d_id, ol_w_id)	order_line(ol_w_id, ol_d_id, ol_o_id, ol_number)
orders(o_id, o_d_id, o_w_id)	orders(o_w_id, o_d_id, o_id)
orders(o_d_id, o_w_id, o_c_id, o_id)	orders(o_w_id, o_d_id, o_c_id)
stock(s_i_id, s_w_id)[4]	stock(s_w_id, s_i_id, s_quantity)
tabela warehouse não foi indexada	warehouse(w_id)
tabela history não foi indexada	tabela history não foi indexada

Tabela 4.6: Índices sempre criados pelo agente e pelo *toolkit*

nos índices criados pelo *toolkit*, as colunas são dispostas em ordem de seletividade. Uma alteração que poderia ser investigada na heurística de escolha de candidatos do agente é a de procurar ordenar as colunas dentro de cada grupo pela seletividade estimada pelo otimizador e não de forma posicional. Isto pode fazer com que índices sejam melhor aproveitados em consultas para as quais eles não foram originalmente projetados. Em nossa carga de trabalho experimental, entretanto, a ordenação das colunas nos índices não trouxe qualquer implicação adicional.

Além desta diferenças, também podemos notar na Tabela 4.6 que existem um índice escolhido pelo agente que não possui equivalente no *toolkit* e um índice utilizado pelo *toolkit* que não foi detectado pelo agente. A decisão do agente de não criar índices sobre a tabela *warehouse* reflete a preocupação do agente com índices que tragam vantagens de desempenho para a carga de trabalho. A tabela *warehouse* registra os armazéns e, em nossos experimentos, possui no máximo quatro tuplas. Sua indexação não traz ganhos de desempenho do ponto de vista do otimizador do sistema. O índice criado pelo *toolkit* nesta tabela é uma implementação de uma restrição de chave primária.

Outra situação interessante que podemos observar é a decisão do agente de criar um índice sobre a coluna `no_w_id` da tabela `new_order`. Existe uma consulta na carga de trabalho para a qual o otimizador estima grande benefício com o uso deste índice quando o tamanho da tabela aumenta. Assim, mesmo tendo esta consulta uma freqüência baixa na carga de trabalho, a amplitude do benefício trazido justifica a criação do índice.

Para termos uma medida que possibilitasse a comparação da qualidade

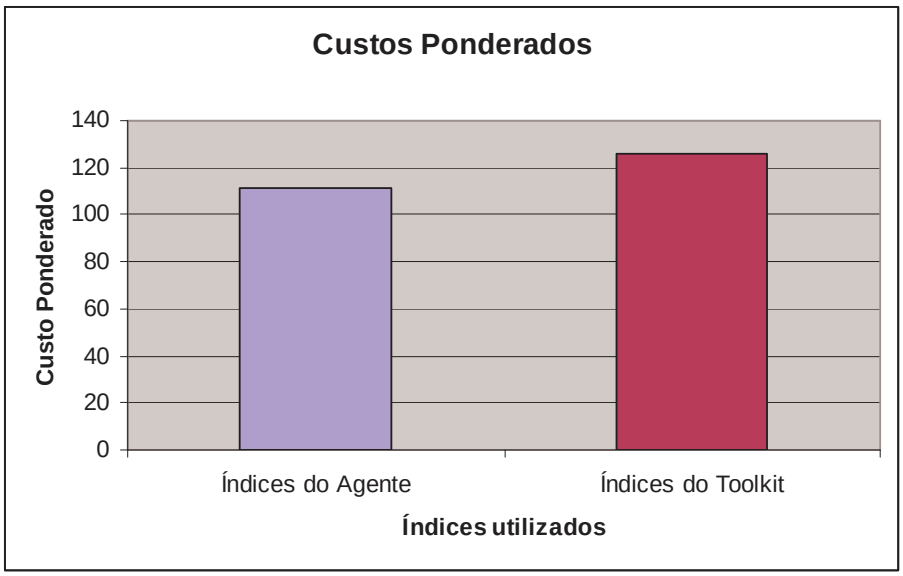

Figura 4.19: Custos ponderados da carga de trabalho

dos índices encontrados pelo agente e sugeridos pelo *toolkit*, utilizamos o otimizador do sistema para estimar o custo de execução de cada comando da carga de trabalho nas duas configurações de índices. Em seguida, ponderamos estes custos pelas freqüências esperadas de cada comando na carga de trabalho, obtendo um custo ponderado para cada comando. Mostramos, na Figura 4.19, a soma dos custos ponderados obtidos para toda a carga de trabalho em uma base de dados com quatro armazéns.

Obtivemos um custo ponderado para os índices utilizados pelo agente 11,64% mais baixo do que o encontrado para os índices utilizados pelo *toolkit*. O índice criado pelo agente na coluna `no_w_id` da tabela `new_order` tem um papel fundamental nesta diferença de custos em favor do projeto encontrado pelo agente. Sem este índice, o projeto de índices encontrado pelo agente teria custos 4,79% mais altos do que o sugerido pelo *toolkit*.

De acordo com as estimativas do otimizador, portanto, se o tempo de duração dos testes fosse suficientemente estendido, o projeto de índices encontrado pelo agente em algum momento se mostraria superior ao sugerido pelo *toolkit* em termos de vazão.

Análise de Comandos Vistos pelo Agente

Em todos os testes realizados, mantivemos a capacidade da fila de mensagens definida no sistema operacional fixa. A fila foi definida de forma a conseguir acomodar 50 mensagens geradas pelos processos `postgres`

para o agente. Apesar de o tamanho da fila no sistema operacional ser definido em *bytes*, é possível calcular o número de mensagens acomodadas na fila pois utilizamos mensagens de tamanho fixo e igual a 2KB em nossa implementação.

Tivemos um comportamento interessante com o uso de uma fila de mensagens de tamanho fixo. À medida que a carga submetida ao sistema aumenta, um maior número de comandos SQL é gerado em um intervalo curto de tempo para o agente. A capacidade da fila é insuficiente para acomodar todos os comandos gerados pelos processos do SGBD. Em nossa implementação, caso o processo do SGBD não seja capaz de enviar o comando para a fila de mensagens do agente, o processo segue sua execução normal e o comando não é enviado.

Isto significa que nem todos os comandos processados pelo SGBD são vistos pelo agente. Na verdade, o agente vê apenas uma parcela destes comandos. Esta parcela é a que consegue ser depositada na fila de mensagens para comunicação com o agente. Mostramos o percentual de comandos vistos pelo agente nos diversos experimentos realizados na Figura 4.20.

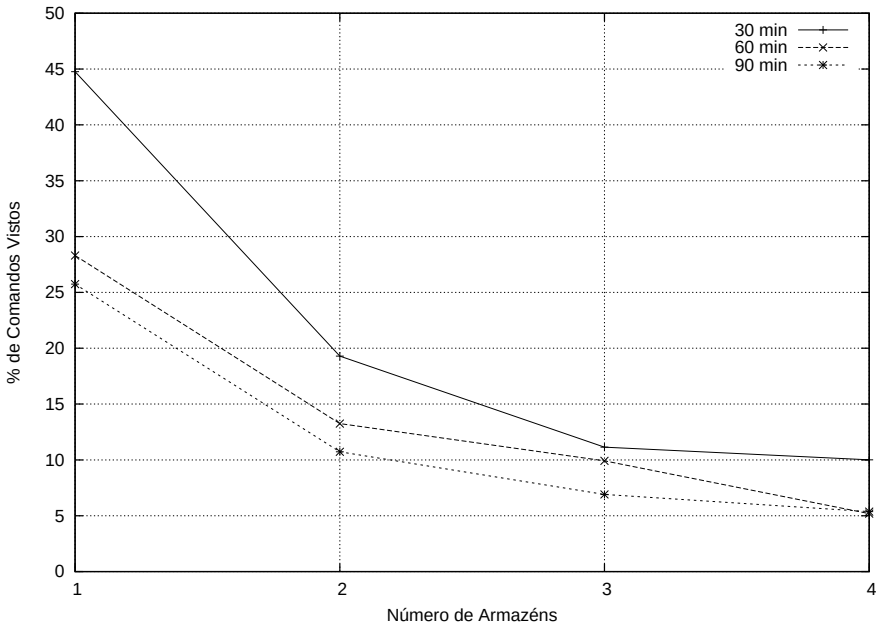

Figura 4.20: Comandos vistos pelo agente nos testes

Repare que, tanto nos testes de duração mais curta quanto nos de duração mais longa, o percentual de comandos que o agente consegue receber e processar diminui à medida que a carga sobre o sistema aumenta. Para os testes acima de 60 minutos, houve uma tendência de estabilização com o agente processando em torno de 5% dos comandos submetidos ao SGBD.

Apesar da queda no percentual de comandos vistos, conforme discutido anteriormente, o agente encontrou índices de boa qualidade em todos os experimentos realizados. Concluímos daí que, mesmo tendo visto percentualmente menos comandos, o que foi visto pelo agente foi significativo o suficiente para que este tomasse decisões acertadas de criação de índices. Associamos o fato de mesmo uma pequena amostra dos comandos processados ter sido significativa à característica da aplicação empregada de sempre submeter as mesmas transações ao SGBD, com comandos SQL explicitamente codificados.

Fizemos, ainda, um teste adicional tornando a colocação das mensagens na fila do agente uma operação síncrona. O objetivo deste teste era o de fazer com que todos os comandos processados chegassem ao agente. O resultado deste teste foi um impasse, em que todos os processos (do SGBD e do agente) ficaram se esperando mutuamente.

Para compreendermos esta situação, lembramos que quando o agente decide criar um índice real é necessária a obtenção de um bloqueio que restringe escritores sobre a tabela base. Se temos transações sendo processadas pelos demais processos do SGBD que possuem bloqueios para escrita sobre a tabela, o agente deve esperar até que as transações terminem para que consiga continuar com seu processamento. Entretanto, como o envio de comandos das transações para o agente é síncrono, as transações precisam esperar que o agente receba os seus comandos para conseguirem, então, terminar. Está criado o impasse.

Esta situação não ocorre quando o envio de mensagens é assíncrono. Caso não haja espaço na fila de mensagens para envio de um comando ao agente, o processo `postgres` correspondente descarta o envio do comando corrente e continua com a sua transação. Isto permite que o agente prossiga com o seu processamento e consuma novas mensagens da fila, possibilitando que novos comandos sejam recebidos.

4.6 Comentários Finais

Neste capítulo, apresentamos a implementação da arquitetura com um único agente para auto-sintonia de índices em SGBDs relacionais discutida no Capítulo 3. Inicialmente, mostramos alguns conceitos da arquitetura do SGBD PostgreSQL que são necessários para a compreensão da nossa implementação. O estudo do modelo de comunicação cliente-servidor empregado pelo SGBD e do seu mecanismo de processamento de consultas foram im-

portantes para a integração do agente com o SGBD e para a inclusão de notificações para o agente no código do sistema.

Em seguida, apresentamos as bases de dados e cargas de trabalho que foram utilizadas para os testes da nossa implementação. Discutimos uma base de dados de vendas simples e mostramos o *toolkit* OSDL DBT-2, que implementa uma carga transacional comparável às encontradas em ambientes reais de uso de SGBDs.

A implementação de índices hipotéticos foi um pré-requisito para o uso de nossa arquitetura de agentes no PostgreSQL. Discutimos quais mudanças conduzimos no servidor para incluir este conceito. Apresentamos, ainda, resultados sobre o desempenho e a qualidade de índices hipotéticos. A criação de índices hipotéticos possui tempo constante mesmo para tabelas de tamanhos crescentes. As estimativas realizadas pelo otimizador ao utilizar este tipo de índice são, em geral, muito próximas dos valores encontrados quando criamos os índices reais correspondentes.

A arquitetura com um único agente foi implementada através do uso de um *framework* para construção de agentes em camadas. Debatemos quais foram as extensões necessárias ao *framework* em cada camada para obtermos um agente capaz de avaliar e criar índices automaticamente. Mostramos como este agente foi integrado ao SGBD PostgreSQL, respeitando o seu modelo de comunicação que cria múltiplos processos no servidor. Apresentamos, ainda, uma adaptação de uma heurística da literatura para escolha de índices candidatos e equações utilizadas para o cálculo de fatores utilizados na heurística de benefícios discutida no Capítulo 3.

Por fim, discutimos diversos resultados experimentais obtidos com a implementação do agente. Utilizamos a carga transacional implementada pelo *toolkit* DBT-2 para avaliar o benefício em termos de vazão que o agente pode trazer através da criação de novos índices. Observamos que o uso do agente se mostra crescentemente mais benéfico para a vazão em experimentos de maior duração. Isto revela que as decisões de criação de índices tendem a trazer mais benefícios no médio e longo prazos do que benefícios imediatos.

Avaliamos a qualidade dos índices escolhidos pelo agente e os comparamos com índices sugeridos pelo próprio *toolkit* DBT-2 para o processamento da carga de trabalho. Pelas estimativas realizadas pelo otimizador, os índices criados pelo agente apresentam um custo ponderado para processamento da carga de trabalho menor do que o custo ponderado dos índices sugeridos pelo *toolkit*.

Avaliamos, ainda, a quantidade dos comandos que foram vistos pelo

agente durante os testes. Percebemos que, em testes com maior duração e carga mais intensa, o agente consegue processar percentualmente menos comandos da carga de trabalho. Mesmo assim, a qualidade dos índices encontrados foi tão boa quanto em situações de menor duração e carga menos intensa. Como a carga de trabalho é implementada através de aplicações que utilizam SQLs fixos, repetidos diversas vezes no SGBD, a quantidade percentualmente menor de comandos obtidos permaneceu significativa para a escolha de índices para a carga de trabalho.

No próximo capítulo, mostramos as conclusões desta dissertação e ressaltamos algumas direções em que o trabalho pode ser estendido.