Pontifícia Universidade Católica
do Rio de Janeiro

Antony Seabra de Medeiros

Particionamento como ação de sintonia fina em Bancos de Dados Relacionais

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática da PUC-Rio

Orientador: Prof. Sérgio Lifschitz

Rio de Janeiro
Abril de 2017

Antony Seabra de Medeiros

Particionamento como ação de sintonia fina em Bancos de Dados Relacionais

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-graduação em Informática do Departamento de Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Sérgio Lifschitz
Orientador
Departamento de Informática – PUC-Rio

Profa. Ana Carolina Brito de Almeida
UERJ

Prof. Rogério Luís de Carvalho Costa
PUC-Rio

Prof. Márcio de Silveira Carvalho
Coordenador Setorial do Centro
Técnico Científico – PUC-Rio

Rio de Janeiro, 06 de Abril de 2017

Antony Seabra de Medeiros

Cursou Tecnólogo em Processamento de Dados pela Faculdade Anglo-Americano e graduou-se em Matemática pela Universidade Federal Fluminense. Cursou pós-graduações *latu-sensu* em Engenharia de Software (PUC-MG), Bancos de Dados (UGF-RJ) e Tecnologia para Internet (COPPE/UFRJ). Trabalhou durante diversos anos com Bancos de Dados Relacionais e Orientados a Objeto, tendo como área de maior interesse a sintonia fina de bancos de dados. Atualmente é analista de suporte no BNDES, onde exerce função executiva na área de TI.

Ficha Catalográfica

Medeiros, Antony Seabra de

Particionamento como ação de sintonia fina em bancos de dados relacionais / Antony Seabra de Medeiros ; orientador: Sérgio Lifschitz. – 2017.
156 f. : il. ; 30 cm

Dissertação (mestrado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2017.
Inclui bibliografia

1. Informática – Teses. 2. Bancos de dados relacionais. 3. SGBD. 4. Sintonia fina. 5. Particionamento. 6. PostgreSQL. I. Lifschitz, Sérgio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

Em primeiro lugar a Deus. Suas bênçãos sem fim me fizeram chegar até aqui.

A minha mãe (*in memoriam*), Estela, que esteve comigo durante toda a vida até o primeiro período neste programa, me incentivando sempre a seguir adiante.

Ao meu pai, Adilson, que me ensinou a programar *Basic* num *Commodore 64* quando eu ainda tinha 12 anos. À minha irmã, Annie, que compartilha comigo o privilégio de ter nascido filho de pais professores. À minha esposa, Luciana, e à toda a minha família pelo suporte, amor e carinho e, em especial, a minha filha, Mariana, que me dá todo dia o sorriso mais lindo do mundo!

Ao meu orientador Prof. Sérgio Lifschitz que soube me trazer a este desfecho com maestria. Suas aulas de excelência em bancos de dados, sua habilidade na condução da pesquisa, sua compreensão diante de minhas limitações e sua amizade durante todo o percurso me fazem levá-lo como referência para minha vida acadêmica e profissional.

A Profa. Ana Carolina Brito de Almeida pelos encontros e orientações durante a caminhada. Ao Prof. Rogério Costa por suas contribuições decisivas durante as defesas. Ao Prof. Marcos Vianna Villas pelas argumentações e sugestões na apresentação de evolução da pesquisa. Muito obrigado por todas as perguntas.

Ao Prof. Edward Hermann Haeusler pelas aulas de Computabilidade. Ao Prof. Ruy Luiz Milidiú pelos cursos de Aprendizado de Máquina. Ao Prof. Antonio Luiz Furtado pelas histórias e ensinamentos maravilhosos.

Aos meus colegas do grupo de pesquisa BioBD, Alain Domínguez, Alejandro Menes, Ema Molina, Julio Omar, Liester Cruz, Otávio Freitas e Rafael Pereira de Oliveira, muito obrigado pelas discussões em tantas apresentações.

A PUC-Rio pelo apoio financeiro. E a todos os demais professores e funcionários do Departamento de Informática da PUC-Rio, em especial à Regina Zanon e ao Cosme Pereira Leal, meu muito obrigado.

Resumo

Medeiros, Antony; Lifschitz, Sérgio. **Particionamento como ação de sintonia fina em Bancos de Dados Relacionais**. Rio de Janeiro, 2017. 156p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

As principais estratégias de sintonia fina utilizadas por administradores de bancos de dados relacionais são a construção de estruturas de acesso, como índices, índices parciais e visões materializadas, e técnicas como desnormalização e reescrita de consultas. Estas técnicas e estruturas de acesso, juntas ou separadas, podem melhorar o desempenho das consultas submetidas ao banco de dados. O particionamento de tabelas do banco de dados, técnica tradicionalmente utilizada para distribuição de dados, também possui potencial para sintonia fina, pois permite que a varredura das tabelas seja realizada somente nas partições que satisfazem os predicados das consultas. Mesmo em consultas com predicados de seletividade alta, cujos planos de execução frequentemente utilizam índices, o particionamento pode oferecer um benefício ainda maior. Esta dissertação de mestrado propõe avaliar o particionamento como ação de sintonia fina de bancos de dados relacionais e, para tanto, desenvolve heurísticas para seleção de estratégias de particionamento e avaliação do seu benefício. Uma avaliação da qualidade dos resultados obtidos é realizada através de experimentos com um *benchmark* padrão para este tipo de pesquisa e mostramos que, em certos casos, é vantajoso particionar dados.

Palavras-chave

bancos de dados relacionais; SGBD; sintonia fina; particionamento

Abstract

Medeiros, Antony; Lifschitz, Sérgio (Advisor). **Partitioning as a tuning action for Relational Databases**. Rio de Janeiro, 2017. 156p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The main fine tuning strategies used by relational database administrators are the construction of access structures, such as indexes, partial indexes and materialized views, and techniques such as denormalization and query rewriting. These techniques and access structures, together or separately, can improve the performance of queries submitted to the database. Database partitioning, a technique traditionally used for data distribution, has also the potential for fine tuning, since it allows the scanning of tables to be performed only on partitions that satisfy query predicates. Even in queries with high selectivity predicates, whose execution plans often use indexes, partitioning can offer even greater benefit. This dissertation proposes to evaluate the partitioning as a fine tuning action of relational databases and, for that, develops heuristics for selection of partitioning strategies and evaluation of its benefit. An evaluation of the quality of the results obtained is carried out through experiments with a standard benchmark for this type of research and we have shown that, in certain cases, it is advantageous to partition data.

keywords

relational databases; DBMS; fine tuning; partitioning

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

A crescente demanda por soluções de tecnologia que aumentem a disponibilidade e o desempenho de aplicações computacionais é resultado da forte expansão do mercado de consumo de informações. Seja nas redes sociais ou nas comunidades científicas, pelas aplicações transacionais ou pelas ferramentas de suporte a decisão, usuários elevam cada vez mais a expectativa de acesso rápido à informação e de atualização da mesma.

Fabricantes e fornecedores de hardware e software têm concentrado seus esforços na implementação de soluções que endereçem estas necessidades, incluindo infraestrutura de redes de alta velocidade, meios de armazenamento redundantes e tolerantes a falhas, sistemas operacionais robustos e sistemas de bancos de dados que suportem grandes volumes de dados e processamento em larga escala.

Especificamente em Sistemas Gerenciadores de Bancos de Dados (SGBDs), alto desempenho no acesso e na atualização dos dados é preponderante, pois contribui de forma relevante para que a expectativa citada seja satisfeita. Neste contexto, a sintonia fina de bancos de dados ganha cada vez mais importância. Sendo uma das atividades principais de um administrador de banco de dados (DBA – *Database Administrator*), é realizada através de técnicas que objetivam aumentar o desempenho das consultas submetidas ao banco de dados.

A sintonia fina de bancos de dados relacionais utiliza tradicionalmente a manutenção de estruturas de acesso que permitam melhor desempenho na busca pelos resultados das consultas. Assim, a construção de índices, índices parciais e visões materializadas são estratégias clássicas de sintonia fina. Neste contexto há outra estratégia de sintonia fina com grande potencial de impacto no desempenho das consultas: o particionamento (Madden, 2011). O particionamento das tabelas do banco de dados pode reduzir o espaço de busca das consultas e melhorar os tempos de resposta das mesmas.

1.1 Motivação da Pesquisa

Quando uma consulta a um banco de dados relacional utiliza um predicado com seletividade baixa, o otimizador do SGBD escolhe varrer a tabela completa (*sequential scan*) ao invés de usar um índice naquele predicado (*index scan*). Nestes casos, o particionamento surge como alternativa para a sintonia fina, na medida em que permite que as varreduras sejam realizadas em um volume menor de dados. Assim, a motivação original desta pesquisa é endereçar o problema da seletividade baixa de predicados que levam à varreduras completas de tabelas.

Além disso, nos casos em que a seletividade é alta o suficiente para que o otimizador utilize um índice, o particionamento pode ser uma opção que ofereça um benefício ainda maior.

Sendo assim, *como uma ação de particionamento influencia o otimizador na geração de planos de execução*?

1.2 Objetivo da Pesquisa

A consequência principal do particionamento horizontal é que a varredura completa sobre uma partição da tabela é realizada sobre um número menor de linhas da tabela, enquanto a consequência principal do particionamento vertical é que a varredura completa é realizada sobre um número menor de colunas da tabela.

Ambas tem o potencial de oferecer benefício: no particionamento horizontal este benefício está diretamente relacionado à seletividade dos predicados utilizados nas consultas da carga de trabalho, enquanto no particionamento vertical, o benefício está diretamente associado às colunas necessárias para satisfazer as consultas da carga de trabalho e ao arranjo de colunas em cada partição.

O objetivo principal desta dissertação é mostrar que o particionamento horizontal pode ser utilizado como estratégia *primeira* de sintonia fina, isto é, uma estratégia que tem o mesmo potencial de benefício quanto todas as outras.

Os objetivos específicos desta dissertação são:

- Mostrar como os planos de execução selecionados pelo otimizador são influenciados pelo particionamento horizontal das tabelas;

- Identificar os casos em que o particionamento horizontal é melhor estratégia de sintonia fina do que índices;

- Mostrar que estes casos existem tanto em cargas de trabalho com predominância de predicados de seletividade baixa (mais frequentes em bancos de dados OLAP), como em cargas de trabalho com predominância de predicados de seletividade alta (mais frequentes em bancos de dados OLTP);

- Desenvolver heurísticas para seleção da estratégia de particionamento horizontal;

- Realizar testes experimentais com TPC-H e avaliar os resultados obtidos.

1.3 Restrições

Faz-se necessário impor restrições porque o particionamento apresenta uma diversidade de opções conceituais e de implementação. As restrições impostas estão pontuadas abaixo:

- Particionamento lógico

O particionamento horizontal será lógico, isto é, partições lógicas são geradas e alocadas dentro da mesma partição física.

- Chave de particionamento

A chave do particionamento horizontal será limitada a uma coluna. Todas as chaves compostas candidatas a chave de particionamento são excluídas do processo de seleção na heurística correspondente.

- Parâmetros de controle

Dois parâmetros de entrada são fornecidos, a saber, o número máximo de partições e o espaço disponível em cada área de armazenamento de tabelas. Enquanto limita o volume de dados em cada área, relevante mesmo com a utilização do particionamento lógico, prepara a proposta para ser adaptada ao particionamento físico.

1.4 Organização do trabalho

No próximo capítulo são apresentados os conceitos e fundamentos básicos necessários ao entendimento da pesquisa. No capítulo 3 são apresentados os trabalhos relacionados ao tema desta dissertação. No capítulo 4 são apresentadas heurísticas para sintonia fina de bancos de dados e o desenvolvimento de extensões para contemplar o particionamento horizontal de tabelas. No capítulo 5 são apresentados 3 cenários distintos e os testes realizados. No mesmo capítulo é mostrado o impacto de atualizações em dois destes cenários. No capítulo 6 são apresentadas as conclusões, contribuições da pesquisa e sugestões para trabalhos futuros.

No apêndice A é mostrado o esquema do TPC-H, *benchmark* de referência para realização de todos os exemplos e testes da pesquisa, e coleta de resultados. No apêndice B é apresentado um resumo da heurística HISAI, base para o desenvolvimento desta pesquisa. No apêndice C estão listados os códigos utilizados para implementação do particionamento nos cenários estudados, e no apêndice D estão os planos de execução de todas as consultas que compõem os cenários estudados.

2 Conceitos e fundamentos

Entre as atividades principais desempenhadas por um administrador de banco de dados está a sintonia fina, atividade que consiste em fazer uma aplicação de banco de dados ser executada em um tempo menor. E, para fazer com que um sistema execute mais rápido, o DBA pode ter que alterar a maneira como as aplicações são construídas, as estruturas de dados e os parâmetros do sistema, além da configuração do sistema operacional e do hardware (Shasha, 2002).

2.1 Sintonia fina

Toda vez que o SGBD recebe uma solicitação de consulta, um plano de execução da mesma é gerado pelo seu otimizador de consultas. A sintonia fina das consultas é fortemente apoiada na análise destes planos de execução, porque o otimizador associa custos às diversas operações que compõem os planos de execução possíveis. A diferença entre estes planos de execução é o conjunto de estruturas de acesso que são utilizadas e os tipos de varredura realizados sobre cada estrutura. A utilização ou não de uma estrutura de acesso é uma decisão do otimizador de consultas do SGBD.

Fazer uma aplicação executar em um tempo menor é melhorar seu desempenho. De fato, através da medição dos tempos de resposta das consultas submetidas ao banco de dados e das ferramentas apropriadas para avaliação das estruturas utilizadas por essas consultas, o DBA pode avaliar o desempenho das consultas e a necessidade de melhoria dos tempos medidos.

O desempenho de um banco de dados relacional é impactado por uma série de fatores, entre eles a latência da conexão com o SGBD, a vazão dos pacotes que trafegam entre o cliente e o SGBD e o tempo de resposta de uma consulta ao banco de dados. A sintonia fina refere-se ao tratamento destes tempos de resposta, empregando técnicas que permitem ao administrador do banco de dados melhorá-los (Bruno, 2011).

A sintonia fina de bancos de dados analisa a carga de trabalho e seleciona estratégias para melhorar os tempos de resposta das consultas mais frequentes (Salles, 2004). As estratégias clássicas de *tuning* utilizadas pelos DBAs incluem índices, visões materializadas com reescrita de consultas, tratamento de bloqueios e níveis de isolamento em procedimentos armazenados. Todas essas estratégias podem ser eficientes e pretendem atender a cenários distintos do processamento que garante a execução de uma consulta.

2.1.1 Índices

Índices fazem parte do projeto físico de um banco de dados e são criados com o objetivo de reduzir o tempo necessário para execução das consultas ao banco de dados. Na sua construção, uma ou mais colunas da tabela são utilizadas como chave(s), conforme o exemplo abaixo:

```
CREATE INDEX i_lineitem_orderkey_partkey_suppkey
ON lineitem (l_orderkey, l_partkey, l_suppkey);
```

Para assegurar um bom desempenho à carga de trabalho submetida ao SGBD, torna-se de vital importância escolher um conjunto apropriado de índices. E essa escolha não é simples, considerando que um banco de dados pode conter centenas de tabelas e que cada tabela pode conter dezenas de colunas, fazendo com que o número de combinações possíveis de índices seja muito grande (Monteiro, 2008).

Índices podem ser primários ou secundários. Índices primários contêm a chave primária da tabela enquanto todos os outros são índices secundários (Ramakrishnan, 2002). Índices secundários oferecem um meio secundário para acessar um registro para o qual já existe um acesso primário (Almeida, 2013), pois cada entrada aponta para a entrada correspondente na tabela.

Com a utilização de um índice, o SGBD passa a ter a alternativa de não precisar mais acessar todas as tuplas de uma tabela para verificar uma determinada condição; de outro modo, se existe um índice que organiza as tuplas de acordo com a condição selecionada, o SGBD pode acessar o índice, percorrer as chaves das tuplas que satisfazem a condição e acessar os dados na tabela. Na medida em que o SGBD precisa ler um número menor de páginas em disco, o desempenho das consultas é melhorado substancialmente, pois o custo da consulta depende também deste custo de E/S.

Na figura 2.1 a tabela está fisicamente ordenada pela coluna *name*, que é a chave primária da tabela. Este é o índice primário. Quatro índices secundários são mostrados, um deles ordenado pelas colunas *age* e *sal*, e outro ordenado somente pela coluna *age*. Assim, uma consulta com uma condição baseada na coluna *age* poderá ser satisfeita com o uso do índice secundário nesta coluna. Neste caso, o SGBD percorre somente as tuplas do índice que satisfazem a condição e, para cada uma delas, recupera os registros correspondentes na tabela.

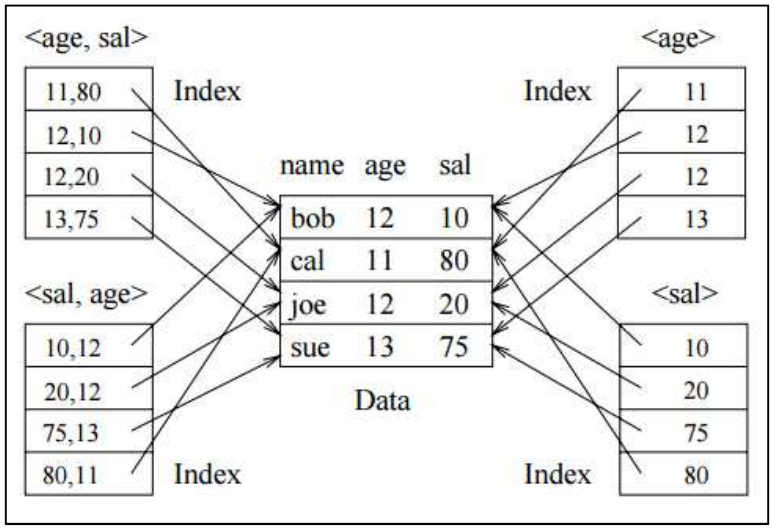

Figura 2.1: Índices primário e secundário (Ramakhrishnan, 2002)

O otimizador do SGBD utiliza informações do próprio banco de dados para determinar os custos associados às estruturas de acesso existentes. Se, para responder uma consulta, o custo de varrer toda uma tabela for menor do que o custo de acessar um índice e depois acessar a tabela para os registros correspondentes, então o otimizador escolhe a primeira opção.

Não somente as informações do banco de dados influenciam nesta decisão, mas também outros fatores como, por exemplo, o quanto uma estrutura está fragmentada. Um índice fragmentado pode degradar o desempenho das operações de busca, principalmente operações de varredura (Morelli, 2006).

2.1.2 Índices parciais

Um índice parcial é um índice criado sobre um subconjunto específico de linhas da tabela. Índices tradicionais são completos, pois tem-se uma entrada no índice para cada linha na tabela. Em índices parciais, somente algumas linhas da tabela possuem entradas no índice (Wu, Madden, 2011).

Para criar um índice parcial, deve-se especificar qual é o subconjunto através de uma cláusula *where*. Qualquer índice que inclui a cláusula *where* no final é considerado um índice parcial. Índices que omitem a cláusula *where* são índices tradicionais.

```
CREATE INDEX pi_lineitem_orderkey_partkey_suppkey
ON lineitem (l_orderkey, l_partkey, l_suppkey)
WHERE l_suppkey BETWEEN 5000 and 7000;
```

Índices parciais são, de fato, estruturas de acesso para sintonia fina com grande potencial, na medida em que podem melhorar o desempenho das consultas que utilizam subconjuntos específicos. Um exemplo típico é uma carga de trabalho com consultas frequentes utilizando uma condição com valores discretos. A alternativa clássica é a criação de um índice completo nesta coluna, mas se somente alguns destes valores discretos ocorrem mais frequentemente nestas consultas, então um índice criado somente para estes valores é mais indicado.

Quando a coluna admite valores contínuos, índices parciais também se aplicam. Além de restringir o índice a valores de maior interesse, índices parciais dão maior acurácia à seletividade dos predicados, especialmente quando possui muitas lacunas. Nestes casos, as estimativas de seletividade podem ser muito imprecisas, diferentemente de quando a distribuição é uniforme. Então, um

índice parcial que exclua essas regiões de lacunas, terá uma estimativa de seletividade bem mais precisa (Stonebraker, 1989).

2.1.3 Visões materializadas

Em bancos de dados relacionais, uma visão é uma tabela cujas linhas não são armazenadas no banco de dados, mas são computadas conforme necessário a partir da sua definição (Ramakrishnan, 2002). Já as visões materializadas são visões cujos dados retornados são fisicamente armazenados no banco de dados.

Tanto para a visão quanto para a visão materializada, é possível executar uma consulta diretamente usando as cláusulas SQL padrão, como se fosse uma tabela do banco de dados. As visões materializadas (VMs) estão disponíveis nas implementações de diversos SGBDs de mercado (Chirkova, 2011). Abaixo tem-se um exemplo de VM criada no PostgreSQL 9.6.1:

```
CREATE MATERIALIZED VIEW mv_lineitem_suppkey AS
SELECT * FROM lineitem
WHERE l_suppkey BETWEEN 5000 AND 7000;
```

O uso das VMs deve ser bastante criterioso porque existem custos envolvidos para sua criação, atualização e utilização. Note-se que quando uma tabela é atualizada, as visões materializadas definidas a partir daquela tabela não são atualizadas automaticamente. Logo, deve haver no SGBD uma tarefa de atualização periódica para as VMs definidas no banco de dados e os procedimentos armazenados possivelmente alterados para referenciar explicitamente tais visões.

Ademais, há também os requisitos de espaço em disco que devem ser considerados, pois os dados armazenados em uma visão materializada estão duplicados. A escolha do conjunto de consultas a serem transformadas em visões materializadas é um problema com complexidade exponencial, dadas as restrições de espaço em disco e de custo de manutenção (Harinarayan, 1996).

De fato, o uso indiscriminado de visões materializadas pode levar a planos de execução com desempenho pior, quando comparados a planos alternativos que não utilizam as VMs. Se o uso de visões materializadas vai resultar em um plano com melhor ou pior desempenho, isto vai depender da consulta e das estatísticas do banco de dados. Para o autor, o ideal é que o próprio otimizador possa decidir sobre o uso ou não de uma VM, entretanto, para tanto, a consulta não pode referenciar explicitamente a visão materializada (Chauduri, 1995).

2.2 Índices hipotéticos

Índices hipotéticos são estruturas de índices virtuais, ou seja, existentes somente no catálogo do banco de dados. Um índice é considerado hipotético quando ele possui apenas informações de metadados, ou seja, não existe fisicamente na base de dados e também não pode ser usado efetivamente como estrutura para acessar o dado hipoteticamente indexado (Bruno, 2011).

Sendo ele percebido pelo otimizador de consultas do SGBD, seu maior benefício é o fato de possibilitar uma maneira de simular e descobrir como seria o plano de execução gerado pelo processador de consultas caso o índice hipotético existisse realmente. Consequentemente, pode-se descobrir quais índices melhorariam o desempenho de uma consulta sem o custo da efetiva criação destes índices no banco de dados.

Certamente esta é uma funcionalidade extremamente importante para DBAs. Índices hipotéticos permitem a criação de planos de execução hipotéticos e a consequente otimização de um plano de execução real. A otimização hipotética utiliza estruturas hipotéticas, como os índices hipotéticos, para gerar um plano de execução hipotético com custo de execução menor que o plano de execução original (Monteiro, 2008).

Em (PGSQL-PUC-RIO, 2016) pode-se encontrar uma implementação de índices hipotéticos para o SGBD PostgreSQL. Seu uso é mostrado a seguir. O primeiro *explain* mostra o plano de execução selecionado pelo otimizador para a consulta especificada, na configuração corrente do banco de dados.

```
EXPLAIN SELECT * FROM lineitem
WHERE l_orderkey=3 AND l_partkey=19036 AND l_suppkey=6540;
 Index Scan using lineitem_pkey on lineitem  (cost=0.43..36.81 rows=1 width=130)
   Index Cond: (l_orderkey = 3)
   Filter: ((l_partkey = 19036) AND (l_suppkey = 6540))
```

Na sequencia, supondo-se que um determinado índice poderá ser benéfico para a consulta, cria-se o índice em modo hipotético:

```
CREATE HYPOTHETICAL INDEX hi_lineitem_orderkey_partkey_suppkey ON
lineitem (l_orderkey, l_partkey, l_suppkey);
```

E utiliza-se o *explain hypothetical* (PGSQL-PUC-RIO, 2016) para verificar se o otimizador seleciona ou não o índice recém-criado:

```
EXPLAIN HYPOTHETICAL SELECT * FROM lineitem
WHERE l_orderkey=3 AND l_partkey=19036 AND l_suppkey=6540;
 Index Scan using hi_lineitem_orderkey_partkey_suppkey on lineitem  (cost=0.06..8.08
rows=1 width=130)
   Index Cond: ((l_orderkey = 3) AND (l_partkey = 19036) AND (l_suppkey = 6540))
```

Neste caso o otimizador do SGBD selecionou o índice hipotético porque o custo do plano gerado (8.08) é menor que o custo do plano sem este índice (36.81). Importante destacar que, para atingir este objetivo, o *explain hypothetical* (PGSQL-PUC-RIO, 2016) deve fazer uma série de estimativas sobre as propriedades do índice hipotético, já que ele não existe fisicamente. Assim, por exemplo, deve ser capaz de estimar o número de páginas em disco ocupadas pelo índice e o número de tuplas inseridas no índice.

Nota-se a diferença nos custos associados aos planos de execução que consideram ou não o índice hipotético. Sendo esta uma consulta frequente na carga de trabalho, conclui-se que este índice hipotético poderia ser materializado para trazer ganho de desempenho para a consulta.

As ferramentas de seleção de índices, tais como *SQL Server Database Engine Advisor* (SQL, 2016), *Oracle SQL Access Advisor* (ORA, 2016) e *DBX* (Almeida et al, 2015) fazem uso de índices hipotéticos com a finalidade de avaliar e sugerir um conjunto de índices candidatos (ou seja, índices que poderão, se criados, melhorar o desempenho das consultas ao banco de dados).

2.3 Particionamento

Através de um particionamento ou fragmentação, uma tabela é subdividida de acordo com uma restrição e cada parte da tabela torna-se um fragmento. No modelo relacional, o particionamento de dados consiste na divisão de uma relação em fragmentos menores através de seleção de tuplas – particionamento horizontal – ou projeção de colunas – particionamento vertical (Ramakrishnan, 2002).

No contexto da sintonia fina, uma consequência direta do particionamento é que ele tem o potencial de aumentar o desempenho das consultas que trabalham com subconjuntos de dados específicos. Na medida em que se pode determinar quais são estes subconjuntos, o particionamento da tabela vai fazer com que as consultas possam ser restringidas às partições da tabela que atendam às condições especificadas pela consulta e, assim, possam ter um desempenho melhorado em relação àquelas que necessitam varrer todo o conjunto de dados.

2.3.1 Tipos de fragmentação

A fragmentação pode ser horizontal ou vertical, ou ainda uma combinação dos dois tipos (híbrida). Na fragmentação horizontal, os subconjuntos são formados por linhas da tabela e, para tanto, uma coluna (e seu valor) deve ser designada como chave da fragmentação. Então, as linhas que têm um determinado valor (ou lista de valores) para esta coluna-chave geram um fragmento, as linhas que possuem outro valor geram outro fragmento, e assim por diante.

No exemplo da figura 2.2, linhas com $totalprice<200000$ formam um fragmento e linhas com $totalprice\geq200000$ formam outro fragmento.

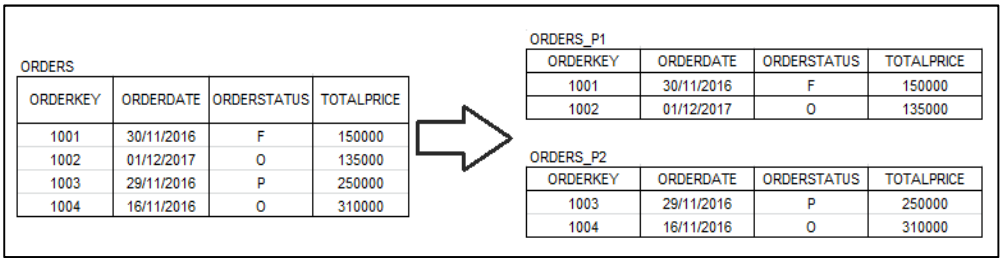

Figura 2.2: Fragmentação horizontal

A fragmentação horizontal pode ser primária ou derivada. A fragmentação horizontal primária é realizada usando predicados da própria relação que será particionada. Na figura 2.2 tem-se uma fragmentação horizontal primária, pois a chave do particionamento é um atributo da própria tabela que está sendo particionada.

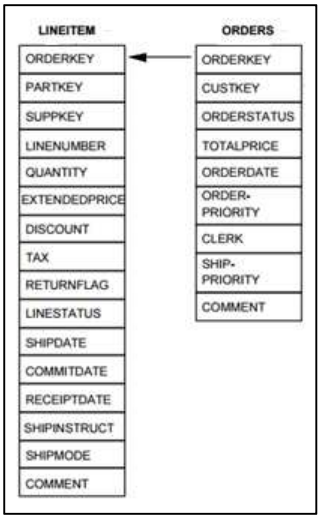

Figura 2.3: Fragmentação horizontal derivada

Se, por outro lado, para fragmentar uma determinada relação, predicados de outra relação são necessários, então tem-se uma fragmentação horizontal derivada. A fragmentação horizontal derivada é definida em uma relação membro (*lineitem* na figura 2.3) de acordo com uma operação de seleção especificada no seu proprietário (*orders*). O objetivo é particionar uma relação membro de acordo com uma condição no proprietário da ligação, sendo que os fragmentos resultantes são definidos apenas para os atributos da relação membro.

Na implementação da fragmentação, deve-se levar em consideração os relacionamentos existentes entre as tabelas, especialmente aqueles que serão frequentemente encontrados nas junções das consultas (Valduriez, 2011).

Na fragmentação vertical, os subconjuntos são colunas da tabela. Neste caso, em todos os fragmentos está presente a chave primária da tabela, o que permite a junção correta dos fragmentos e a reconstrução da tabela original.

No exemplo da figura 2.4, as colunas *orderkey* e *orderdate* formam um fragmento e as colunas *orderkey*, *orderstatus* e *totalprice*, outro.

ORDERS

ORDERKEY	ORDERDATE	ORDERSTATUS	TOTALPRICE
1001	30/11/2016	F	150000
1002	01/12/2017	O	135000
1003	29/11/2016	P	250000
1004	16/11/2016	O	310000

ORDERS_P1

ORDERKEY	ORDERDATE
1001	30/11/2016
1002	01/12/2017
1003	29/11/2016
1004	16/11/2016

ORDERS_P2

ORDERKEY	ORDERSTATUS	TOTALPRICE
1001	F	150000
1002	O	135000
1003	P	250000
1004	O	310000

Figura 2.4: Fragmentação vertical

Ambos os tipos de fragmentação devem apresentar três características marcantes para que possamos garantir sua correção: completitude, reconstrução e disjunção.

A completitude diz respeito à decomposição de uma relação em fragmentos e a garantia de que uma tupla qualquer encontra-se em um dos fragmentos; a reconstrução é a garantia de que existe um operador relacional capaz de reconstruir a relação completamente a partir dos seus fragmentos; e a disjunção é a característica que uma tupla qualquer está em apenas um dos fragmentos gerados (Valduriez, 2011).

Há ainda a fragmentação híbrida, que utiliza os dois tipos, horizontal e vertical, para criar uma cadeia de fragmentações. Uma alternativa, por exemplo, é fragmentar uma relação verticalmente e, em seguida, fragmentar horizontalmente para distribuir as linhas de cada partição vertical.

2.3.2 Conjuntos *minterm*

Os predicados usados nas consultas podem ser simples ou compostos. Predicados simples são do tipo $p_j : A_i \ \theta \ value$ onde θ é um operador de comparação (igual, diferente, maior, menor), por exemplo, *l_returnflag='N'* ou *l_tax > 0.5*.

Para que um conjunto de predicados gere as chaves para um particionamento, é preciso garantir que este conjunto seja mínimo e completo (*minterm*). Para formar um conjunto como esse, cada predicado simples deve fazer parte do conjunto na sua forma natural e na sua forma negada.

```
m1: l_returnflag = "N" ^ l_tax <= 0.5
m2: l_returnflag = "N" ^ l_tax > 0.5
m3: ¬( l_returnflag = "N") ^ l_tax <= 0.5
m4: ¬( l_returnflag = "N") ^ l_tax > 0.5
```

Sendo assim, sabendo que a aplicação trata registros com *l_returnflag* = {A, N, R}, para que o nosso conjunto de predicados seja completo, é preciso redefini-lo:

```
Pr = { l_returnflag="A", l_returnflag="N", l_returnflag="R", l_tax<=0.5, l_tax>0.5 }
```

Além de ser completo, todo conjunto de predicados deve ser mínimo, isto é, só conter predicados realmente utilizados pelas aplicações, considerados relevantes (Valduriez, 2011). Se adicionarmos *l_linestatus*='O' à *Pr*, o conjunto deixa de ser mínimo. Portanto, neste exemplo o conjunto de predicados *minterm* é:

```
m1: l_returnflag = "A" ^ l_tax <= 5000
m2: l_returnflag = "A" ^ l_tax > 5000
m3: l_returnflag = "N" ^ l_tax <= 5000
m4: l_returnflag = "N" ^ l_tax > 5000
m5: l_returnflag = "R" ^ l_tax <= 5000
m6: l_returnflag = "R" ^ l_tax > 5000
```

2.3.3 Particionamento lógico e físico

O particionamento deve considerar a alocação das partições das tabelas, isto é, onde as partições serão armazenadas fisicamente. As partições podem ser alocadas em um único SGBD ou podem ser distribuídas em dois ou mais SGBDs. Além disso, elas podem ser armazenadas em uma mesma área de armazenamento de tabelas ou em diferentes áreas de armazenamento.

O particionamento das tabelas pode ser lógico ou físico. No particionamento lógico, as partições são alocadas em uma mesma área de armazenamento de tabelas, em um único SGBD, conforme ilustrado na figura 2.5.

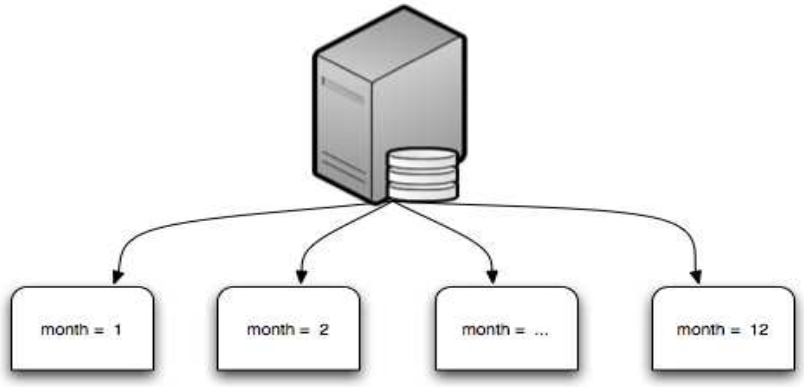

Figura 2.5: Particionamento lógico

Quando o particionamento é físico, as partições são armazenadas em áreas de armazenamento distintas, em um ou mais SGBDs.

2.3.4 Replicação

O particionamento é normalmente aplicado nas tabelas mais frequentemente acessadas pela carga de trabalho e cujas varreduras representam os maiores custos nos planos de execução gerados pelo otimizador do SGBD.

Quando o particionamento é físico, é preciso garantir que as tabelas menores, por exemplo, as tabelas dimensão de um *data warehouse*, estejam disponíveis de modo a permitir junções locais entre as tabelas particionadas e estas tabelas menores. Isto é garantido pela replicação, que consiste na manutenção de réplicas

atualizadas das linhas de uma tabela em áreas de armazenamento distintas. Há trabalhos como, por exemplo, (Curino, 2010), que combinam particionamento e replicação para garantir que as consultas acessem dados em uma única área de armazenamento de tabelas.

Quando o particionamento é lógico, a replicação não é necessária, já que todas as partições das tabelas estão armazenadas fisicamente em uma única área de armazenamento.

2.3.5 Particionamento virtual

O particionamento virtual é criado através da replicação. Replica-se todo o banco de dados para *n* nós e, como explica (Akal et al. 2002), as consultas são reescritas adicionando-se um predicado do tipo *range* à cláusula *where* da consulta original. Assim, a consulta pode ser executada em todos os nós do *cluster* conforme a figura 2.6:

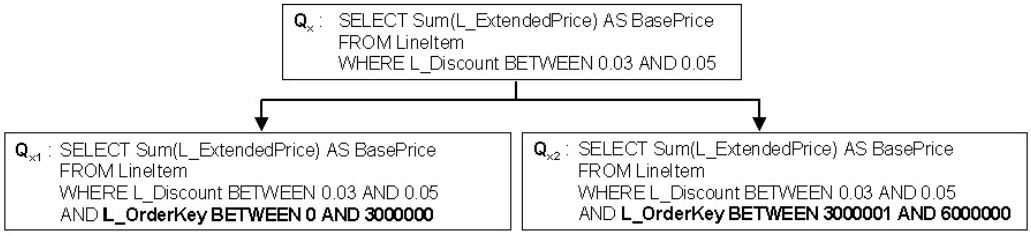

Figura 2.6: Particionamento virtual (Akal et al., 2002)

O particionamento virtual também pode ser lógico ou físico. Segundo (Furtado et al., 2005), o particionamento físico (real) requer um projeto cuidadoso para distribuir o banco de dados e gerar as partições físicas, sendo bastante sensível aos padrões das consultas.

2.4 Estratégias de sintonia fina

Como estratégias de sintonia fina, índices, índices parciais e visões materializadas são alternativas para o otimizador, para seleção do melhor plano de execução. Considerando um cenário onde tais estruturas foram criadas (seções 2.1.1, 2.1.2 e 2.1.3), as estimativas de custo geradas pelo otimizador permitem selecionar aquela que vai compor o melhor plano de execução. Na consulta abaixo, o otimizador seleciona o índice parcial *pi_lineitem_orderkey_partkey _suppkey* com um custo = 8.45.

```
EXPLAIN SELECT * FROM lineitem
WHERE l_orderkey=3 AND l_partkey=19036 AND l_suppkey=6540;
 Index Scan using pi_lineitem_orderkey_partkey_suppkey on lineitem  (cost=0.43..8.45
rows=1 width=130)
   Index Cond: ((l_orderkey = 3) AND (l_partkey = 19036) AND (l_suppkey = 6540))
```

Ao verificar o índice hipotético *hi_lineitem_orderkey_partkey_suppkey*, obtem-se um custo ainda menor, 8.08, conforme abaixo.

```
EXPLAIN HYPOTHETICAL select * FROM lineitem
WHERE l_orderkey=3 AND l_partkey=19036 AND l_suppkey=6540;
 Index Scan using hi_lineitem_orderkey_partkey_suppkey on lineitem  (cost=0.06..8.08
rows=1 width=130)
   Index Cond: ((l_orderkey = 3) AND (l_partkey = 19036) AND (l_suppkey = 6540))
```

Tome-se, por outro exemplo, o cenário de um banco de dados de imóveis destinados à venda. Usuários emitem consultas a este banco de dados, pesquisando por imóveis em seu bairro, em sua cidade, pelo número de quartos ou por outra característica qualquer. Neste cenário, as consultas mais frequentes são:

```
SELECT * FROM unidade u – consulta_1
JOIN endereco e ON (e.codigo = u.endereco)
WHERE e.unidadeFederacao = 'RJ'
AND e.municipio = 'Rio de Janeiro'
AND e.bairro = 'Barra da Tijuca'
```

```
SELECT * FROM unidade u – consulta_2
JOIN endereco e ON (e.codigo = u.endereco)
WHERE e.unidadeFederacao = 'RJ'
AND e.municipio = 'Rio de Janeiro'
```

```
SELECT * FROM unidade u – consulta_3
JOIN endereco e ON (e.codigo = u.endereco)
WHERE e.unidadeFederacao = 'RJ'
```

As consultas utilizam as colunas *unidadeFederacao*, *municipio* e *bairro* em suas cláusulas *where*. Considere-se também, para este cenário, que existem imóveis à venda em todos os bairros, em todos os municípios e em todas as unidades da federação do Brasil, e que a distribuição de valores para estes atributos é uniforme, isto é, há o mesmo número de imóveis para cada bairro, para cada município e para cada unidade da federação.

Sabendo-se que o Brasil possui 27 unidades da federação (26 estados mais o distrito federal), cerca de 5.500 cidades e 14.000 bairros, tem-se a seguinte distribuição de tuplas na tabela *endereco*:

Coluna	**Distribuição**
Unidade da Federação	3,7%
Município	0,018%
Bairro	0,007%

Tabela 2.1: Seletividade

De acordo com (Morelli, 2006), a seletividade determina a utilização ou não de um índice pelos otimizadores de consulta dos SGBDs. Seletividade é uma medida do percentual de tuplas de uma tabela que satisfaz um predicado de uma consulta, ou seja, seletividade é uma propriedade do predicado da consulta. De acordo com (Technet, 2009), seletividade é a fração de linhas do conjunto de entrada que satisfaz um predicado. Então, se temos 100.000 linhas em uma tabela e uma consulta usa um predicado que retorna uma linha desta tabela, a seletividade deste predicado nesta consulta é 1/100.000 = 0,001%. Os percentuais de distribuição mostrados na tabela 2.1 são, na verdade, suas seletividades quando utilizados como predicados em consultas.

Quanto maior a seletividade de um predicado (seletividade alta), menor é o percentual de tuplas retornadas pela consulta que usa o predicado. Quanto menor a seletividade de um predicado (seletividade baixa), maior é o percentual de tuplas retornadas pela consulta que usa o predicado. Para ilustrar o conceito de seletividade, seja a tabela *endereco* com 1.000 linhas, sendo 100 com *uf=”RJ”* e 900 com *uf=”SP”*. Considere também que existe um índice secundário *non-clustered* criado usando esta coluna *uf* como chave. Abaixo os planos de execução das consultas com predicado nesta coluna.

```
EXPLAIN SELECT * FROM endereco
WHERE uf='SP';
 Seq Scan on endereco  (cost=0.00..109.50 rows=900 width=164)
   Filter: (uf = 'SP')
```

```
EXPLAIN SELECT * FROM endereco
WHERE uf='RJ';
 Seq Scan on endereco  (cost=0.00..126.25 rows=100 width=164)
   Filter: (uf = 'RJ')
```

Nota-se que o otimizador não seleciona o índice na coluna *uf* para nenhum dos dois predicados. Isto ocorre porque as estimativas de custo do SGBD sugerem ao otimizador que é melhor percorrer a tabela inteira e aplicar o filtro do que percorrer o índice e acessar as linhas na tabela.

De fato, quanto maior a seletividade de um predicado, maior a probabilidade de que um índice na coluna correspondente seja utilizado pelo otimizador. Nestes casos, como as consultas que utilizam estes predicados retornam poucas tuplas, o otimizador opta por um *Index Scan* ao invés de um *Table Scan*. De outro modo, colunas com predicados de baixa seletividade não são boas candidatas a chaves de índices, pois o otimizador prefere varrer a tabela inteira a ter que varrer o índice e acessar o registro correspondente na tabela.

Considere então que mais linhas com *uf=”SP”* sejam inseridas na tabela de modo a aumentar a seletividade do predicado *uf=”RJ”*. No SGBD PostgreSQL 9.6.1, somente quando o número de linhas de *uf=”SP”* chega a aproximadamente 3.000, é que o otimizador seleciona o índice para o predicado *uf=”RJ”* (3,33% aproximadamente).

```
EXPLAIN SELECT * FROM endereco
WHERE uf='RJ';
 Index Scan using i_endereco_uf on endereco  (cost=0.28..62.76 rows=100 width=164)
   Index Cond: (uf = 'RJ')
```

No SGBD SQL Server 2008 R2 (SQL, 2008) este mesmo teste conferiu resultado de 0,2% para que o otimizador selecionasse o índice secundário para o plano de execução da consulta.

Dentre todas as atividades relacionadas à sintonia de bancos de dados, o ajuste das estruturas de índices é certamente uma das mais relevantes (Monteiro, 2008).

Ao criar um índice pelas colunas *unidadeFederacao*, *municipio* e *bairro*, observa-se que o plano de execução da *consulta_1* executa um *Index Scan* no índice criado.

Ao criar um índice pelas colunas *unidadeFederacao* e *municipio*, observa-se que o plano de execução da *consulta_2* executa um *Index Scan* no índice criado.

Ao criar um índice apenas na coluna *unidadeFederacao*, observa-se que o plano de execução da *consulta_3* não executa um *Index Scan* no índice criado; ao contrário disso, continua executando um *Table Scan* em toda a tabela.

No cenário utilizado como exemplo, nota-se que a coluna *unidadeFederacao* possui baixa seletividade, o que faz com que os planos de execução da *consulta_3* utilizem um *Table Scan*. Já as colunas *municipio* e *bairro* possuem alta seletividade, fazendo com que índices criados com base nestas colunas tenham bom potencial de seleção pelo otimizador.

O particionamento da tabela *endereco* pode ser uma estratégia de sintonia fina, na medida em que vai alocar em diferentes partições os endereços de unidades da federação distintas, diminuindo o espaço de varredura de *Sequential Scans* na tabela *endereco*.

A estratégia de índices parciais pode ser considerada neste exemplo. Tais estruturas são índices *non-clustered* construídos sobre um subconjunto definido a partir de um predicado e que só são utilizados pelos otimizadores dos SGBDs quando constatado que somente um subconjunto restrito do índice parcial é acessado pela consulta (Wu, Madden, 2011). Sendo assim, um índice parcial construído sobre a coluna *uf*, restrição *uf='RJ'*, poderia ser selecionado pelo otimizador.

A utilização de uma estratégia ou outra deve se basear na análise global da carga de trabalho. No exemplo dado, as consultas com *uf='SP'* não seriam atendidas pelo índice parcial. Logo, a sintonia fina deve considerar a frequência das consultas que utilizam o predicado *uf='RJ'*. Caso a frequência seja muito maior do que aquelas que usam o predicado *uf='SP'*, a estratégia de índices parciais pode ser uma alternativa melhor do que o particionamento.

2.5 Histogramas

Histogramas representam aproximações das distribuições de frequência dos valores dos atributos das relações do banco de dados (Ioannidis, Poosala, 1995). Especificamente, o histograma de um atributo é obtido através dos pares (valor, frequência) e os valores do atributo são utilizados para gerar faixas, sendo cada faixa associada a um *bucket*.

Existem diferentes tipos de histogramas. Os denominados *triviais* consideram uma distribuição uniforme dos pares pelos *buckets*, ou seja, consideram que existe uma distribuição uniforme das frequências por todo o domínio de valores do atributo.

Outros dois tipos são o *equi-width* e o *equi-depth*.

Em histogramas *equi-width* o tamanho da faixa de valores em cada *bucket* é o mesmo, ou seja, todos os *buckets* contêm o mesmo número de valores, independente da frequência de cada valor. Em histogramas *equi-depth* a soma das frequências dos valores em cada *bucket* é a mesma, independente do número de valores em cada *bucket*.

Em [SQL, 2016] um histograma é implementado como um relatório estatístico que mostra a frequência de valores dentro de faixas de valores que se situam entre um determinado mínimo e máximo. As faixas de valores são normalmente organizadas em *steps*, com cada *step* tendo um valor mínimo e máximo, conforme exemplo abaixo:

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1992-01-03	0	1	0	00001,00
1992-02-21	24216,8	608,3768	47	00513,38
1992-03-18	31241,85	1086,387	25	1245,053
1992-03-29	15184,59	1347,12	10	1512,521
1992-04-14	28972,89	1955,497	15	1924,202
1992-04-26	28143,85	2390,052	11	02548,61
1992-05-07	21162,43	2781,151	10	2107,968
1992-05-30	52491,55	2954,973	22	2377,111
1992-06-21	53015,15	3172,25	21	2515,128
1992-07-04	28885,62	1825,13	12	2397,865
1992-07-17	30325,54	2433,507	12	2517,396
1992-07-30	29409,23	2042,408	12	2441,331
1992-08-06	16624,5	1998,952	6	2759,263

As faixas de valores são delimitadas em RANGE_HI_KEY e o número de valores em cada *step* está informado em RANGE_ROWS. O segundo step compreende os valores [1992-01-03, 1992-02-21) e possui um número estimado de 24216 linhas.

EQ_ROWS é o número estimado de linhas cujo valor é igual ao limite superior do *step*, DISTINCT_RANGE_ROWS é o número de valores distintos da coluna dentro do *step* e AVG_RANGE_ROWS é o número médio de linhas com valores duplicados na coluna.

Em [ORA, 2016], um histograma de frequência armazena o número de linhas para cada valor – ou faixa de valores – da coluna, conforme mostrado abaixo. *Endpoint_value* é o valor mais alto da coluna no *bucket*, *Endpoint_number* é o número de linhas existentes na tabela até aquele valor e *Frequency* é a frequência acumulada.

ENDPOINT_VALUE	ENDPOINT_NUMBER	FREQUENCY
0	5000	5000
1	5535	535
2	6104	569
3	6679	575
4	7243	564
5	7773	530
6	8329	556
7	8889	560
8	9469	580
9	10000	531

Conforme [PGSQL, 2016], o histograma divide a faixa de valores de uma coluna em intervalos de mesma frequência de ocorrência na tabela. Então esta divisão informada pelo histograma contém partes (ou partições) com aproximadamente o mesmo número de linhas em cada parte.

```
                    histogram_bounds
-----------------------------------------------------
 {0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995}
```

Nota-se que os histogramas gerados pelos SGBDs fornecem um modo de particionar tabelas, sendo que os diferentes tipos de histogramas podem ser utilizados para gerar um particionamento que atenda a diferentes necessidades, por exemplo, volumes ou frequências de acesso semelhantes em cada partição.

2.6 Resumo do capítulo

Neste capítulo foram apresentados os conceitos e fundamentos básicos da sintonia fina de bancos de dados relacionais, com ênfase nas estruturas de acesso tradicionais utilizadas pelo otimizador de consultas do SGBD, como índices, índices parciais e visões materializadas.

Além disso, foram apresentados os conceitos correspondentes ao particionamento de tabelas. Foi mostrada a diferença entre o particionamento horizontal e o particionamento vertical e como ambos podem reduzir o espaço de varredura de uma tabela. Foram mostradas também as diferenças entre o particionamento físico e lógico, além do conceito de particionamento virtual.

O conceito de seletividade ganhou atenção maior, na medida da sua importância para a sintonia fina. Um exemplo com um banco de dados de imóveis é dado e testes mostram o limiar da seletividade que faz o otimizador selecionar um índice ou não.

No final do capítulo são mostrados os histogramas e os tipos existentes. As faixas de valores que compõem os histogramas *equi-width* possuem o mesmo tamanho, enquanto as faixas de valores que compõem os histogramas *equi-depth* possuem a mesma frequência. Como exemplo, sejam 6 valores distintos (1..6) em um conjunto, sendo que os dois últimos se repetem 2 vezes mais que os quatro primeiros. Um histograma *equi-width* de dois *buckets* contem as faixas (1..3) e (4..6). Um histograma *equi-depth* de dois *buckets* contém as faixas (1..4) e (5..6).

São fornecidos exemplos reais de histogramas em SGBDs padrão de mercado, evidenciando que são uma ferramenta que pode ser utilizada na seleção da estratégia de particionamento.

3 Trabalhos relacionados

Não foram encontrados na literatura acadêmica uma grande diversidade de trabalhos relacionados a particionamento como ação de sintonia fina em bancos de dados relacionais. Neste capítulo os trabalhos relacionados estão organizados em dois grupos: aqueles que expõem técnicas para utilizá-lo como ação de sintonia fina, e aqueles que tratam da seleção de chaves de particionamento das tabelas.

3.1 Estratégias de sintonia fina

Em (Bellatreche, 2004) é mostrada uma comparação entre o uso de índices e particionamento com o *benchmark* APB-1 (APB1, 1998), destinado a análises com bancos de dados OLAP. Neste trabalho, os cenários criados para os testes objetivam variar as frequências das consultas potenciais e a quantidade de operações de atualização, o que impacta diretamente a decisão da estratégia de sintonia fina. Compara os tempos de resposta na execução de uma carga de trabalho com as diferentes ações de sintonia fina.

Percebe-se claramente uma tendência nos resultados coletados em (Bellatreche, 2004). Ao aumentar-se o número de consultas na carga de trabalho, mantendo-se o mesmo número de atualizações, o particionamento tende a ser uma estratégia melhor do que os índices para a sintonia fina. Para os tempos de resposta das consultas, o particionamento é mais eficiente, enquanto os índices somente passam a apresentar melhor desempenho quando a seletividade do predicado é alta. Para os tempos de atualização, os índices são mais eficientes.

As conclusões de (Bellatreche, 2004) indicam que o particionamento é recomendado quando a seletividade dos predicados é baixa ou quando a frequência das operações de atualização é baixa comparada à frequência de consultas. Índices são recomendados quando a seletividade dos predicados é alta ou a frequência das operações de atualização é similar à frequência de consultas.

O trabalho de (Agrawal, 2004) pesquisa a utilização de índices e particionamento simultaneamente e mostra a diferença entre as abordagens integrada e não-integrada na seleção de estratégias de sintonia fina. Utiliza o bechmark TPC-H (TPC, 2016) e considera a consulta abaixo.

```
SELECT l_returnflag, l_linestatus, sum(l_quantity), count(*)
FROM lineitem
WHERE l_shipdate <= '1998/12/08'
GROUP BY l_returnflag, l_linestatus
ORDER BY l_returnflag, l_linestatus
```

Em uma abordagem não integrada, o melhor índice é escolhido independentemente do particionamento realizado, isto é, benefícios e custos de índices e particionamento são estimados de forma isolada. Em uma abordagem integrada, índice e particionamento são considerados em conjunto para avaliação de benefícios e custos.

Para a consulta acima, em uma abordagem não integrada, a seleção do melhor índice é (*l_shipdate*, *l_returnflag*, *l_linestatus*, *l_quantity*), enquanto o particionamento mais adequado é o do tipo *hash* em (*l_returnflag*, *l_shipdate*). Entretanto, esta não é a alternativa que maximiza o desempenho da consulta. Ao considerar as duas estratégias de forma integrada obtem-se um índice em (*l_returnflag*, *l_linestatus*, *l_shipdate*, *l_quantity*) e o particionamento do tipo *range* em (*l_shipdate*).

Abordagem	Índice	Particionamento
Integrada	*(l_returnflag, l_linestatus, l_shipdate, l_ quantity*)	Range em *(l_shipdate*)
Não integrada	(*l_shipdate, l_returnflag, l_linestatus, l_quantity*)	Hash em (*l_returnflag, l_linestatus*)

Tabela 3.1: Abordagens integrada e não integrada (Agrawal, 2004)

A consulta roda com desempenho 30% superior com a abordagem integrada. Assim, (Agrawal, 2004) evidencia que as ações de sintonia fina, ao serem consideradas como uma única ação em conjunto, podem gerar um benefício ainda maior do que quando consideradas de forma isolada.

O trabalho relatado em (Furtado et al., 2005) avalia o desempenho de uma carga de trabalho OLAP usando particionamento físico e virtual em um *cluster* de bancos de dados. O trabalho explora o processamento da mesma consulta em diversos subconjuntos da tabela para reduzir o tempo de execução das mesmas.

O particionamento físico gera os subconjuntos da tabela e os aloca em diferentes servidores. Já o particionamento virtual usa a replicação completa do banco de dados nos diferentes servidores e a redefinição da consulta com os predicados que fazem a varredura nos subconjuntos correspondentes. O trabalho mostra que a estratégia de particionamento virtual pode apresentar benefícios para a sintonia fina, mas depende da reescrita das consultas (Furtado et al., 2005).

Nesta linha de pesquisa, o trabalho de (Furtado et al., 2005) apresenta uma variação do particionamento virtual denominada AVP – *Adaptative Virtual Partitioning* – que executa uma sintonia no tamanho das partições virtuais, tentando mantê-lo o menor possível. Partições grandes são um problema para o particionamento virtual quando o otimizador seleciona um *Sequential Scan* na tabela ao invés de um *Index Scan*. Independente do SGBD, a sintonia se baseia exclusivamente no comportamento das consultas, não considerando características como tamanho dos discos nos servidores, tamanho da memória principal, entre outras.

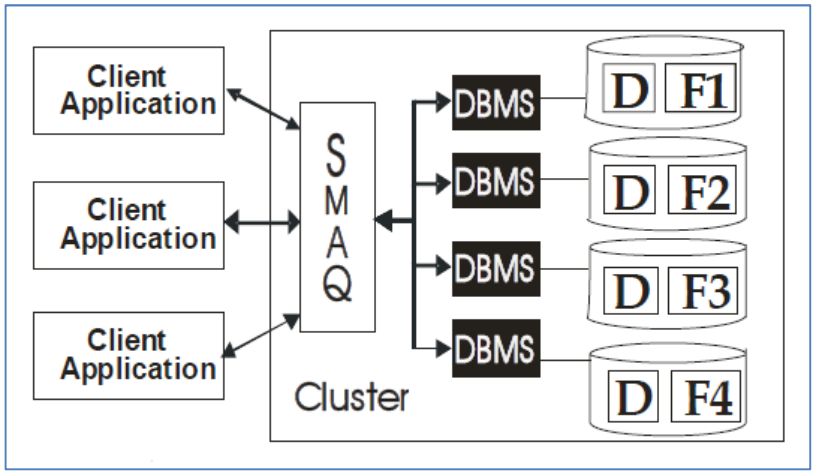

Figura 3.1: Particionamento virtual (Furtado et al., 2005)

A figura 3.1 mostra o componente *SMAQ* (Smashing Queries) recebendo as solicitações dos clientes, que são consultas OLAP. O banco de dados está fragmentado horizontalmente em 4 partições (F1..F4) e D representa o conjunto de tabelas dimensão, replicadas em todos os nós. SMAQ é responsável por reescrever as consultas recebidas e redirecioná-las para as partições correspondentes. Os resultados do trabalho mostram bons resultados no uso do AVP em *clusters* de até 32 nós, similares ao uso do particionamento vertical com replicação completa.

Em (Costa, Furtado, 2006), os autores propõem uma arquitetura – GRID–NPDW – *Node Partitioned Data Warehouse* – para implementação de grandes *warehouses* em *grids* (conjunto de recursos computacionais distribuídos) com foco em disponibilidade e desempenho. A proposta é baseada em particionamento, alocando e realocando os fragmentos de acordo com as características dinâmicas de cada nó e de cada sítio e garantindo balanceamento de carga no arranjo das alocações.

Um requisito relevante para distribuição de *warehouses* em *grids* é a contingência, pois nestes ambientes as condições das redes e a disponibilidade dos equipamentos é imprevisível (Costa, Furtado, 2006). Assim, uma característica fundamental da solução proposta deve ser a manutenção de réplicas dos dados em diversos nós, de tal forma que no evento de uma falha em um dos nós, outro possa receber as solicitações enviadas para aquele nó.

Na arquitetura GRID–NPDW as tabelas muito grandes são particionadas de acordo com a carga de trabalho e as tabelas dimensão são replicadas em todos os servidores. A replicação das tabelas dimensão é realizada para evitar-se migração de dados entre nós no processamento de junções.

As partições definidas são replicadas em grupos – PRG – *Partitioned Replica Groups* – para permitir a continuidade do processamento mesmo em caso de falha de diversos nós e evitar que somente um nó assuma todo o processamento de um nó que falhou.

Para tanto, a réplica de uma partição é dividida em partes e essas partes são copiadas para nós diferentes, de forma que todos eles assumam as solicitações em caso de falha do nó proprietário da partição principal. Os nós são organizados em grupos, de modo que as partições nos nós de um grupo sejam replicadas para nós de outros grupos, permitindo que todo um grupo possa falhar sem afetar o processamento das consultas.

O trabalho em (Lima et al., 2009) também explora o particionamento virtual e o processamento da mesma consulta em diversos nós de um cluster de banco de dados. O trabalho mostra a importância do desenho de uma distribuição de dados para a eficiência do paralelismo na execução das consultas e propõe uma estratégia de projeto que maximize o desempenho de cargas de trabalho OLAP. Como replicar completamente um banco de dados pelos nós de um *cluster* tem um alto custo pelo consumo de espaço em disco, a estratégia é baseada em particionamento físico e virtual combinada com técnicas de replicação.

3.2 Seleção de chaves de particionamento

O trabalho em (Curino et al., 2010) utiliza grafos para representar uma carga de trabalho e algoritmos de particionamento de grafos para selecionar uma estratégia de particionamento para as tabelas envolvidas. *Schism*, a abordagem proposta, também considera a replicação como parte da solução.

Focado em ambientes OLTP, esta abordagem objetiva utilizar as estratégias de particionamento e replicação para melhorar a escalabilidade de bancos de dados distribuídos *shared-nothing*. Segundo (Curino et al., 2010), como as transações distribuídas são caras em ambientes OLTP, o objetivo é minimizar o número de transações distribuídas, enquanto produz partições balanceadas.

Schism trabalha em duas fases. Na primeira, um grafo é gerado a partir da carga de trabalho, sendo que os nós do grafo são as tuplas do banco de dados. Uma aresta é criada entre dois nós sempre que as tuplas correspondentes forem acessadas pela mesma transação. Na representação abaixo, as cinco tuplas da tabela *account* são acessadas por quatro transações.

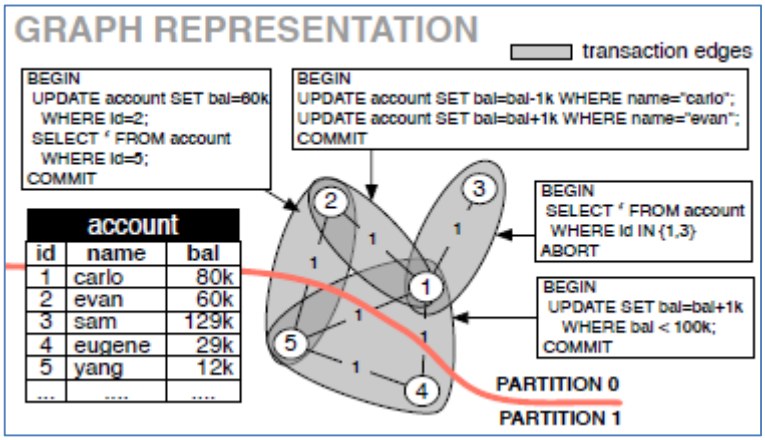

Figura 3.2: Particionamento usando grafos (Curino et al., 2010)

Em seguida, um algoritmo de particionamento de grafos é executado para dividir o grafo em *k* partições balanceadas que minimizem o número de transações em mais de uma partição. Para o balanceamento, o peso de uma partição é o somatório dos pesos das arestas localizadas naquela partição.

Na segunda fase, uma explanação do grafo particionado é gerada, isto é, um modelo que capture os mapeamentos (tupla, partição) gerados na fase anterior é produzido através do algoritmo *decision tree*. A saída do algoritmo é um conjunto de regras que pode ser utilizado diretamente com os predicados para um particionamento do tipo *range*. No exemplo dado, temos a seguinte saída:

$$\begin{aligned} (id = 1) &\rightarrow \ partitions = \{0, 1\} \\ (2 \leq id < 4) &\rightarrow \ partition = 0 \\ (id \geq 4) &\rightarrow \ partition = 1 \end{aligned}$$

Figura 3.3: Modelo de particionamento (Curino et al., 2010)

O trabalho em (Curino et al., 2011) apresenta um método para particionar tabelas, que depende como as transações e os dados se relacionam entre si. *A* estratégia de particionamento é selecionada com o uso de grafos, assim como em (Curino et al., 2010), mapeando linhas das tabelas para nós e minimizando o número de transações entre nós.

Como evidencia (Curino et al, 2011), cargas de trabalho OLTP são caracterizadas por transações com pouco paralelismo interno. O caminho para escalar essas cargas de trabalho é particionar os dados de uma forma que minimize o número de transações multi-nó (isto é, a maioria das transações deve ser completada tocando dados em apenas um nó) e, em seguida, alocar as diferentes partições em diferentes nós. O objetivo é minimizar o número de transações distribuídas, que incorrem em sobrecarga tanto por causa do trabalho extra feito em cada nó, como também pelo aumento do tempo gasto mantendo bloqueios nos servidores.

(Wu, Madden, 2011) apresenta *Shinobi*, um sistema que usa particionamento horizontal como mecanismo de sintonia fina. Particiona as tabelas de tal maneira que regiões das tabelas consultadas em junções são armazenadas juntas, separadas das regiões menos frequentemente acessadas. Índices são criados somente nestas partições mais consultadas e, ao longo do tempo, ajusta dinamicamente as partições e os índices, adequando-se às mudanças na carga de trabalho.

O custo das consultas é calculado com base nas varreduras de tabelas e índices em cada partição. Segundo (Wu, Madden, 2011), para uma determinada consulta, caso um índice não seja utilizado, então o custo do *Sequential Scan* na tabela, na partição correspondente, é contabilizado. Junções entre uma tabela T1 com chave primária *k* e uma tabela T2 com chave estrangeira *fk* referenciando *k* são tratadas como uma varredura em T1 por *k* e uma varredura em T2 por *fk*. Caso um índice seja usado, o custo depende do índice estar ordenado pela chave de busca (*clustered*) ou não (*non-clustered*). Se for *clustered* o custo do *Index Scan* no índice, na partição correspondente, é contabilizado.

As conclusões em (Wu, Madden, 2011) mostram que o particionamento de fato reduz significativamente os custos das consultas quando o conjunto de dados não está ordenado fisicamente pelas chaves de particionamento, enquanto a indexação seletiva pode reduzir o tamanho dos índices e seus custos.

Na figura 3.4 pode-se visualizar a arquitetura de *Shinobi* (Wu, Madden, 2011), com um componente que implementa o modelo de custos conectado ao componente que observa a carga de trabalho. O componente *Dynamic Repartitioner* emite os comandos para o particionamento das tabelas e o componente *Query Rewriter* reescreve as consultas para adequá-las ao particionamento.

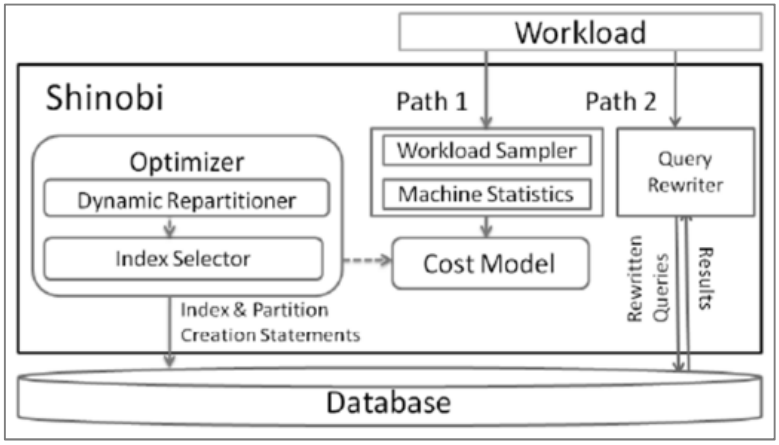

Figura 3.4: Arquitetura de *Shinobi* (Wu, Madden, 2011)

(Liroz-Gistau et al., 2012) apresenta *DynPart*, um algoritmo de particionamento dinâmico para bancos de dados científicos muito grandes em crescimento contínuo. Um esquema de particionamento eficiente tenta minimizar o número de fragmentos acessados na execução de uma consulta, reduzindo, assim, a sobrecarga associada para lidar com uma execução distribuída.

Soluções usando particionamento vertical, tais como *column-store databases*, podem ser úteis para o projeto físico em cada nó, mas não fornecem um particionamento distribuído eficiente, em particular para aplicações com consultas com várias dimensões, onde junções exigiriam transferências de dados entre os nós.

Abordagens horizontais tradicionais, como *hash* ou *range*, não são capazes de capturar o comportamento complexo dos padrões de acesso de aplicações científicas, especialmente porque costumam fazer uso de relacionamentos complexos sobre grandes conjuntos de colunas, e são difíceis de serem pré-definidos.

O trabalho faz uma diferenciação entre o uso do particionamento para aplicações estáticas e aplicações dinâmicas com crescimento contínuo das bases de dados. Segundo (Liroz-Gistau et al., 2012), a abordagem de grafos que representem relacionamentos entre consultas e dados é eficiente mas sempre requer que todo o grafo seja explorado para obter a estratégia de particionamento. Em cenários onde novas tuplas são inseridas frequente e continuamente, caso das aplicações científicas, esta estratégia pode se tornar proibitiva.

Para o particionamento estático, a eficiência do particionamento para uma dada consulta é a razão entre o mínimo de fragmentos possíveis e o número de fragmentos acessados pela consulta. Quando este número é 1, a eficiência é máxima. E a eficiência de uma estratégia de parcionamento para uma dada carga de trabalho é o somatório das eficiências encontradas em cada consulta da carga de trabalho.

Os resultados das experiências em (Liroz-Gistau et al., 2012) mostram que a estratégia de particionamento dinâmico é capaz de lidar eficazmente com os dados de aplicações científicas, mas que também sua utilização não está limitada a este tipo de aplicação, podendo ser usado para particionamento de dados em muitas outras aplicações em que os itens de dados são atualizados continuamente.

(Wang, 2013) apresenta ASAWA - *Automatic Selection based on Attribute Weight Analysis* – uma estratégia que calcula uma pontuação global para cada atributo usado nas consultas com base na dependência entre as tabelas, descobre atributos correlacionados e seleciona as chaves de particionamento pelas melhores pontuações obtidas.

Para ASAWA as chaves de particionamento candidatas de cada tabela são as chaves primárias, chaves estrangeiras e as colunas declaradas como *Unique* ou *Not Null*. Pesos diferenciados são atribuídos para cada uma dessas restrições, o que pode influenciar a decisão do algoritmo e depende do tipo de aplicação que utiliza o banco de dados.

O algoritmo então utiliza os pesos calculados para considerar as estratégias de particionamento e seleciona aquela que produz o melhor desempenho para as consultas da carga de trabalho, ou seja, o algoritmo considera todas as combinações de atributos para chegar à melhor combinação. Para cada combinação, chama o otimizador para estimar o tempo de execução das consultas.

O trabalho busca coletar resultados com o *benchmark* TPC-H e consultas selecionadas. A conclusão é que de fato o algoritmo proporciona ganhos de desempenho consideráveis quando comparadas à outras estratégias de particionamento.

3.3 Resumo do capítulo

Os trabalhos (Bellatreche, 2004) e (Agrawal, 2004) mostram o potencial do particionamento como ação de sintonia fina. Enquanto o primeiro compara ações de sintonia fina com resultados obtidos em testes experimentais e aponta conclusões sobre o uso de índices e particionamento, o segundo mostra como essas ações podem ser integradas para obter um benefício maior do que as mesmas ações implementadas de forma não integrada.

Outros trabalhos, como (Furtado et al., 2005) e (Lima et al., 2009), utilizam o particionamento virtual para distribuir tarefas em agrupamentos de servidores. Já (Costa, Furtado, 2006) utiliza particionamento físico e replicação em *grids* para prover melhor desempenho e escalabilidade em sistemas de bancos de dados.

Os trabalhos (Madden, 2011) e (Wang, 2013) tratam da seleção da estratégia de particionamento. O primeiro apresenta um modelo de custos com base nas varreduras de tabelas e índices e o segundo apresenta um método para seleção de chaves de particionamento baseado no peso dos atributos. Em (Liroz-Gistau et al., 2012) pode-se verificar a diferença entre o uso do particionamento para aplicações estáticas e aplicações dinâmicas com crescimento contínuo das bases de dados.

4 Particionamento como ação de sintonia fina

Neste capítulo é apresentado um método para abordar o problema do particionamento e como ele pode ser implementado.

4.1 Abordagem para o problema do particionamento

Selecionar uma estratégia de particionamento para um esquema de banco de dados e uma carga de trabalho envolve percorrer etapas bem definidas. Tais etapas estão mostradas na figura 4.1.

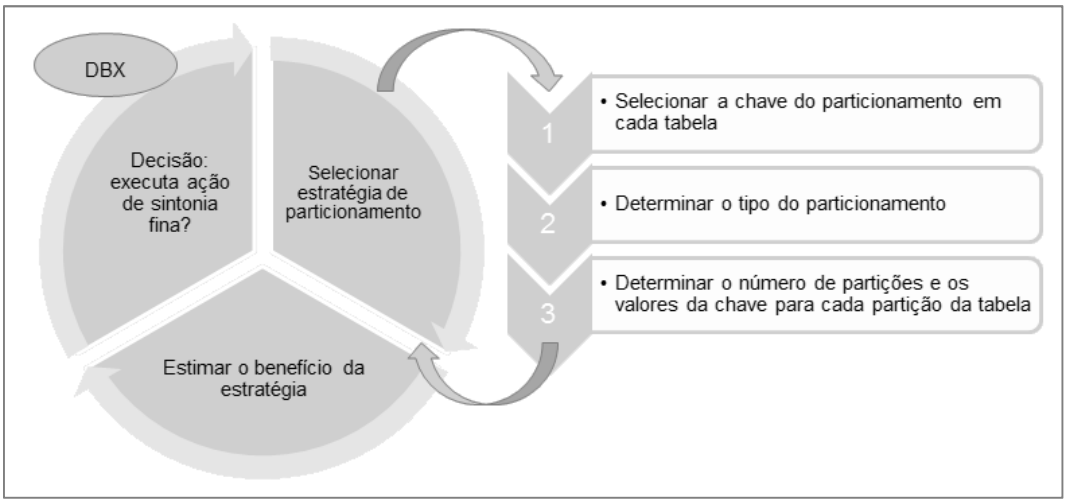

Figura 4.1: O problema do particionamento

Um ciclo se inicia durante a fase de observação da carga de trabalho, na qual uma estratégia de particionamento para o esquema do banco de dados é gerada. A duração desta fase é finita e determinada pelo DBA. A partir desta estratégia e da carga de trabalho observada, o benefício do particionamento é estimado, levando-se em consideração o custo para sua implementação.

Neste trabalho, a implantação do particionamento nas tabelas do banco de dados não depende somente do seu benefício e do seu custo. Pretende-se que uma comparação entre o benefício da estratégia de particionamento e da estratégia de índices seja realizada. Esta comparação deverá responder à pergunta *Implantar ação de sintonia fina?*

4.2 Particionamento hipotético

A capacidade intrínseca da heurística HISAI (apêndice B) de calcular o benefício de índices hipotéticos e, portanto, determinar o valor total do benefício de uma dada configuração, permite a ela decidir pela criação ou não de tais estruturas de acesso.

Para avaliar as diferentes estratégias de particionamento, sugere-se que um conceito semelhante seja utilizado. Assim, o *particionamento hipotético* objetiva selecionar uma estratégia de particionamento para uma dada carga de trabalho com base no seu benefício total, sem implementar o particionamento propriamente dito. Para tanto, deve-se avaliar tanto o benefício da estratégia para cada consulta na carga de trabalho, como o custo associado à implementação do particionamento.

Seja W uma carga de trabalho submetida a um banco de dados e $B(Q_i)$ o benefício de um particionamento para a consulta $i \in W$. Para uma dada consulta i da carga de trabalho, seu plano de execução utiliza operadores de varredura em estruturas de acesso, a saber, uma tabela ou um índice. Quando um índice não é utilizado, tem-se um operador do tipo *Sequential Scan* no plano de execução. Nota-se que quando uma tabela é particionada, a varredura ocorre somente na(s) partição(ões) da tabela que satisfaz(em) o predicado. Logo, o benefício $B(Q_i)$ de um particionamento para uma consulta assim formada pode ser calculado pela diferença entre o custo do *Sequential Scan* na tabela inteira e o custo do *Sequential Scan* na tabela particionada.

Por outro lado, quando um índice é utilizado, tem-se um operador do tipo *Index Scan* (ou *Index Seek*) no plano de execução. O benefício $B(Q_i)$ de um particionamento para uma consulta assim formada poderia ser calculado pela diferença entre o custo do *Index Scan* (ou *Index Seek*) no índice não-particionado e o custo do *Index Scan* (ou *Index Seek*) no índice particionado. Entretanto, como as seletividades dos predicados nesta tabela poderão sofrer uma diminuição, considera-se que o índice não será utilizado e, portanto, o pode ser calculado pela

diferença entre o custo do *Index Scan* (ou *Index Seek*) no índice não-particionado e o custo do *Sequential Scan* na tabela particionada.

Assim, para que o particionamento hipotético possa ser utilizado, é fundamental ser possível estimar o custo de uma varredura em uma estrutura particionada. Percebe-se que o mesmo princípio vale também para uma visão materializada, tratada como uma tabela, como também para um índice parcial, similar a um índice completo.

Seja *C(SS)* o custo do *Sequential Scan* em uma tabela, *C(IS)* o custo do *Index Scan* em um índice e $C(SSP_k)$ o custo do *Sequential Scan* em uma tabela hipoteticamente particionada através da chave de particionamento P_k. Considerando $n=1..t$ o número de varreduras contidas no plano de execução de Q_i, o benefício $B(Q_i)$ é o somatório dos benefícios alcançados para cada varredura:

SE operador = Sequential Scan
$B(Q_i) = B(Q_i) + \sum_{n=1..t} (C(SS_n) - C(SSP_{kn}))$
SENÃO
$B(Q_i) = B(Q_i) + \sum_{n=1..t} (C(IS_n) - C(SSP_{kn}))$
FIM-SE

Considerando a criação de partições balanceadas, isto é, com uma distribuição uniforme do volume de dados em cada uma delas, efetuar uma varredura em uma estrutura em uma única partição significa percorrer o número total de linhas da tabela dividido pelo número total de partições.

Seja NP_k o número de partições gerado pelo particionamento de uma tabela usando a chave P_k e $NP(SSP_k)$ o número de partições acessadas em uma busca na estrutura particionada. Logo, o custo do *Sequential Scan* em uma estrutura particionada, *C(SSPk)*, pode ser assim estimado:

$C(SSP_k) = (C(SS) / NP_k) * NP(SSP_k)$

Da mesma forma, o custo do *Index Scan* num índice, *C(ISPk)*, poderia ser estimado como:

$$C(ISP_k) = (C(IS) / NP_k) * NP(ISP_k)$$

Ao analisar uma carga de trabalho, o benefício acumulado de um particionamento hipotético *AB(W)* será, portanto, o somatório dos benefícios alcançados em cada consulta da carga de trabalho multiplicados pela frequência de cada consulta.

$$AB(W) = \sum(B(Q_i) \times F(Q_i))$$

Duas considerações fundamentais são necessárias na implementação do particionamento hipotético. Primeiro, a fragmentação horizontal derivada gera um particionamento em cadeia, que pode melhorar o desempenho das consultas que usam junção nessas tabelas e o benefício gerado pelo particionamento. Quando um particionamento hipotético é criado para uma determinada tabela, um particionamento hipotético para todas as tabelas que fazem parte da cadeia deve ser criado também.

Como exemplo, no banco de dados do TPC-H, o particionamento da tabela *orders* leva ao particionamento da tabela *lineitem*. Isto pode garantir que numa consulta com junção entre as duas tabelas, não somente as partições das tabelas que satisfazem os predicados sejam acessadas, mas também que somente as partições-filhas das partições-mães sejam acessadas.

Segundo, o particionamento não somente altera os volumes de dados a serem percorridos pelos operadores de um plano de execução, mas pode influenciar as decisões do otimizador na medida em que tais operadores são escolhidos com base nesses volumes de dados e nas estatísticas correspondentes. Ao alterar estratégias de junção, por exemplo, o benefício do particionamento deixa de ser tão somente a diferença no custo da varredura.

Com bases nestas duas considerações, o benefício do particionamento para uma consulta e, portanto, para a carga de trabalho, pode sofrer variações. Estas variações dependem fundamentalmente da implementação do particionamento no SGBD e pode ser necessário fazer ajustes nas estimativas.

Para calcular o benefício total de um particionamento hipotético, é preciso levar em consideração o custo para a implementação deste particionamento. Existem basicamente três custos distintos associados à implementação de um particionamento: o custo de criação das áreas de armazenamento das partições, o custo de implantação do esquema particionado e o custo de movimentação dos dados para as partições criadas.

O custo de *criação* das áreas de armazenamento depende do tipo de particionamento (lógico ou físico), número de partições e número de nós (servidores), além do tamanho de cada partição. Neste trabalho, considerando que o particionamento é lógico, ou seja, que as partições são criadas em um único servidor e na mesma área de armazenamento de tabelas, o custo de criação das partições pode ser desprezado.

O custo de *implantação* do esquema particionado é o custo de alteração do esquema para implementar o particionamento, isto é, o custo da execução de todos os comandos DDL para criação das tabelas particionadas e índices nas partições correspondentes.

O custo de *movimentação* dos dados é o custo das operações *insert* de todas as linhas das tabelas particionadas e não-particionadas em todas as partições, incluindo-se o custo de atualização dos índices, mais o custo de execução das operações *delete* de todas as linhas de todas as tabelas originais.

Portanto, o custo total para implementação do particionamento é dado conforme abaixo:

$$C(P) = C(P_{CRIAÇÃO}) + C(P_{IMPLANTAÇÃO}) + C(P_{MIGRAÇÃO})$$

4.3 Histograma de partições hipotéticas

Conforme visto no capítulo 2, seção 2.5, o histograma de uma coluna contém a distribuição dos valores encontrados na coluna. Esta distribuição está organizada em grupos delimitados por valores mínimo e máximo: são as faixas de valores contidas no histograma. Ao utilizar estas faixas para definir os valores para a chave de particionamento, respeitando o parâmetro de número de partições e assumindo que as estatísticas do banco de dados estejam atualizadas, garante-se o balanceamento das partições geradas, pois as distribuições encontradas nos histogramas dos SGBDs de mercado apresentam distribuições uniformes dos valores.

Nota-se, entretanto, que esta distribuição privilegia o número de linhas em cada partição e nenhuma relação guarda com a carga de trabalho submetida ao banco de dados. O balanceamento gerado por tal distribuição é um balanceamento do volume de dados e não da carga de trabalho, pois é possível que a frequência de consultas em algumas partições seja maior do que em outras e que haja consultas que necessitem acessar mais de uma partição para obter seus resultados.

Com o objetivo de analisar as frequências das consultas nas partições geradas pelo histograma de frequência, considera-se a geração de um *histograma das partições hipotéticas*, isto é, um histograma cujos valores de referência são as faixas de valores que definem as partições, conforme a figura 4.2.

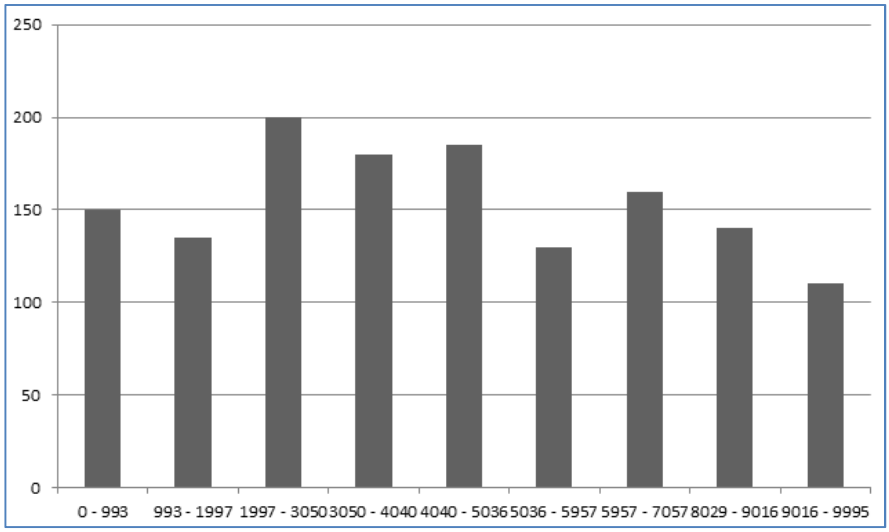

Figura 4.2: Histograma de frequência das consultas sobre as partições

Para os casos citados nos quais a frequência de consultas em algumas partições é maior do que em outras, pode-se ter um histograma conforme a figura 4.3.

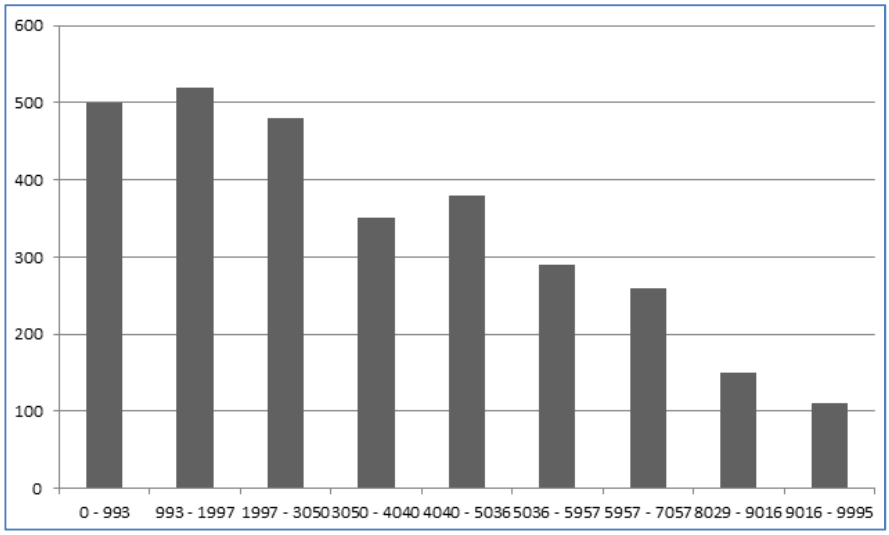

Figura 4.3: Histograma de frequência não balanceado sobre as partições

A alternativa de balanceamento com respeito à frequência das consultas é relevante para cenários onde o balanceamento de carga é indispensável. Em ambientes em nuvem, por exemplo, os requisitos impostos pelos níveis de acordo de serviço entre fornecedor e cliente podem fazer com que este balanceamento se torne imprescindível.

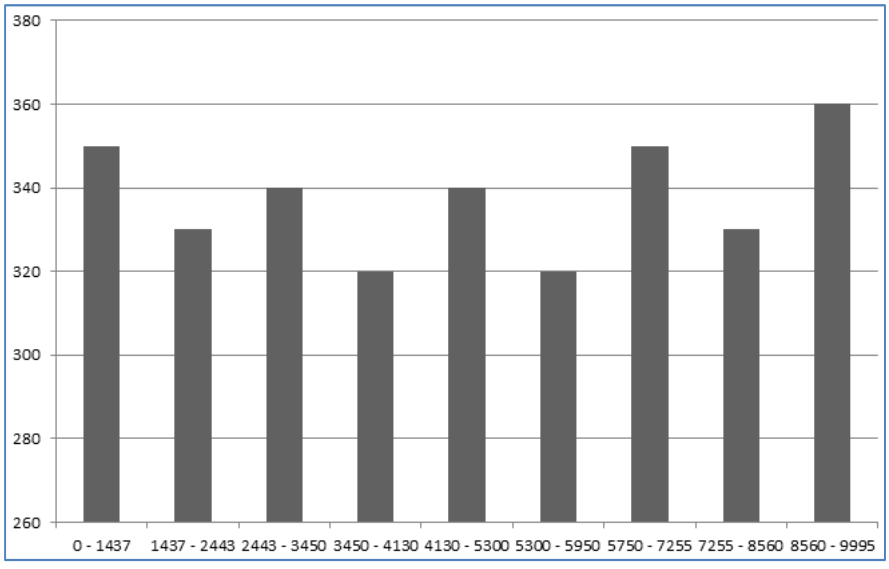

Figura 4.4: Histograma de frequência balanceado sobre as partições

4.4 Heurística HISAP

As heurísticas específicas para sintonia fina de bancos de dados relacionais tratam de explicitar as regras que levam a selecionar uma estratégia de *tuning* para uma determinada carga de trabalho, seja a criação de uma estrutura de acesso ou a utilização de uma técnica, como o particionamento do banco de dados. Na literatura acadêmica há uma série de trabalhos sobre este tema, como (Chaudhuri, 1997), (Lohman, 2000).

Em (Lohman, 2000) encontra-se a especificação da heurística SAEFIS – *Smart Column Enumaration for Index Scans* – que é a base para diversos outros trabalhos, como (Monteiro, 2008), (Carvalho, 2011) e (Almeida, 2013). Esta heurística determina que os índices hipotéticos sejam aqueles cujas chaves sejam as colunas usadas nas cláusulas *where* e *join*, e também as colunas usadas nas cláusulas *select*, *group by* e *order by*, mais a composição das colunas destes dois grupos.

Nestas mesmas pesquisas, uma outra heurística específica para otimização da carga de trabalho é utilizada para associar um valor de benefício a um determinado índice hipotético. Como explicado em (Lohman, 2000), o benefício de um índice é definido como a melhoria no tempo estimado de execução de todas as consultas que façam uso dele, multiplicado pela frequência com que cada consulta ocorre na carga de trabalho. Esta heurística é denominada *Heurística de Benefícios* (Costa, 2002), (Salles, 2004) e consiste basicamente em acumular um valor de benefício para cada índice hipotético que esteja sendo considerado para uma dada carga de trabalho submetida ao banco de dados.

Tendo como base a heurística de enumeração de índices candidatos e a heurística de benefícios, propõe-se nesta seção a heurística HISAP – Heurística Integrada para Seleção e Acompanhamento de Partições – e as adaptações necessárias para a seleção da estratégia de particionamento.

Com respeito à heurística de enumeração de índices candidatos, argumenta-se que ela é suficiente para a enumeração de chaves de particionamento candidatas. Como visto no capítulo 2, seção 2.4, a seletividade de um predicado contribui de forma determinante para as decisões do otimizador e as colunas candidatas à chave de particionamento são as participantes das cláusulas *where*, *group by* e *order by*, subconjunto daquelas selecionadas pela heurística SAEFIS (Lohman, 2000).

4.4.1 – Passo 1 – Lista de predicados

Da mesma forma que HISAI, HISAP analisa todas as consultas da carga de trabalho e os planos de execução correspondentes. Numa primeira fase a heurística constrói uma lista de predicados simples utilizados nas consultas e, para cada predicado, mantém um contador para registrar a frequência de acesso.

Note-se que os predicados são decompostos independentemente do tipo de operador utilizado no plano de execução. Na heurística HISAI importa analisar os predicados de operações *Sequential Scan*. São eles os candidatos a chaves de índices. Em HISAP, caso mantida a mesma restrição, a heurística estaria propondo o particionamento como estratégia secundária, para os casos onde os índices não seriam considerados benéficos a ponto de deixarem de ser hipotéticos.

Na heurística HISAP o particionamento é tratado como estratégia principal e todas as operações são analisadas, incluindo *Index Scan* e *Index Seek*, porque é possível que predicados para os quais o otimizador do SGBD selecione um índice sejam chaves candidatas a particionamento e de fato ofereçam um benefício maior com tal ação de sintonia fina. Não somente as consultas, mas também as atualizações (*update* e *delete*) são analisadas. Da mesma forma, os predicados simples são monitorados, juntamente com um contador para registrar a frequência de atualização.

Apresenta-se a seguir *IHSTIS_CPk – Candidate Partition Key Enumeration* – que mantém esta lista de predicados usados nas consultas e atualizações.

```
Require: a task T (<SQL expression Q, execution plan P, cost estimate CE_P>) as input
01: {Search the query execution plan HP looking for Scan or Seek operations}
02: Initialize the predicates list (L_predicates = ∅)
03: for each operation s ∈ HP do
04:   if s has fillters used in WHERE/JOIN clauses (Case 1) then
05:     for each attribute a_i used in WHERE/JOIN do
06:       if p_ai ∉ L_predicates then {p_ai is (table, simple predicate) }
07:         L_predicates = L_predicates ∪ {p_ai,0,0,0}
08:       end if
09:       if task is a SELECT
10:         -- increment access frequency
11:       else if task is a UPDATE or a DELETE
12:         -- increment update frequency
13:       end if
14:     end for
15:   end if
16:   {Combine the columns not involved with a filter (Case 2)}
17:   for each attribute b_i used in SELECT that is used in GROUP/ORDER clauses do
18:     if p_b_i ∉ L_predicates then { p_bi is (table, simple predicate) }
19:       L_predicates = L_predicates ∪ {p_bi,0,0,0}
20:     end if
21:     if task is a SELECT
22:       -- increment access frequency
23:     else if task is a UPDATE or a DELETE
24:       -- increment update frequency
25:     end if
26:   end for
27: end for
```

Figura 4.5: Algoritmo IHSTIS_CPk

Na heurística proposta, somente predicados simples utilizados nas consultas são armazenados. Assim, caso haja predicados compostos (utilizando cláusulas *and* e *or*), estes são decompostos para que somente predicados simples sejam considerados.

Para cada predicado, a heurística também deve registrar o número de vezes em que foi referenciado em um *select* (*access frequency*), o número de vezes em que foi referenciado em *update* ou *delete* (*update frequency*) e o custo total de varredura da tabela com este predicado. Então, quando o predicado ainda não existe em $L_{predicates}$, este é adicionado com frequência 1 para os contadores de acesso e de atualização. As referências seguintes incrementam este valor.

4.4.2 – Passo 2 – Lista de chaves candidatas

Com base na lista de predicados, uma lista de chaves candidatas é construída, acumulando-se a frequência de acesso, a frequência de atualizações e o custo. Para cada chave candidata, *(access frequency – update frequency)* é calculado e, para todos os valores negativos, a entrada correspondente é removida da lista.

Em seguida, a lista é classificada em ordem decrescente de *scan cost*, o que garante tratar primeiramente os predicados com maior custo para a varredura das tabelas. A lista é percorrida e, para cada chave candidata, as subfases da seleção da estratégia de particionamento são executadas, onde são definidos o tipo de particionamento, o número de partições e o valor da chave para cada partição.

Ao gerar um particionamento hipotético para uma tabela, conforme visto na seção 4.2, são gerados os particionamentos das tabelas-filha. Assim, as chaves candidatas das tabelas-filha também devem ser removidas da lista ao término da geração do particionamento hipotético de uma tabela, como pode ser observado na figura 4.6 linha 16, iniciando um novo ciclo que analisa novos subconjuntos de tabelas do esquema atual do banco de dados.

Um número melhor de partições pode ser determinado pela aplicação do algoritmo BEA (*Bond Energy Algorithm*), que faz uso de uma matriz de afinidades para gerar novos grupos com afinidades maiores. Em (Valduriez, 2011), o algoritmo é utilizado para o particionamento vertical. Substituindo-se as *colunas* referenciadas nas consultas pelas *partições* acessadas nas consultas, obtem-se uma matriz de afinidades para o particinamento horizontal.

4.4.3 – Passo 3 – Estimativa de benefícios

Uma configuração P é composta pelas tabelas hipoteticamente particionadas no passo anterior. Neste passo, o benefício total desta configuração P é calculado.

Require: (<number_of_partitions, histogram_choice>) as input
01: Initialize the candidates list ($L_{candidates} = \emptyset$)
02: for each predicate p ϵ $L_{predicates}$ do
03: Let c be the column corresponding to predicate p
04: if c ∉ $L_{candidates}$ then
05: $L_{candidates} = L_{candidates} \cup$ {c, access frequency(c), update frequency(c), scan cost(c)}
06: else
07: Add access frequency(c), update frequency(c) and scan cost(c) to $L_{candidates}$ (c)
08: end if
09: end for
10: sort $L_{candidates}$ in descendant order of (access frequency – update frequency) and remove all entries with negative values
11: sort $L_{candidates}$ in descendant order of (scan cost)
12: for each candidate c in $L_{candidates}$
13: If <histogram_choice> is to use the access frequency histogram then create candidate column's access frequency histogram of c based on candidate column's histogram and $L_{predicates}$ (*)
14: According to <histogram_choice>, perform hypothetical partitioning of table(c) based on column's histogram or column's access frequency histogram, limited to <number_of_ partitions>
15: Perform hypothetical derived partitioning starting from table(c), updating configuration P
16: Delete from $L_{candidates}$ all candidates from table(c) and derived partitioning' tables
17: end for
18: Compute accumulated benefit of configuration P
19: Update P and AB(P) in LM

Figura 4.6: Algoritmo IHSTIS_CPk

O cálculo do benefício segue as regras definidas na seção 4.2 deste capítulo. Desta forma, todas as consultas da carga de trabalho são acessadas para que uma estimativa de benefício seja calculada para cada consulta. O benefício total é o somatório dos benefícios de cada consulta, menos o custo para implementação do particionamento.

4.4.4 – Passo 4 – Implantação do particionamento

A integração entre as heurísticas HISAP e HISAI prevê que os benefícios das configurações C_P e C_I sejam comparadas para que somente uma delas seja aplicada ao esquema.

```
01: if (B C̄ > B_C x k) and (B C̄ > B_P x k_P) then
02:     for all index i ∈ C̄ do
03:         if i ∈ L_Hypothetical then
04:             Create physically the index i
05:             state(i) ← real
06:         end if
07:     end for
08:     for all index i ∈ L_Real do
09:         if i ∉ C̄ then
10:             Drop index i
11:             state(i) ← hypothetical
12:             if AB_i > 0 then
13:                 AB_i = 0
14:             end if
15:         end if
16:     end for
17: end if
18: for all index i ∈ L_ind_frag do
19:     Reindex i
20: end for
21: if (B_P x k_P > B_C x k) and (B_P x k_P > B_C̄) then
22:     create tables' partitions according to P
23: end if
```

Figura 4.7: Algoritmo IHSTIS_CPk

Os benefícios acumulados da configuração de índices C_I e da configuração de particionamento C_P são submetidos a uma transformação pelos fatores k e k_P, respectivamente. Estes fatores podem ser utilizados pelo DBA para ajustar os benefícios gerados pelas estratégias de sintonia fina (Luhring et al, 2007), (Monteiro, 2012).

4.5 Resumo do capítulo

Este capítulo apresentou o problema do particionamento e propôs um modo para endereçá-lo, através de um ciclo que se inicia após a fase de observação da carga de trabalho e que seleciona uma estratégia de particionamento para as tabelas do banco de dados. A partir desta estratégia e da carga de trabalho observada, o benefício do particionamento é estimado, levando-se em consideração o custo para sua implementação.

Foi apresentado também o conceito de particionamento hipotético, que objetiva selecionar uma estratégia de particionamento para uma dada carga de trabalho com base no seu benefício total, sem implementar o particionamento propriamente dito. Isto inclui estimar o benefício através dos custos de varreduras em estruturas particionadas.

No escopo da seleção da estratégia de particionamento está a definição das chaves de particionamento e dos valores da chave em cada partição. Foi mostrado que os histogramas existentes nos SGBDs podem ser utilizados para gerar as partições, privilegiando o balanceamento do volume de dados ou o balanceamento da carga de trabalho, através do histograma de partições hipotéticas.

No final do capítulo é apresentada a heurística HISAP, na qual o particionamento é tratado como estratégia principal de sintonia fina. São listados os passos que guiam a execução do algoritmo, a saber, lista de predicados, lista de chaves candidatas, estimativa de benefícios e implantação do particionamento.

A integração entre as heurísticas HISAP e HISAI é reforçada, para que somente uma das configurações (C_P ou $C_{I)}$ seja aplicada ao esquema.

5 Resultados experimentais

A seguir apresenta-se os resultados encontrados com a implementação da estratégia de particionamento como ação de sintonia fina, utilizando a heurística HISAP mostrada no capítulo 4, em modo semi-automático.

Com o objetivo de realizar testes adequados e coletar os resultados relevantes para a pesquisa, utilizou-se o *benchmark* de referência TPC-H (TPC, 2016). As bases de dados, com volumes de 1GB, 10 GB e 50 GB, foram instaladas sob os SGBDs *Oracle 12c* e *PostgreSQL 9.6.1*. O *Oracle 12c* foi instalado sob as configurações padrão em um servidor Intel Core i7 3960 com 32 GB de memória RAM e HD de 1 TB, rodando sistema operacional Windows 10 64 bits. Já o *PostgreSQL 9.6.1* foi instalado em um servidor Intel Core i3 com 4 GB de memória RAM e HD de 500 GB, rodando sistema operacional Windows 7. No apêndice A pode ser encontrado o esquema do banco de dados TPC-H.

Nos dois SGBDs, foram criados três bancos de dados, *tpch*, *tpch-i* e *tpch-p*, para coletar os resultados das consultas com, respectivamente, nenhuma estratégia de sintonia fina, índices como ação de sintonia fina e particionamento como ação de sintonia fina. Foram utilizados 3 cenários de teste com características distintas para avaliação das heurísticas de seleção da estratégia de particionamento. Em todos os cenários são utilizadas as consultas fornecidas pelo próprio *benchmark*, submetidas aos SGBDs através de *scripts Powershell*. Arbitrariamente, o número máximo de partições é *cinco* e a opção de histograma é *balanceado por volume*.

A métrica usada para avaliação dos resultados é o tempo de execução da consulta. Os planos gerados pelos otimizadores de SGBDs apresentam outras métricas observadas frequentemente por DBAs, como custo de CPU, custo de E/S, custo total e tempo estimado de execução. Entretanto, o objetivo da sintonia fina é diminuir o tempo total da execução das consultas e, sendo assim, esta é a métrica usada nesta pesquisa. A contagem do tempo é realizada pelo próprio SGBD e o tempo de execução de cada consulta apresentada a seguir é a média de três execuções, a partir da segunda execução. Todos os planos de execução estão mostrados no apêndice D.

5.1 Cenário 1

Este cenário de teste é composto pela execução de variações da consulta 1. Os planos de execução e os resultados mostrados foram obtidos no PostgreSQL.

```
SELECT L_RETURNFLAG, L_LINESTATUS, SUM(L_QUANTITY) AS SUM_QTY,
SUM(L_EXTENDEDPRICE) AS SUM_BASE_PRICE,
SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS SUM_DISC_PRICE,
SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)*(1+L_TAX)) AS SUM_CHARGE,
AVG(L_QUANTITY) AS AVG_QTY, AVG(L_EXTENDEDPRICE) AS AVG_PRICE,
AVG(L_DISCOUNT) AS AVG_DISC, COUNT(*) AS COUNT_ORDER
FROM LINEITEM WHERE L_SHIPDATE <= '1993-01-01'
GROUP BY L_RETURNFLAG, L_LINESTATUS
ORDER BY L_RETURNFLAG, L_LINESTATUS
```

O plano de execução em *tpch* mostra um operador *Sequential Scan* na tabela *lineitem*, utilizando o filtro apropriado na coluna *l_shipdate*.


```
[
  {
    "Execution Time": 3567.668,
    "Planning Time": 0.15,
    "Plan": {
      "Node Type": "Sort",
      "Sort Key": ["l_returnflag", "l_linestatus"],
      "Plans": [
        {
          "Node Type": "Aggregate",
          "Plans": [
            {
              "Node Type": "Seq Scan",
              "Relation Name": "lineitem",
              "Filter": "(l_shipdate <= '1993-01-01')",
              "Parallel Aware": false,
              "Total Cost": 197220.14,
            }
          ],
          "Group Key": ["l_returnflag", "l_linestatus"],
          "Total Cost": 227539.78
        }
      ],
      "Total Cost": 227539.88
    },
  }
]
```


Figura 5.1: Cenário 1

5.1.1 Benefício da configuração de índices

O maior custo neste plano de execução está no *Sequential Scan* da tabela *lineitem*, conforme pode ser visto na figura 5.1. Uma ação de sintonia fina sugerida por HISAI para esta consulta pode ser a criação de um índice *btree* nas colunas (*l_shipdate*, *l_returnflag*, *l_linestatus*), justamente as colunas que aparecem nas cláusulas *where* e *group By*. O otimizador seleciona este índice quando a seletividade é suficientemente alta e retorna o plano de execução:


```
[
  {
    "Execution Time": 3362.235,
    "Planning Time": 2.527,
    "Plan": {
      "Sort Key": ["l_returnflag", "l_linestatus"],
      "Node Type": "Sort",
      "Plans": [
        {
          "Node Type": "Aggregate",
          "Plans": [
            {
              "Node Type": "Bitmap Heap Scan",
              "Plans": [
                {
                  "Node Type": "Bitmap Index Scan",
                  "Index Name": "i_lineitem_shipdate_returnflag_linestatus",
                  "Index Cond": "(l_shipdate <= '1993-01-01')",
                  "Parallel Aware": false,
                  "Total Cost": 14042.38,
                }
              ],
              "Relation Name": "lineitem",
              "Parallel Aware": false,
              "Total Cost": 146003.7
            }
          ],
          "Group Key": ["l_returnflag", "l_linestatus"],
          "Parallel Aware": false,
          "Total Cost": 176414.26
        }
      ],
      "Parallel Aware": false,
      "Total Cost": 176414.36
    },
  }
]
```

Figura 5.2: Cenário 1 – Configuração de índices

5.1.2 Benefício da configuração de particionamento

Ao considerar a estratégia de particionamento, as colunas candidatas que farão parte de $L_{candidates}$ em HISAP são *l_shipdate*, *l_returnflag* e *l_linestatus*, sendo que *l_returnflag* e *l_linestatus* tem maior frequência de acesso que *l_shipdate*.

A heurística aqui proposta seleciona a chave candidata *l_shipdate* e obtém o histograma correspondente à coluna nas estatísticas do banco de dados. Considerando a distribuição uniforme dos valores utilizados nos predicados da consulta e o tipo *range*, são geradas cinco partições. O plano de execução gerado pelo otimizador para esta configuração está na figura 5.3:


```
[
  {
    "Execution Time": 2973.509,
    "Planning Time": 0.325,
    "Plan": {
      "Startup Cost": 128706.08,
      "Plans": [
        {
          "Plans": [
            {
              "Plans": [
                {
                  "Filter": "(l_shipdate <= '1993-01-01')",
                  "Node Type": "Seq Scan",
                  "Relation Name": "lineitem_p1",
                  "Total Cost": 36787.31
                }
              ],
            }
          ],
          "Sort Key": [
            "lineitem.l_returnflag", "lineitem.l_linestatus"
          ],
          "Total Cost": 130596.35
        }
      ],
      "Group Key": [
        "lineitem.l_returnflag", "lineitem.l_linestatus"
      ],
      "Total Cost": 161940.67
    },
  }
]
```


Figura 5.3: Cenário 1 – Configuração de particionamento

Nota-se que o *Sequential Scan* agora é executado somente sobre a partição que contém as linhas correspondentes (*lineitem_p1*), resultando em um custo para a operação bem menor que o *Sequential Scan* na tabela inteira. O gráfico a seguir mostra os resultados encontrados neste experimento:

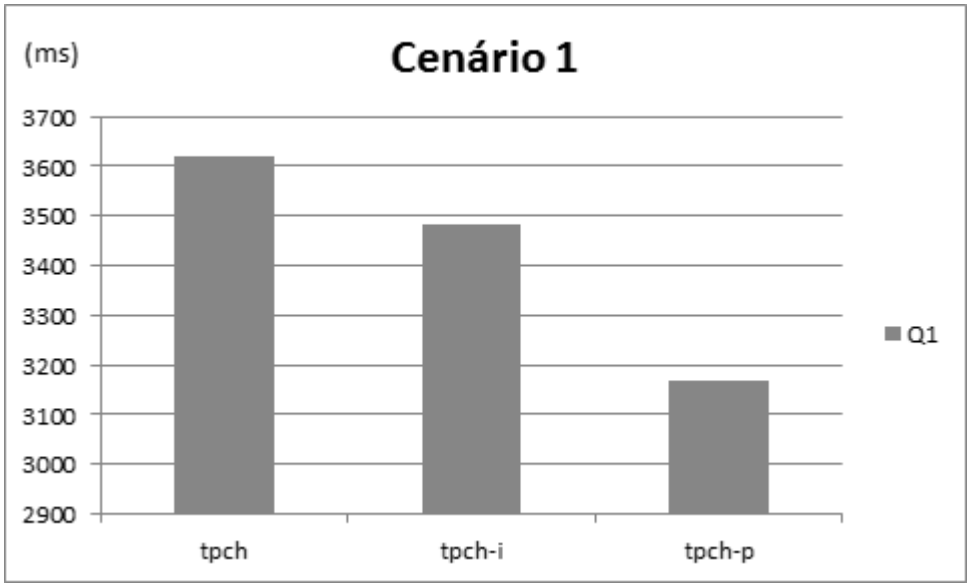

Figura 5.4: Cenário 1 - Resultados

5.1.3 Aumento do número de partições acessadas

Ao variar-se o predicado da consulta, diferentes partições poderão ser acessadas. Quanto mais abrangente for a faixa de valores contemplada pelo predicado da consulta, maior o número de partições a serem acessadas e, consequentemente, maiores o custo da consulta e o tempo necessário para executá-la.

Por exemplo, com o filtro *where l_shipdate ≤ '1996-01-01'*, o *Sequential Scan* é executado sobre as partições que contém as linhas correspondentes, neste caso 3 partições.

Por outro lado, ao aumentar-se a faixa de valores do predicado, sua seletividade é diminuída, o que acarreta diretamente na diminuição da probabilidade de o otimizador selecionar um índice para seu plano de execução. O plano de execução desta mesma consulta no banco de dados *tpch-i* não seleciona o índice anteriormente utilizado nas colunas (*l_shipdate*, *l_returnflag*, *l_linestatus*), porque ao aumentar a faixa de datas que satisfaz o predicado, este passa a ter uma

seletividade mais baixa tal que o otimizador não seleciona o índice. Ao contrário, observa-se um *Sequential Scan* na tabela inteira.

Observa-se, portanto, que o crescimento do número de partições acessadas por uma consulta está associado à diminuição da seletividade do predicado e, consequência direta, a diminuição da probabilidade do uso de índices.

Com base no estudo do comportamento da consulta Q1 quando varia-se o predicado da mesma, forçando a varredura em n=1, 2, 3, 4 e 5 partições da tabela *lineitem*, o gráfico abaixo mostra o evolução do custo da consulta nos bancos de dados *tpch*, *tpch-i* e *tpch-p*.

Em n=1 o custo em C_P (configuração de particionamento em HISAP) é bem menor. Para n=2, com o aumento da faixa de valores que resulta no acesso a duas partições da tabela, o custo em C_I é quase igual ao aferido em C, sendo o custo em C_P ainda bem menor. Para n=3, o índice já não é mais selecionado e C_P se mantém bem menor, resultado do acesso a 3 partições da tabela. Daí por diante o custo cresce até se igualar ao custo da varredura completa da tabela em n=5.

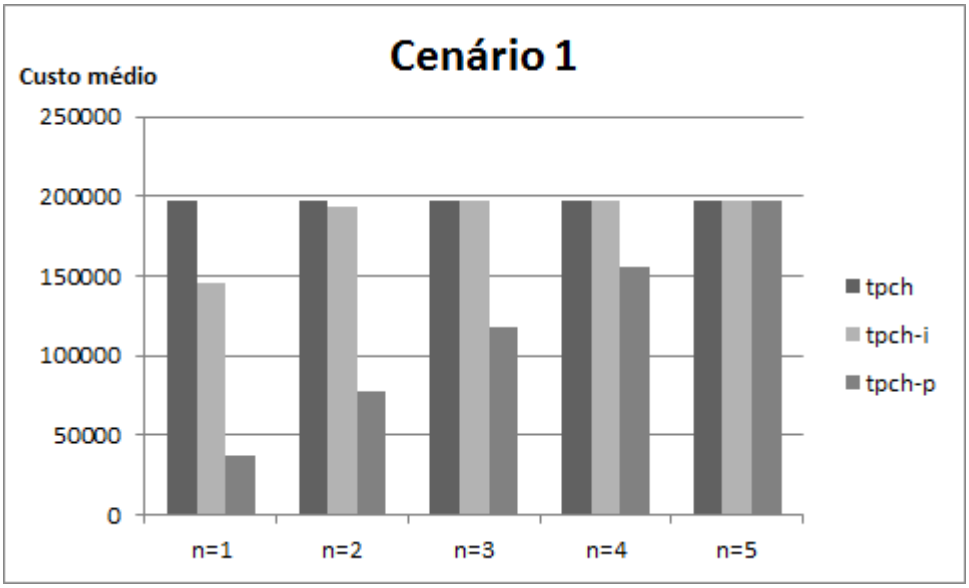

Figura 5.5: Cenário 1 – Aumento do número de partições acessadas

5.2 Cenário 2

O cenário 2 é composto pela execução de variações das consultas 3 e 8, sendo que a frequência f(Q8) é igual a *oito* vezes a frequência f(Q3).

```
SELECT L_ORDERKEY,
SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE,
O_ORDERDATE, O_SHIPPRIORITY
FROM CUSTOMER, ORDERS, LINEITEM
WHERE C_MKTSEGMENT = 'BUILDING'
AND C_CUSTKEY = O_CUSTKEY
AND L_ORDERKEY = O_ORDERKEY AND
O_ORDERDATE < '1994-03-15'
AND L_SHIPDATE > '1999-03-15'
GROUP BY L_ORDERKEY, O_ORDERDATE, O_SHIPPRIORITY
ORDER BY REVENUE DESC, O_ORDERDATE
LIMIT 10
```

A consulta 3 acima utiliza dois predicados, a saber, nas colunas *o_orderdate* e *l_shipdate*. Ainda, faz referência no *where* uma vez a cada uma das colunas especificadas, incrementa esta frequência para *l_orderkey*, *o_orderdate*, e *o_shippriority* na cláusula *Group By* e incrementa mais uma vez a frequência de *o_orderdate* na cláusula *Order By*.

Sua execução mostra, inicialmente, um *Sequential Scan* em *customer* e um *Sequential Scan* em *orders* (D.4). Ao primeiro é aplicado um filtro para *c_mktsegment = 'BUILDING'* e ao segundo é aplicado um filtro para *o_orderdate<'1994-03-15'*. Nas duas saídas é aplicado uma junção usando *(orders.o_custkey = customer.c_custkey).* Ao analisar-se o plano de execução da consulta em *tpch*, observa-se que o maior custo está associado ao *Sequential Scan* da tabela *orders*, no qual é aplicado o filtro em *orderdate*. O *Sequential Scan* em *orders* possui *TotalCost=46576*, enquanto o *Sequential Scan* em *lineitem* possui *TotalCost=1.57*.

Em seguida um *Sequential Scan* é executado sobre *lineitem* com o filtro *l_shipdate > '1999-03-15'* e uma junção é realizada com o resultado anterior usando (*lineitem.l_orderkey = orders.o_orderkey*), sendo o agrupamento e a classificação realizados na última fase. O tempo médio de 3 execuções para esta consulta foi 1895,256 milisegundos.

Ao executar-se a consulta em *tpch-i*, nota-se (D.5) que o otimizador seleciona os índices *i_orders_orderdate* e *i_lineitem_shipdate*, porque as seletividades dos predicados correspondentes são altas o suficiente para fazer com que a opção que utilize tais estruturas de acesso tenha um custo menor. Assim, um *Index Scan* é realizado em ambas as tabelas, como mostra o plano de execução. Outro índice utilizado é *i_customer_mktsegment*, fazendo com que o custo deste plano seja menor que o custo do plano gerado em *tpch*. O tempo médio de 3 execuções para esta consulta foi 0,103 milisegundos.

A consulta 8 abaixo incrementa ainda mais a frequência em *o_orderdate* ao acessá-la duas vezes na claúsula *where*. Há predicados em *r_name* e *p_type*. Os demais predicados realizam as junções e cada chave candidata é acessada uma única vez.

```
SELECT O_YEAR, SUM(CASE WHEN NATION = 'BRAZIL' THEN VOLUME
ELSE 0 END)/SUM(VOLUME) AS MKT_SHARE
FROM (SELECT O_ORDERDATE AS O_YEAR, L_EXTENDEDPRICE*(1-
L_DISCOUNT) AS VOLUME, N2.N_NAME AS NATION
FROM PART, SUPPLIER, LINEITEM, ORDERS, CUSTOMER, NATION N1,
NATION N2, REGION
WHERE P_PARTKEY = L_PARTKEY AND S_SUPPKEY = L_SUPPKEY
AND L_ORDERKEY = O_ORDERKEY AND O_CUSTKEY = C_CUSTKEY
AND C_NATIONKEY = N1.N_NATIONKEY
AND N1.N_REGIONKEY = R_REGIONKEY AND R_NAME = 'AMERICA'
AND S_NATIONKEY = N2.N_NATIONKEY
AND O_ORDERDATE BETWEEN '1995-01-01' AND '1996-12-31'
AND P_TYPE= 'ECONOMY ANODIZED STEEL') AS ALL_NATIONS
GROUP BY O_YEAR
ORDER BY O_YEAR
```

Esta consulta em *tpch* é resolvida com varreduras sequenciais e a aplicação dos filtros apropriados durante as varreduras (D.7). As linhas de *orders* e *lineitem* são unidas através de um *Nested Loop Join* e o maior custo no plano de execução está relacionado ao *Sequential Scan* de *orders*.

A última varredura está em *customer* e na sequencia o agrupamento e a ordenação. O tempo médio de 3 execuções para esta consulta foi 3468,662 milisegundos.

Em *tpch-i* o índice *i_orders_orderdate* é selecionado para a varredura de *orders*, enquanto para *lineitem* a varredura sequencial é preferida (D.8). O tempo médio de 3 execuções para esta consulta foi 3223,417 milisegundos.

A tabela abaixo mostra os custos associados aos *Sequential Scan* e *Index Scan* em cada coluna candidata:

Coluna	Q3		Q8	
	tpch	tpch-i	tpch	tpch-i
c_mktsegment	0.47	0.47	-	-
o_orderdate	8.16	8.45	50326 (x8)	44208 (x8)
l_shipdate	197283	8.45	-	-
r_name	-	-	12.13 (x8)	3.31 (x8)
p_type	-	-	0.44 (x8)	0.44 (x8)

Tabela 5.1: Cenário 2

O maior custo entre os predicados em ambas as consultas está associado à varredura da tabela *orders*, com filtro na coluna *orderdate*. Em ambos os planos esta junção é implementada por um operador *Nested Loop Join*.

Seleciona-se portanto o particionamento horizontal na coluna *o_orderdate* da tabela *orders*. As faixas de valores são construídas a partir do histograma da coluna para cinco partições.

5.2.1 Fragmentação horizontal derivada

No cenário descrito acima, o particionamento ocorre a partir da tabela particionada *orders*, na coluna *o_orderdate*. Sua chave primária é simples em *o_orderkey* e possui uma tabela-filha, *lineitem*, cuja chave primária é composta por (*l_orderkey*, *l_linenumber*). Esta tabela-filha também deverá ser particionada em *l_orderkey*, mantendo juntas as linhas da tabela-filha correspondentes à linha proprietária. Logo, a fragmentação derivada parte de *orders* e atinge *lineitem*.

Em *tpch-p* a execução da consulta 3 apresenta varreduras completas somente nas partições da tabela *orders* que satisfazem o predicado e em todas as partições de *lineitem* (D.6). Ao realizar os *Sequential Scan* nas partições, tanto para a tabela *orders* quanto para a tabela *lineitem*, o plano mostra um *Nested Loop Join* para juntar as linhas das duas tabelas baseado no critério *orders.o_orderkey* = *lineitem.l_orderkey*.

Nota-se que o otimizador não percorreu somente as partições da tabela *lineitem* que correspondem às partições da tabela *orders*. Isto era o esperado, pois a tabela *lineitem* foi particionada por referência à tabela *orders*. Entretanto, como o particionamento por referência não é explícito, ou seja, está implícito na implementação, o SGBD não percebe que poderia percorrer somente as partições 1 a 3 de *lineitem*. O tempo médio de 3 execuções para esta consulta foi 1526,891 milisegundos.

A execução da consulta 8 em *tpch-p* também apresenta o mesmo comportamento para as varreduras sequenciais das partições de *lineitem* (D.9). Para *orders*, somente as partições que satisfazem o predicado em *orderdate* são acessadas.

O tempo médio de 3 execuções para esta consulta foi 2569,979 milisegundos.

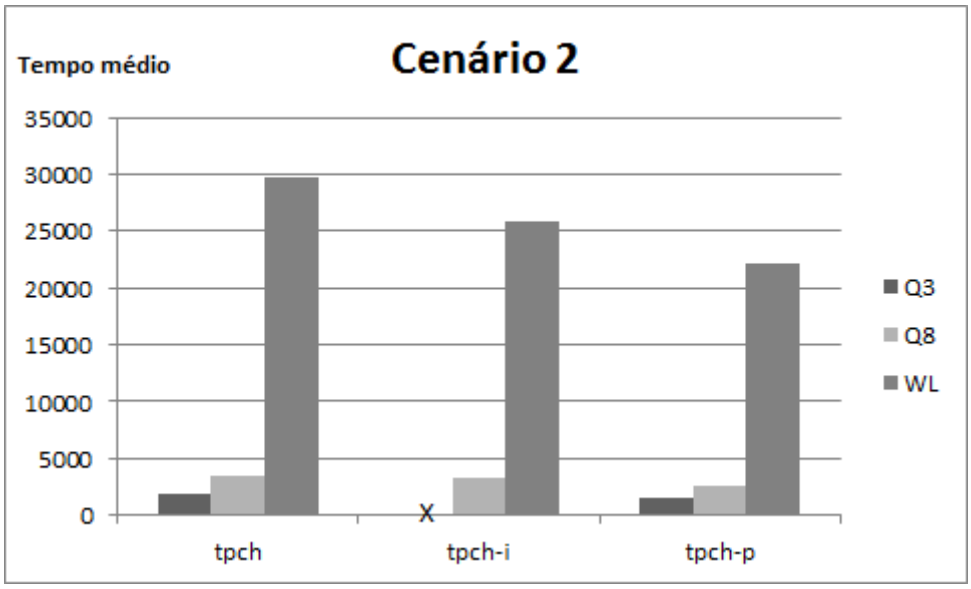

Figura 5.6: Cenário 2 – Resultados

5.2.2 Particionamento nativo no Oracle 12c

No Oracle 12c, em *tpch* (figura 5.7), para a execução da consulta 3, o otimizador escolhe executar um *full scan* na tabela *orders*, filtrando as linhas que satisfazem a condição *o_orderdate<'1995-03-15'*, da mesma forma que executa um *full scan* na tabela *lineitem*, filtrando as linhas que satisfazem a condição *l_shipdate>'1995-03-15*'. As linhas provenientes das duas tabelas são unidas por um *Hash Join* por *l_orderkey=o_orderkey*.

Uma varredura completa é realizada em *customer* e as linhas que satisfazem *c_mktsegment="building"* são unidas por um *Hash Join* com as linhas provenientes da junção anterior por *c_custkey=o_custkey*.

Estas linhas então são agrupadas, ordenadas e suas colunas selecionadas. Ao executar esta mesma consulta em *tpch-p*, sob a estratégia de particionamento selecionada, a saber, *orders* particionada por *orderdate* e *lineitem* particionada por referência, obtem-se o plano apresentado na figura 5.7. Nele, verifica-se um *full scan* nas partições 1 a 3 da tabela *lineitem*.

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			18692
PX COORDINATOR			
PX SEND	:TQ10003	QC (ORDER)	18692
SORT		ORDER BY	18692
PX RECEIVE			18692
PX SEND	:TQ10002	RANGE	18692
HASH		GROUP BY	18692
PX RECEIVE			18692
PX SEND	:TQ10001	HASH	18692
HASH		GROUP BY	18692
HASH JOIN			18689
Access Predicates			
C_CUSTKEY=O_CUSTKEY			
TABLE ACCESS	CUSTOMER	FULL	516
Filter Predicates			
C_MKTSEGMENT='BUILDING'			
HASH JOIN			18173
Access Predicates			
L_ORDERKEY=O_ORDERKEY			
JOIN FILTER	:BF0000	CREATE	3615
PX RECEIVE			3615
PX SEND	:TQ10000	BROADCAST	3615
PX BLOCK		ITERATOR	3615
TABLE ACCESS	ORDERS	FULL	3615
Filter Predicates			
O_ORDERDATE			
JOIN FILTER	:BF0000	USE	14554
PX BLOCK		ITERATOR	14554
TABLE ACCESS	LINEITEM	FULL	14554
Filter Predicates			
AND			
L_SHIPDATE>TO_DATE(' 1995-03-15 00:00:00', 'syyyy-mm-dd hh24:mi:ss')			
SYS_OP_BLOOM_FILTER(:BF0000,L_ORDERKEY)			

Figura 5.7: Cenário 2 – Particionamento nativo

Estas são as partições que satisfazem o predicado correspondente à tabela *orders* mas, como *lineitem* está particionada por referência ao particionamento de *orders*, são percorridas as partições de *lineitem* que satisfazem a condição *o_orderdate<'1995-03-15'*, filtrando as linhas que satisfazem a condição *l_shipdate>'1995-03-15*'.

Da mesma forma que *lineitem*, as partições 1 a 3 da tabela *orders* são percorridas, tratando-se de uma varredura completa com filtro em *orderdate*. Uma varredura completa é realizada em *customer* e as linhas que satisfazem *c_mktsegment="building"* são unidas por um *Hash Join* com as linhas provenientes da junção anterior por *c_custkey=o_custkey*. Outro *Hash Join* é realizado entre as linhas desta última junção com as linhas retornadas de *lineitem*. Estas linhas então são agrupadas, ordenadas e suas colunas selecionadas.

OPERATION	OBJECT_NAME	OPTIONS	COST	PARTITION_START	PARTITION_STOP
SELECT STATEMENT			17225		
PX COORDINATOR					
PX SEND	:TQ10003	QC (ORDER)	17225		
SORT		ORDER BY	17225		
PX RECEIVE			17225		
PX SEND	:TQ10002	RANGE	17225		
HASH		GROUP BY	17225		
PX RECEIVE			17225		
PX SEND	:TQ10001	HASH	17225		
HASH		GROUP BY	17225		
HASH JOIN			17224		
Access Predicates					
L_ORDERKEY=O_ORDERKEY					
JOIN FILTER	:BF0000	CREATE	2712		
PX RECEIVE			2712		
PX SEND	:TQ10000	BROADCAST	2712		
HASH JOIN			2712		
Access Predicates					
C_CUSTKEY=O_CUSTKEY					
JOIN FILTER	:BF0001	CREATE	516		
TABLE ACCESS	CUSTOMER	FULL	516		
Filter Predicates					
C_MKTSEGMENT='BUILDING'					
JOIN FILTER	:BF0001	USE	2195		
PX BLOCK		ITERATOR	2195	1	3
TABLE ACCESS	ORDERS	FULL	2195	1	3
Filter Predicates					
AND					
O_ORDERDATE					
SYS_OP_BLOOM_FILTER(:BF0001,O_CUSTKEY)					
JOIN FILTER	:BF0000	USE	14509		
PX BLOCK		ITERATOR	14509	1	3
TABLE ACCESS	LINEITEM	FULL	14509	1	3
Filter Predicates					
AND					
L_SHIPDATE>TO_DATE(' 1995-03-15 00:00:00', 'syyyy-mm-dd hh24:mi:ss')					
SYS_OP_BLOOM_FILTER(:BF0000,L_ORDERKEY)					

Figura 5.8: Cenário 2 – Particionamento nativo

No Oracle 12c há uma implementação nativa do particionamento horizontal, incluindo uma cláusula para o particionamento por referência, *Partition By Reference*. Isto torna explícito o arranjo das partições das tabelas e permite ao otimizador percorrer, nas tabelas-filhas, as mesmas partições das tabelas-mães.

5.2.3 Aumento do volume de dados

Qual o aumento do benefício do particionamento quando o volume de dados aumenta? Para responder a esta pergunta, os mesmos testes do Cenário 2 (onde não houve ganho expressivo com a estratégia) foram realizados no PostgreSQL com os tamanhos 10GB e 50GB. Os resultados são mostrados na figura 5.9.

Para Q8 o ganho de benefício relativo da estratégia de particionamento em relação à estratégia de índice foi de 21,67% no crescimento a 10GB e de 24,85% no crescimento a 50GB.

Para a carga de trabalho (WL) o ganho de benefício relativo da estratégia de particionamento em relação à estratégia de índice foi de 16,58% no crescimento a 10GB e de 19,61% no crescimento a 50GB.

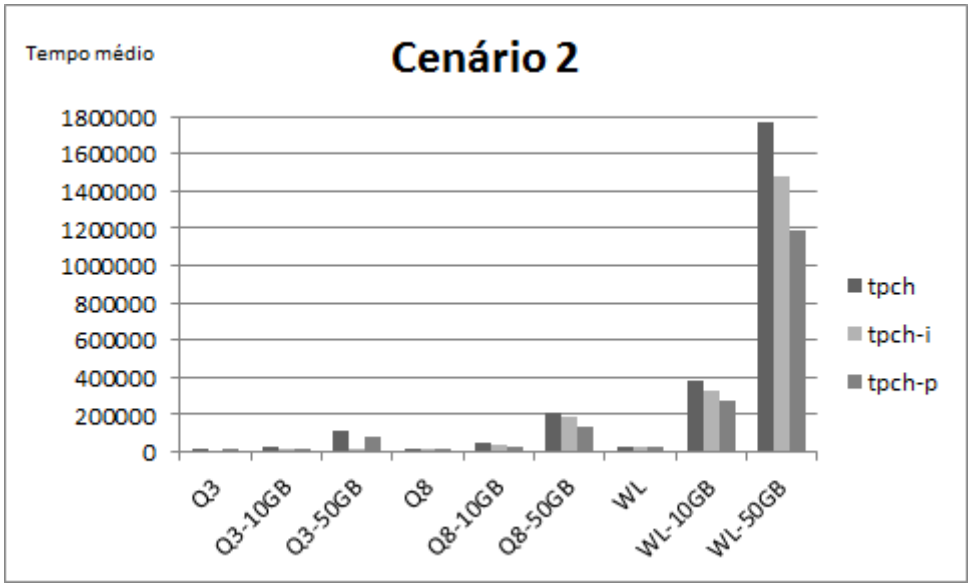

Figura 5.9: Cenário 2 - Aumento do volume de dados

Conclui-se então que, em um cenário onde existe uma configuração de particionamento que ofereça um benefício, quanto maior o volume de dados, maior o benefício. Assim, para bases de dados onde já existe um particionamento implementado, o crescimento das mesmas tende a aumentar o benefício do particionamento.

5.3 Cenário 3

Este cenário é composto pela execução de variações das consultas 5 e 12, reproduzidas abaixo. A frequência das consultas é a mesma.

```
SELECT N_NAME, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE
FROM CUSTOMER, ORDERS, LINEITEM, SUPPLIER, NATION, REGION
WHERE C_CUSTKEY = O_CUSTKEY
AND L_ORDERKEY = O_ORDERKEY
AND L_SUPPKEY = S_SUPPKEY
AND C_NATIONKEY = S_NATIONKEY
AND S_NATIONKEY = N_NATIONKEY
AND N_REGIONKEY = R_REGIONKEY
AND R_NAME = 'ASIA'
AND O_ORDERDATE >= '1994-01-01'
AND O_ORDERDATE < '1995-01-01'
GROUP BY N_NAME
ORDER BY REVENUE DESC
```

A consulta 5 faz a junção de duas tabelas fato, *orders* e *lineitem*, e quatro tabelas dimensão, *customer*, *supplier*, *nation* e *region* e executa uma agregação e uma ordenação. O plano de execução desta consulta em *tpch* (D.10) inicialmente mostra varreduras completas nas tabelas *region* e *nation* e um *hash join* entre as tabelas. À esta saída é aplicado uma junção com o retorno do *Sequential Scan* em *customer*. Em seguida, à esta saída é aplicado uma junção com o retorno do *Sequential Scan* em *orders* usando *(orders.o_custkey = customer.c_custkey)* e um *Hash Inner Join* que realiza a junção de *orders* com *lineitem* em *orderkey*. E uma última junção com a saída da varredura de *supllier* usando *(lineitem.l_suppkey = supplier.s_suppkey).* Na sequencia faz-se o agrupamento e a ordenação. O tempo médio de 3 execuções para esta consulta foi 1550.944 milisegundos.

Ao executar-se a consulta em *tpch-i*, nota-se (D.11) que o otimizador efetua varreduras sequenciais em *region*, *nation* e *customer*, entretanto utiliza o índice *i_orders_orderdate* o que, de fato, diminui o custo na junção imediatamente posterior, comparado com o plano anterior. O tempo médio de 3 execuções para esta consulta foi 1144.654 milisegundos.

Operação	Q5	
	tpch	tpch-i
(5) Hash Join (l_suppkey, s_suppkey; c_nationkey, s_nationkey)	62078.12	47718.58
(4) Nested Loop Join (orders, lineitem)	61554.36	47194.82
(3) Hash Join (o_custkey, c_custkey)	57014.43	42654.9
(2) Hash Join (c_nationkey, n_nationkey)	5821.82	5821.82
(1) Hash Join (n_regionkey, r_regionkey)	24.48	24.48
(1.1) Seq Scan nation	11.7	11.7
(1.2) Seq Scan region (r_name)	12.13	12.13
(2.1) Seq Scan customer	5226	5226
(3.1) Seq Scan orders (o_orderdate)	50326	35966.46
(4.1) Index Scan lineitem	3.27	3.27
(5.1) Seq Scan supplier	332	332

Tabela 5.2: Cenário 3 – Q5

O maior custo neste plano de execução é o *Seq Scan* em *orders*. Os custos de junções não devem ser considerados com o seu valor agregado, mas com a diferença entre o custo agregado e o custo da operação imediatamente anterior (*Cost – Startup Cost*). Por exemplo, o custo do *Hash Join* entre *orders* e *customer* é a diferença entre o custo do próprio *Hash Join* (57014.43) e o custo do *Seq Scan* em *orders* (50326), igual a (6688.43).

A consulta 12 acessa apenas as duas tabelas fato, mas apresenta uma característica diversa: o plano de execução apresenta o custo da varredura em *lineitem* para todo o filtro, que inclui as colunas *l_shipmode*, *l_commitdate*, *l_receiptdate* e *l_shipdate* (D.13).

```
SELECT L_SHIPMODE,
SUM(CASE WHEN O_ORDERPRIORITY = '1-URGENT' OR
O_ORDERPRIORITY = '2-HIGH' THEN 1 ELSE 0 END) AS HIGH_LINE_COUNT,
SUM(CASE WHEN O_ORDERPRIORITY <> '1-URGENT' AND
O_ORDERPRIORITY <> '2-HIGH' THEN 1 ELSE 0 END ) AS LOW_LINE_COUNT
FROM ORDERS, LINEITEM
WHERE O_ORDERKEY = L_ORDERKEY
AND L_SHIPMODE IN ('MAIL','SHIP')
AND L_COMMITDATE < L_RECEIPTDATE
AND L_SHIPDATE < L_COMMITDATE
AND L_RECEIPTDATE >= '1994-01-01'
AND L_RECEIPTDATE < '1995-01-01'
GROUP BY L_SHIPMODE
ORDER BY L_SHIPMODE
```


O plano de execução em *tpch* mostra um *Sequential Scan* em *lineitem* seguido de um *Sort*. O resultado passa por um *Merge Join* com o resultado do *Index Scan* em *orders_pkey*. Nova classificação e agregação terminam o plano. O tempo médio de 3 execuções para esta consulta foi 2882.827 milisegundos.

Em *tpch-i*, o *Sequential Scan* em *lineitem* dá lugar a um *Bitmap Index Scan* em *i_lineitem_shipmode_receiptdate*. O tempo médio de 3 execuções para esta consulta foi 2308.479 milisegundos.

A tabela 5.3 mostra os custos associados aos operadores nos planos de execução em *tpch* e *tpchi*. O maior custo na carga de trabalho está associado à varredura da tabela *lineitem*. Este custo é maior que os custos da query 5 e possui filtro composto nas colunas *l_shipmode*, *l_commitdate*, *l_receiptdate* e *l_shipdate*. Dentre essas colunas, as que possuem maior frequência de acesso são *l_shipmode* e *l_receiptdate*. Dessas duas, a que possui o maior número de valores distintos e que, portanto, possui a seletividade potencial mais alta, é *l_shipmode*.

Operação	Q12	
	tpch	tpch-i
(1) Merge Inner Join (o_orderkey, l_orderkey)	330351.55	208114.27
(1.1) Seq Scan lineitem (l_shipmode, l_commitdate, l_receiptdate, l_shipdate)	257295.07	-
(1.1) Bitmap Index Scan lineitem (l_shipmode, l_receiptdate), Bitmap Heap Scan lineitem (l_shipdate, l_commitdate)	-	135057.79
(1.2) Index Scan orders	66789.43	66789.43

Tabela 5.3: Cenário 3 – Q12

Seleciona-se, portanto, o particionamento horizontal do tipo *list* na coluna *l_shipmode* da tabela *lineitem*. Esta coluna possui sete valores distintos. Através da matriz de afinidades chega-se a cinco partições e seus valores discretos correspondentes.

Neste cenário, a tabela *lineitem* não possui tabelas-filhas. Logo, o particionamento ocorre apenas nesta tabela. Sua chave primária é composta por *(l_orderkey*, *l_linenumber*). Para estas duas colunas, somente *l_orderkey* possui uma chave estrangeira em outra tabela apontando para esta coluna. Esta tabela é *orders*, que não será particionada.

Esta estratégia de particionamento gera benefício para a consulta 12, mas não para a consulta 5. A consulta 12 se beneficia da estratégia porque o operador de busca será aplicado somente às partições que satisfazem o predicado em *l_receiptdate*. Já a consulta 5, que não possui nenhum predicado para a tabela *lineitem*, não se beneficia da estratégia porque o operador de busca será aplicado a todas as partições. Neste cenário, o benefício total acumulado da configuração de índices é maior que o da configuração de particionamento.

5.3.1 Junções sobre partições de tabelas

De fato, a execução da consulta 5 em *tpch-p* inicialmente mostra varreduras completas nas tabelas *region* e *nation* e um *hash join* entre as tabelas. À esta saída é aplicado uma junção com o retorno do *Sequential Scan* em *supplier*. Em seguida, todas as partições da tabela *lineitem* são percorridas e à esta saída é aplicado uma junção com o retorno da varredura em *supplier*. Após, tem-se o *Sequential Scan* em *orders* e um *Nested Loop Join* para unir *orders* e *lineitem*. E uma última junção com a saída da varredura de *customer* usando *(orders.o_custkey = customer.c_custkey).* Na sequencia faz-se o agrupamento e a ordenação. O tempo médio de 3 execuções para esta consulta foi 8264,239 milisegundos, um tempo bem maior que os aferidos em *tpch* e *tpch-i*.

A execução da consulta 12 em *tpch-p* apresenta varreduras sequenciais apenas nas partições de *lineitem* que satisfazem o filtro composto. A esta saída é aplicado um *Merge Join* com o retorno do *Index Scan* em *orders*. O tempo médio de 3 execuções para esta consulta foi 1756.91 milisegundos.

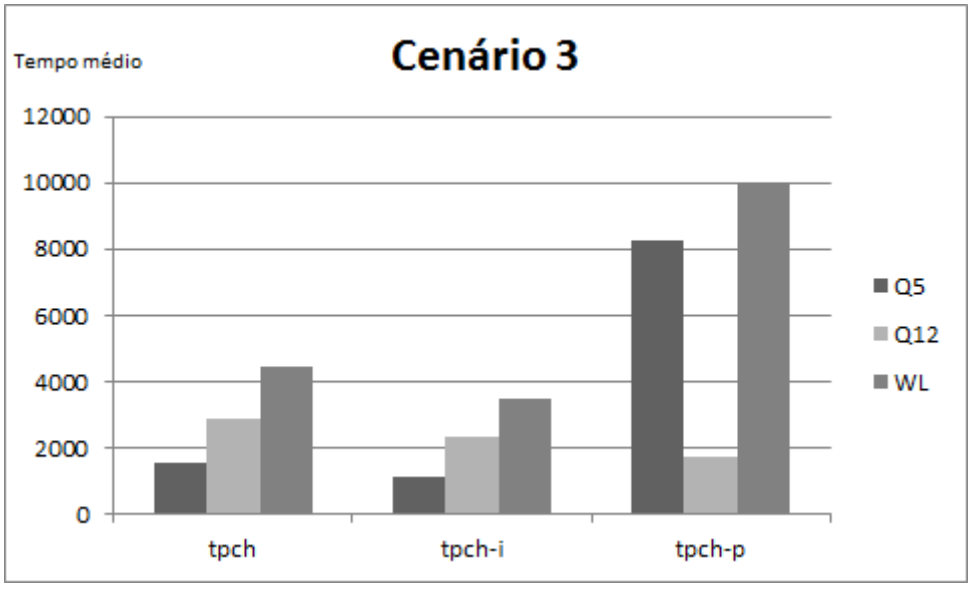

Figura 5.10: Cenário 3 – Resultados

Por que o desempenho de uma consulta (ou de uma carga de trabalho) pode ser pior com a estratégia de particionamento?

A junção entre *orders* e *lineitem* no plano de execução da consulta em *tpch* é realizada por um operador *Nested Loop Join*, indicando que o otimizador sabe que apenas um pequeno percentual de linhas da tabela *lineitem* será selecionado para cada linha da tabela *orders*. Em *tpch-i* o otimizador também seleciona o operador *Nested Loop Join*, após o uso do índice em *orders*.

5.3.2 Malefício da configuração de particionamento

O impacto do particionamento sobre o otimizador de consultas do SGBD é tal que este altera a estratégia de junção entre *orders* e *lineitem*. E esta alteração provoca, então, uma alteração ainda maior no plano de execução, pois ao invés de encaminhar as junções de *region*, *nation*, *customer* e *orders* antes da junção de *orders* com *lineitem*, o otimizador decide realizar as junções de *region*, *nation*, *supplier* e *lineitem* antes de encaminhar a junção com *orders*.

De fato, os papéis das tabelas *orders* e *lineitem* no *Nested Loops* são invertidos no plano gerado pelo otimizador. A tabela *lineitem* passa a ser considerada *outer*, gerando *scans* na tabela *orders*, exatamente o contrário do encontrado em *tpch*. Este impacto na decisão do otimizador gera o alto custo de execução do plano em *tpch-p* e, consequentemente, o tempo maior encontrado nas medições realizadas.

Nota-se, portanto, que o particionamento não só não gerou um benefício para a consulta Q5. O particionamento gerou um malefício não captado pela heurística. Este malefício pode ocorrer toda vez que o otimizador inverter os papéis no *Nested Loops* devido a estatísticas não precisas de tabelas particionadas. Ao particionar tabelas logicamente, o SGBD passa a produzir estatísticas para cada partição da tabela e não para toda a tabela particionada.

5.4 Atualizações

Esta seção estuda as atualizações nos cenários 2 e 3. Seja a seguinte atualização executada no cenário 2, no qual a tabela *orders* foi particionada por *o_orderdate* e a tabela *lineitem* foi particionada por referência a *orders*.

```
UPDATE orders SET o_comment='Lorem ipsum dolor sit amet, '
 WHERE o_orderdate BETWEEN '1994-06-01' AND '1997-10-01'
```

Esta atualização é executada em *tpch* através de um *Sequential Scan* em 29923,080 milisegundos. Em *tpch-p* esta mesma atualização é executada em 32237,294 milisegundos com *Sequential Scans* nas partições 2 a 5 de *orders*.

Seja agora a atualização executada no cenário 3, no qual a tabela *lineitem* foi particionada por *l_shipmode*.

```
UPDATE lineitem SET l_comment='Lorem ipsum dolor sit amet, '
            WHERE l_shipmode IN ('MAIL', 'SHIP')
```

Esta atualização é executada em *tpch* através de um *Sequential Scan* em 150492,298 milisegundos. Em *tpch-p* esta mesma atualização é executada em 63459,862 milisegundos com *Sequential Scan* somente na partição 3 de *lineitem*.

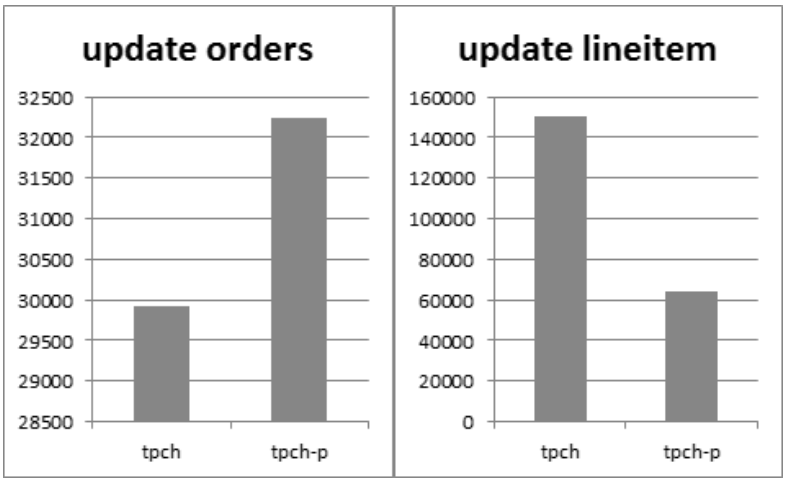

Figura 5.10: Atualizações

Os tempos do update no cenário 2 são semelhantes, porém os tempos medidos para a atualização no cenário 3 indicam que a atualização teve um desempenho melhor com a configuração e particionamento.

Seja agora a atualização abaixo executada nos cenários 2 e 3.

```
UPDATE lineitem AS l
SET l_comment='Lorem ipsum dolor sit amet, '
FROM orders AS o
WHERE l.l_orderkey = o.o_orderkey
AND o_orderdate BETWEEN '1994-06-01' and '1997-10-01'
```

Esta atualização é executada em *tpch* em 39,33 minutos. Em *tpch-p* no cenário 2 esta mesma atualização é executada em 10,13 minutos e em *tpch-p* no cenário 3 esta mesma atualização é executada em 21,92 minutos.

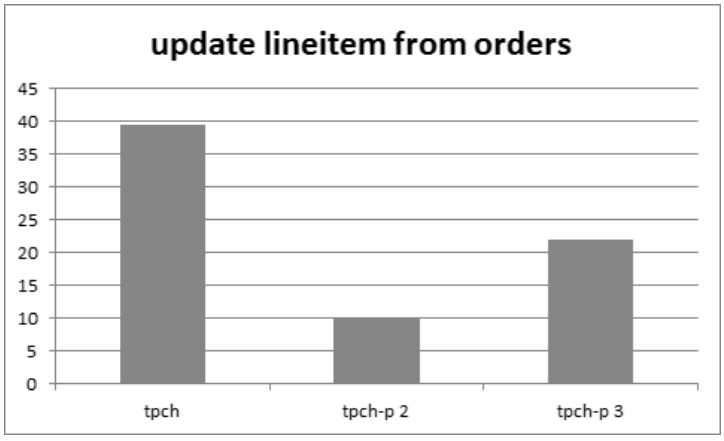

Figura 5.11: Atualizações

Os testes realizados mostram que atualizações também podem se beneficiar do particionamento. Ao executar atualizações com cláusulas *where* que utilizem os mesmos predicados com os quais a estratégia de particionamento foi criada, o benefício também é propagado. Não foram encontrados malefícios para atualizações com predicados outros, que não os chaves de particionamento, na presença ou não de índices *non-clustered*.

5.5 Resumo do capítulo

Neste capítulo foram apresentados 3 cenários de teste com o objetivo de avaliar a heurística HISAP e coletar resultados sobre o uso do particionamento.

No cenário 1 foi constatado o potencial de benefício que o particionamento pode oferecer. Foi mostrado também que o aumento do número de partições acessadas por uma consulta não significa necessariamente que o benefício do particionamento é menor; ao contrário, constatou-se que o benefício do particionamento tornou-se ainda maior que o benefício oferecido pela configuração de índices, dado que a seletividade do predicado utilizado na consulta diminuiu e o índice deixou de ser selecionado pelo otimizador de consultas.

O cenário 2 mostrou como a frequência de uma consulta impacta a estimativa de benefícios da heurística HISAP e permite que o benefício do particionamento seja aplicado a toda a carga de trabalho. Este cenário mostrou também a importância da implementação nativa do particionamento. No Oracle 12c, que possui tal implementação, somente as partições-filhas das partições-mães foram acessadas no plano de execução da consulta Q5, enquanto no PostgreSQL 9.6.1, que não possui uma implementação nativa para o particionamento horizontal, todas as partições da tabela filha, *lineitem*, foram acessadas.

O cenário 3 mostrou que o particionamento também pode gerar um malefício para as consultas de uma carga de trabalho. Este malefício foi constatado quando o otimizador inverteu o papel das tabelas em um *Nested Loops Join*. Isto ocorreu porque o banco de dados não manteve estatísticas precisas sobre as tabelas particionadas e escolheu a maior tabela, *lineitem*, como a tabela *outer* do *Nested Loop*, causando um malefício para a consulta e para toda a carga de trabalho.

No final do capítulo, foi mostrado que atualizações no banco de dados também podem se beneficiar da estratégia de particionamento.

6 Conclusões e trabalhos futuros

Nesta pesquisa foi mostrado que o particionamento pode ser usado como estratégia *principal* de sintonia fina, da mesma forma que índices, índices parciais e visões materializadas. Muito embora ataque em um primeiro momento o problema da seletividade baixa, que faz com que o otimizador do banco de dados não selecione os índices existentes para o plano de execução das consultas, oferece um potencial de desempenho que pode ser superior mesmo quando a seletividade é alta o suficiente para que os índices sejam selecionados.

Diante dos cenários estudados e dos resultados coletados, pode-se apontar as seguintes principais conclusões da pesquisa:

- O particionamento horizontal é uma alternativa com grande potencial de sintonia fina para consultas com predicados de seletividade baixa, cujos planos de execução não utilizam índices secundários. É também uma estratégia principal de sintonia fina, cujo ganho de desempenho pode ser comparado aos de índices, índices parciais e visões materializadas;

- Atualizações também podem se beneficiar do particionamento, principalmente aquelas com cláusulas *where* que utilizem os mesmos predicados com os quais a estratégia de particionamento foi criada;

- Definir uma estratégia de particionamento é um problema de alta complexidade na medida em que, diferentemente dos índices e visões, somente um particionamento pode ser aplicado a cada tabela do banco de dados. Tal estratégia pode oferecer tanto um benefício quanto um malefício para as consultas e atualizações da carga de trabalho;

- A distribuição dos valores para a chave do particionamento da tabela pode priorizar o balanceamento do volume de dados ou o balanceamento da carga de trabalho. Para a sintonia fina, um balanceamento da carga de trabalho pode oferecer um desempenho melhor do que um balanceamento do volume de dados;

- Quando as consultas frequentes da carga de trabalho possuem junções em tabelas particionadas em cadeia, é importante que o SGBD possua uma implementação nativa para o particionamento, pois proporciona ao otimizador percorrer somente as partições que satisfazem os predicados e realizar as junções nas linhas das tabelas particionadas, garantindo melhor desempenho para as consultas;

- Há ainda casos em que o particionamento gera um malefício para a carga de trabalho, piorando o desempenho de consultas. Estes casos são aqueles nos quais as estatísticas do SGBD tornam-se imprecisas para as tabelas particionadas, o que acarreta, por exemplo, inversão de papéis em operadores do tipo *Nested Loops Join*, quando a tabela que deveria ser "*inner*" passa a ser "*outer*".

6.1 Contribuições

Além disso, pode-se pontuar as seguintes principais contribuições desta pesquisa:

- Inclusão do particionamento horizontal lógico como estratégia de sintonia fina de primeira ordem, assim como índices, índices parciais e visões materializadas;

- Comprovação dos benefícios do particionamento para consultas e atualizações;

- Heurística para seleção da estratégia de particionamento;

- Heurística para estimativa de benefício do particionamento.

A heurística HISAP proposta neste trabalho de pesquisa modela o problema da seleção da estratégia de particionamento com base no custo das operações de leitura das tabelas. Com base nos resultados coletados, as cargas de trabalho com consultas que utilizam predicados com seletividade baixa mais frequentemente possuem potencial de benefício maior.

Neste sentido, os bancos de dados OLAP e os bancos de dados OLTP que possuem consultas OLAP (em fechamentos mensais, por exemplo) tendem a se beneficiar mais desta estratégia.

6.2 Trabalhos futuros

Há uma série de oportunidades de melhorias para esta pesquisa.

A principal e mais relevante delas para o uso do particionamento como ação de sintonia é a sua integração com as outras estratégias, como índices e visões materializadas.

As restrições impostas no trabalho são alvo de melhorias. O particionamento vertical, integrado ou não com o particionamento horizontal, possui também excelente potencial para a sintonia fina. Trabalhar com mais de um atributo na chave de particionamento é uma melhora fundamental na pesquisa, pois existem na literatura trabalhos que indicam que bons ganhos em desempenho através do particionamento horizontal foram obtidos com chaves compostas com dois ou três atributos.

Dotar a heurística da capacidade de capturar o malefício do particionamento é fundamental para torná-la ainda mais precisa. Conforme visto no cenário 3, não capturar o malefício pode implicar na sugestão de uma configuração de particionamento que termina por piorar o desempenho geral da carga de trabalho submetida ao banco de dados.

Sendo assim, pode-se listar os principais trabalhos futuros:

- Chaves de particionamento com mais de um atributo;
- Estimativa do malefício do particionamento;
- Avaliação do particionamento vertical;
- Integração do particionamento com outras estratégias de sintonia fina;
- Avaliação do particionamento físico, distribuído ou não;

Para a avaliação do particionamento físico, sabe-se que as partições das tabelas podem ser alocadas em áreas de armazenamento diferentes, no mesmo servidor ou em servidores diferentes. Um modelo de alocação deve ser preparado para atender o problema do particionamento físico. Isto é especialmente relevante na medida em que cada servidor possui restrições diferentes uns dos outros, o que pode implicar em alocar mais de uma partição em um único servidor, por exemplo.

Finalmente, é preciso endereçar, no escopo da sintonia fina integrada, o problema do aumento gradativo da seletividade dos predicados de tabelas particionadas. Seja uma tabela particionada em um determinado momento. Com o passar do tempo, os predicados tendem a ficar mais seletivos a ponto de fazer com que um índice fosse utilizado, caso a tabela não estivesse particionada. Portanto, um "desparticionamento" poderia ser necessário na constatação de que índices, ou outra estratégia de sintonia fina, trariam um benefício maior do que a configuração de particionamento.

<>

Referências Bibliográficas

Agrawal, S.; Narasayya, V. & Yang, B., 2004. **Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design**. In: Proceedings of the ACM SIGMOD, pp. 359–370.

Akal, F.; Böhm, K.; and Schek, H; 2002. **OLAP Query Evaluation in a Database Cluster: a Performance Study on Intra-Query Parallelism**. East European Conference on Advances in Databases and Information Systems (ADBIS), Slovakia, pp. 218-231.

Almeida, A.C.B., 2013. **Framework para apoiar a sintonia fina de banco de dados**. Tese de Doutorado, Departamento de Informática da PUC-Rio.

Almeida, A.C.B.; Brayner, A., Monteiro, J.M.; Lifschitz, S.; Oliveira, R.P., 2015. **DBX: um framework para auto-sintonia fina baseado em planos hipotéticos**. Proceedings of the 30th Brazilian Symposium on Databases pp.149-154.

APB1, 1998. **OLAP Council**. APB-1 OLAP Benchmark Release II. http://www.olapcouncil.org/research/bmarkly.htm [Acessado em 11 de abril de 2017].

AZURE, 2017. **Azure SQL Database Documentation**; http://azure.microsoft.com/pt-br/documentation/services/sql-database [Acessado em 11 de abril de 2017]..

Bellatreche L., Schneider M., Lorinquer H., Mohania M., 2004. **Bringing Together Partitioning, Materialized Views and Indexes to Optimize Performance of Relational Data Warehouses**. In: Kambayashi Y., Mohania M., Wöß W. (eds) Data Warehousing and Knowledge Discovery. DaWaK 2004.

Bhuyar, P. R.; Gawande, A. D.; Deshmukh, A. B., 2012. **Horizontal Fragmentation Technique in Distributed Database**. International Journal of Scientific and Research Publications, 2(5).

Carvalho, A. W., 2011. **Criação Automática de Visões Materializadas em SGBDs Relacionais**. Dissertação de Mestrado, Departamento de Informática da PUC-Rio.

Ceri, S.; Negri, M. & Pelagatti, G., 1982. **Horizontal data partitioning in database design**. In: Proceedings of the ACM SIGMOD.

Chaudhuri, S.; Krishnamurthy, R.; Potamianos, S. & Shim, K., 1995. **Optimizing queries with materialized views**. ICDE, pp.190.

Chaudhuri, S. & Narasayya, V., 1997. **An efficient, cost-driven index selection tool for Microsoft Sql Server**. In: Proceedings of the International conference on Very Large Databases (VLDB), pp.146–155.

Cheng, C.; Lee, W. & Wong, K., 2002. **A Genetic Algorithm-Based Clustering Approach for Database Partitioning**. IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews, 32(3).

Chirkova, R. & Yang, J., 2011. **Materialized Views**. Foundations and Trends in Databases. pp. 296-405.

Costa, R. L. C.; Furtado, P.; 2006. **Data Warehouse in Grids with high QoS**. 8th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2006), pp. 207-217.

Costa, R. L. C.; Lifschitz, S., 2002. **Index self-tuning and agent-based databases**. In: Proceedings of the Latin American Conference on Informatics (CLEI), 2002. 3.3.1, 4.2.2.

Costa, R. L. C.; Lifschitz, S.; Noronha, M. & Salles, M., 2005. **Implementation of an agent architecture for automated index tuning**. International Conference on Data Engineering Workshops (ICDEW).

Curino, C.; Jones, E.; Zhang, Y. & Madden, S., 2010. **Schism: a workload-driven approach to database replication and partitioning**. In: Proceedings of the VLDB, pp. 48–57.

Curino, C; Jones, E., Popa, R. A., Madden, S., Wu, E., Malviya, N.; 2011. **Relational Cloud: The case for a database service**. New England Database Summit.

Curino, C.; Pavlo, A.; Zdonik, S, 2012. **Skew-aware Automatic Database Partitioning in Shared-Nothing Parallel OLTP Systems**. In: Proceedings of the ACM SIGMOD, pp. 61–72.

DTA, 2017. **Microsoft Database Engine Tuning Advisor**. Available at: https://msdn.microsoft.com/en-us/library/ms166575.aspx [Acessado em 11 de abril de 2017].

DTECH, 2016. **Top Ten Largest Databases in the World**. http://csnipuntech.blogspot.com.br/2014/05/top-10-largest-databases-in-world.html. [Acessado em 5 de abril de 2016].

Furtado, C.; Lima, A. A. B.; Pacitti, E.; Valduriez, P.; Mattoso, M; 2005. **Physical and Virtual Partitioning in OLAP Database Clusters**. SBAC-PAD In: Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing, pp. 143-150.

Ioannidis, Y., Poosala, V., 1995: **Histogram-Based Solutions to Diverse Database Estimation Problems**. Data Engineering Bulletin 18(3).

Khedkar, S. & Gawande, A., 2014. **Data Partitioning Technique to Improve Cloud Data Storage Security**. (IJCSIT) International Journal of Computer Science and Information Technologies, 5(3), pp. 3347-3350.

Kossmann, D.; 2000. **The state of the Art in Distributed Query Processing**. ACM Computing Surveys, 32(4).

Lohman, G.; Valentin G.; Zilio, D.; Zuliani, M. & Skelley, A., 2000. **DB2 advisor**: an optimizer smart enough to recommend its own indexes. In: proceedings of the IEEE International Conference on Data Engineering (ICDE), pp. 101–110.

Lifschitz, S.; Milanes, A.; Salles, M. V.; 2004. **Estado da arte em auto-sintonia de SGBD relacionais**. Relatório técnico, Departamento de Informática, PUC-RIO.

Lima, A. A. B., Furtado, C.; Valduriez, P.; Mattoso, M.; 2009. **Parallel OLAP query processing in database clusters with data replication**. Distributed and Parallel Database Journal 25(1-2), pp. 97-123.

Liroz-Gistau, M.; Akbarinia, R.; Pacitti, E.; Porto, F.; Valduriez, P.; 2012. **Dynamic Workload-based Partitioning for Large Scale Databases**. DEXA'2012: 23rd International Conference on Database and Expert Systems Applications, pp.183-190.

Luhring M.; Sattler K. U.; Schmidt, K.; Schallehn, E., 2007. **Autonomous management of soft indexes**. In: Proceedings of the IEEE International Conference on Data Engineering, pp. 450–458.

Monteiro, J. M., 2008. **Uma abordagem não intrusiva para a manutenção automática do projeto físico de bancos de dados**. Tese de doutorado, Departamento de Informática da PUC-Rio.

Monteiro, J. M.; Lifschitz, S.; Brayner, A., 2012. **AIM-HypoPlans**: Automatic Index Maintenance using Hypothetical Query Execution Plans.

Morelli, E.M.T., 2006. **Recriação Automática de Índices em um SGBD Relacional**. Dissertação de Mestrado, Departamento de Informática da PUC-Rio.

Nehme, R. & Bruno, N., 2011. **Automated Partitioning Design in Parallel Database Systems**. In: Proc. of the ACM SIGMOD, pp. 1137–1148.

Oliveira, R.P., 2014. **Sintonia fina baseada em ontologia: o caso de visões materializadas**. Dissertação de Mestrado, Departamento de Informática da PUC-Rio.

ORA, 2016. **Optimizing Access Paths with SQL Access Advisor**. Available at: https://docs.oracle.com/database/121/TGSQL/tgsql_ sqlaccess.htm [Acessado em 7 de Setembro de 2016].

Papadomanolakis, S. & Ailamaki, A. 2004. **Autopart: Automating schema design for large scientific databases using data partitioning**. In: Proceedings of SSDBM.

PGSQL, 2016. **PostgreSQL Documentation**; http://www.postgresql.org/ docs [Acessado em 8 de agosto de 2016].

PGSQL-PUC-RIO, 2016. **Grupo de Auto-sintonia de Banco de Dados da PUC-Rio**; http://www.inf.puc-rio.br/~postgresql [Acessado em 8 de agosto de 2016].

Ramakhrishnan, R.; Gehrke, J., 2002. **Database Management Systems**. 3rd. edition. Mcgraw-Hill. 1098p.

Salles, M. V., 2004. **Criação autônoma de índices em bancos de dados**. Dissertação de Mestrado, Departamento de Informática da PUC-Rio.

Shasha, D. & Bonnet, P., 2002. **Database Tuning: Principles, Experiments, and Troubleshooting Techniques**, Elsevier Science.

SQL, 2008. **SQL Server 2008 R2**. https://msdn.microsoft.com/pt-br/library/ms143287(v=sql.105).aspx. [Acessado em 7 de setembro de 2016].

SQL, 2016. **SQL Server Database Engine Tuning Advisor**, https://technet.microsoft.com/en-us/library/ms173494(v=sql.105).aspx [Acessado em 7 de setembro de 2016].

TPC, 2016. **TPC-H Benchmark**, http://www.tpc.org/tpch [Acessado em 7 de Setembro de 2016].

N. Duan, B. Gao, C. J. Guo, J. M. Zhang; 2016. **Method and system for database partition**. US 9317577 B2. Disponível em http://www.google.com/patents/US9317577 [Acessado em 6 de agosto de 2016].

Valduriez, P.; Özsu, M.T., 2011. **Principles Of Distributed Database Systems**. 3rd. edition. Springer 2011. 845p.

Wiederhold, G.; 2001. **Database Design**. McGraw-Hill, 2nd. Edition.

Wu, E. & Madden, S., 2011. **Partitioning Techniques for Fine-Grained Indexing**. International Conference on Data Engineering, pp. 1127-1138.

Wang, X.; Chen, J.; & Du, X.; 2013. **ASAWA: An Automatic Partition Key Selection Strategy**. APWeb 2013, LNCS 7808, pp. 609–620.

Zilio, D., 1994. **Partitioning Key Selection for a Shared-Nothing Parallel Database System**. IBM Research Report RC 19820 (87739), IBM Research Division, T.J. Watson Research Center, NY.

Apêndice A – Esquema do TPC-H

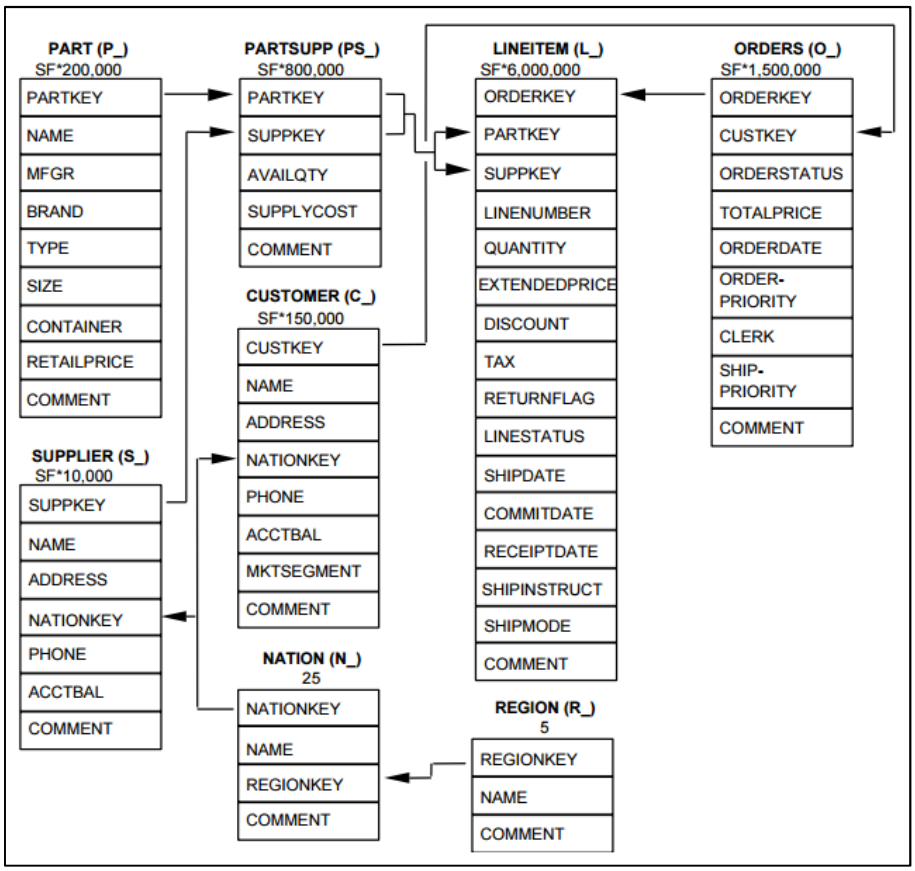

Figura A.1: Esquema do banco de dados TPCH (TPC, 2016)

Apêndice B – Heurística HISAI

Este apêndice apresenta um breve resumo da Heurística HISAI – Heurística Integrada para Seleção e Acompanhamento de Índices (Monteiro, 2008). Os pseudo-códigos dos algoritmos que a implementam estão em (Monteiro, 2012).

Inicialmente, analisa-se o algoritmo de análise de índices candidatos (IHSTIS_CI). Entre as linhas 6 e 17 o algoritmo percorre o plano de execução das *queries* em busca das operações de varredura sequencial de tabelas (*Sequential Scan*) e adiciona à lista de índices candidatos ($L_{candidates}$) as chaves para índices hipotéticos geradas conforme a a especificação da heurística SAEFIS (Lohman, 2000).

Algorithm 2 IHSTIS CI Heuristic: Candidate Index Analysis

Require: a task T (<SQL expression Q, execution plan P, cost estimate CE_P>) as input
1: HP = P {the hypothetical plan HP is initialized with the real plan P}
2: if T ∉ LM then
3: $T_{IND} = \varphi$ {The indexes set for task T}
4: end if
5: Search the query execution plan HP looking for Seq Scan operations
6: for each seq scan operation s ∈ HP do
7: Initialize the candidate index list ($L_{candidates} = \varphi$)
8: if s has filters used in WHERE/JOIN clauses {Case 1} then
9: for each attribute a_i used in where/join do
10: $L_{candidates} = L_{candidates} \cup \{h\ a_i\}$ {h a_i is an index on a_i}
11: end for
12: end if
13: {Combine the columns not involved with a filter (Case 2)}
14: for each attribute b_i used in SELECT/GROUP/ORDER clauses do
15: $L_{candidates} = L_{candidates} \cup \{h\ b_i\}$ {h b_i is an index on b_i}
16: end for
17: $L_{candidates} = L_{candidates} \cup \{h\ ab_i\}$ {h ab_i is an index on columns used in Case 1 + columns used in Case 2}
18: for all candidate index k ∈ $L_{candidates}$ do
19: Estimates the cost of Index Scan ($CE_{IS_{k\ primary}}$) using a primary (clustered) index k primary
20: Estimates the cost of Index Scan ($CE_{IS_{k_a\ secondary}}$) using a secondary (non-clustered) index k secondary
21: V = {k primary} ∪ {k secondary} {a set with two versions of k, one as primary and the other as secondary index}
22: for each index k ∈ V do
23: if k ∉ LM then
24: $L_{Hypothetical} = L_{Hypothetical} \cup k$
25: $AB_k = 0$
26: $NQ_k = 0$ {Number of queries where k may be used}
27: end if
28: if k ∉ T_{IND} {T_{IND} is the set of indexes used to process the task T} then
29: $T_{IND} = T_{IND} \cup \{k\}$
30: end if
31: $NQ_k = NQ_k + 1$
32: if k ∈ $L_{Hypothetical}$ {k is hypothetical index} then
33: if k == primary {k is primary (clustered) index} then
34: if $CE_{IS_{k\ primary}} < CE_{FS}$ then
35: $AB_{k\ primary} = AB_{k\ primary} + (CE_{SS} - CE_{IS_{k\ primary}})$
36: end if
37: else if k == secondary {k is secondary (non-clustered) index} then
38: if $CE_{IS_{k\ secondary}} < CE_{FS}$ then
39: $AB_{ksecondary} = AB_{ksecondary} + (CE_{SS} - CE_{IS_{k\ secondary}})$
40: end if
41: end if
42: end if
43: end for
44: end for
45: HP = HP changing the seq scan operation s by a index scan operation
46: end for

Figura B.1: Heurística IHSTIS_CI (Monteiro, 2012)

Na sequencia, entre as linhas 18 e 42, o benefício para cada índice é acumulado (AB_k), sendo ele a diferença entre o custo estimado do plano de execução utilizando a varredura sequencial da tabela e o custo estimado do plano utilizando o próprio índice hipotético.

A heurística também trata os índices reais (algoritmo IHSTIS_RI), isto é, aqueles que de fato existem no banco de dados. Quando um índice real não é utilizado no plano de execução, seu benefício acumulado é decrementado (linhas 2 e 3), e quando é utilizado, seu benefício acumulado é incrementado (linha 9), conforme mostrado na figura B.2.

Algorithm 3 IHSTIS RI Heuristic: Real Indexes Analysis

1: for each index k, such that $k \in T_{IND}$ and $k \in L_{Real}$ do
2: if $k \notin P$ then
3: $AB_k = AB_k - |(AB_k / NQ_k)|$ {the acummulated benefit AB_k for k is decremented}
4: end if
5: end for
6: Search the query execution plan P looking for a Index Scan operation
7: if a Index Scan operation is found then
8: Let i be the index used in the Index Scan operation
9: $AB_i = AB_i + (CE_{SS} - CE_{IS})$
10: if $i \notin T_{IND}$ then
11: $T_{IND} = T_{IND} \cup \{i\}$
12: end if
13: end if

Figura B.2: Heurística IHSTIS_RI (Monteiro, 2012)

Quanto às operações de atualização de dados, o algoritmo IHSTIS_U calcula o custo de atualização do índice e decrementa do benefício acumulado este custo. Ademais, mantém uma lista de índices fragmentados com respeito a um limite definido pelo DBA.

Algorithm 4 IHSTIS U Heuristic: Update Analysis

1: for all index i that need be updated do
2: $CE_{U_i} = HT_i \times N_{O_{TT}} \times F$ {CE_{U_i} is the estimated cost for updating index structure i, F is the cost of updating one node of i, N_{QU} the number of tuples to be inserted/deleted and HT_i represents i's height}
3: $AB_i = AB_i - CE_{U_i}$
4: FL_i = Fragmentation level of index i
5: TU_i = Total times the index i was used
6: if $(i \in L_{Real})$ and $(FL_i * \frac{v}{TU_i} > \text{fraglimit})$ then
7: $L_{ind\ frag} = L_{ind\ frag} \cup \{i\}$ {set of fragmented indexes}
8: end if
9: end for

Figura B.3: Heurística IHSTIS_U (Monteiro, 2012)

Os três algoritmos mostrados são chamados em sequencia na fase de observação da ferramenta de sintonia. Na medida em que a carga de trabalho vai sendo analisada, os benefícios de cada estrutura hipotética vão sendo calculados. Conforme (Monteiro, 2012), a heurística trabalha com duas configurações, a saber, C e C_I. Em C está a configuração atual do banco de dados e em C_I está a configuração do banco com os índices reais e hipotéticos.

Os índices em I' são colocados em ordem decrescente de benefício e o algoritmo varre I' a partir do índice que apresenta o maior benefício relativo. Para um determinado índice I'[k], $P_{I'[k]}$ é a estimativa de espaço em disco necessário para criar o índice. Para uma determinada restrição definida pelo DBA, *storage_constraint*, o algoritmo chega a uma nova configuração C_I, que minimiza os tempos de resposta das consultas enquanto observa a restrição de espaço em disco.

Algorithm 5 Greedy Algorithm for Index Structure Configuration Selection

```
1: I°[1···n] ← sort(I) by the relative benefit {I represents the set of all index structures (real and hypothetical)}
2: C ← φ
3: available space ← storage constraint
4: total benefit ← 0
5: for all k ← 1···n do
6:     if ((type(I°[k]) = P) and there is no other primary index on the same table) or ((type(I°[k]) = S) then
7:         if (available space - P_I°[k]) > 0 then
8:             C ← C ∪ {I°[k]}
9:             available space ← available space - P_I°[k]
10:            total benefit ← total benefit + AB_I°[k]
11:        end if
12:    end if
13: end for
14: return C
```

Figura B.4: Algoritmo guloso para seleção da configuração (Monteiro, 2012)

A mudança de uma configuração C para uma configuração C_I ocorre quando o benefício total da configuração C_I é maior que o benefício da configuração atual C vezes um fator *k* definido pelo DBA. Neste caso, cada índice hipotético em C_I é criado fisicamente no banco de dados (figura B.5, linha 4). Da mesma forma, é verificado se existe algum índice em C que não existe em C_I. Neste caso, o índice é apagado do banco de dados, seu estado é configurado como hipotético e seu benefício acumulado é zerado (linhas 10 a 12). Finalmente, para todos os índices com índice de fragmentação abaixo do limite definido (L_{ind_frag}), um *reindex* é executado (linha 19), conforme figura B.5.

Algorithm 6 Algorithm for Indexes Structures Configuration Update

```
1: if B_C̄ ≥ B_C × k then
2:     for all index i ∈ C do
3:         if i ∈ L_Hypothetical then
4:             Create physically the index i
5:             state(i) ← real
6:         end if
7:     end for
8:     for all index i ∈ L_Real do
9:         if i ∉ C then
10:            Drop index i
11:            state(i) ←hypothetical
12:            if AB_i ≥ 0 then
13:                AB_i = 0
14:            end if
15:        end if
16:    end for
17: end if
18: for all index i ∈ L_ind frag do
19:     Reindex i
20: end for
```

Figura B.5: Algoritmo de atualização da configuração (Monteiro, 2012)

Apêndice C – Implementação do particionamento

C.1 Cenário 1

O particionamento horizontal da tabela *lineitem* é definido com base na coluna *l_shipdate*. Cada uma das cinco partições da tabela contém a definição das faixas de valores para esta coluna.

```
create table lineitem_p1 ( constraint lineitem_p1_pkey primary
key (l_linenumber, l_orderkey),
constraint lineitem_l_orderkey_fkey foreign key (l_orderkey)
references orders (o_orderkey),
constraint lineitem_l_partkey_fkey foreign key (l_partkey,
l_suppkey) references partsupp (ps_partkey, ps_suppkey),
check ( l_shipdate<='1993-05-26' ) ) inherits(lineitem);
```

```
create table lineitem_p2 ( constraint lineitem_p2_pkey primary
key (l_linenumber, l_orderkey),
constraint lineitem_l_orderkey_fkey foreign key (l_orderkey)
references orders (o_orderkey),
constraint lineitem_l_partkey_fkey foreign key (l_partkey,
l_suppkey) references partsupp (ps_partkey, ps_suppkey),
check ( l_shipdate>'1993-05-26' and l_shipdate<='1994-10-07' ) )
inherits(lineitem);
```

```
create table lineitem_p3 ( constraint lineitem_p3_pkey primary
key (l_linenumber, l_orderkey),
constraint lineitem_l_orderkey_fkey foreign key (l_orderkey)
references orders (o_orderkey),
constraint lineitem_l_partkey_fkey foreign key (l_partkey,
l_suppkey) references partsupp (ps_partkey, ps_suppkey),
check ( l_shipdate>'1994-10-07' and l_shipdate<='1996-02-10' ) )
inherits(lineitem);
```

```
create table lineitem_p4 ( constraint lineitem_p4_pkey primary
key (l_linenumber, l_orderkey),
constraint lineitem_l_orderkey_fkey foreign key (l_orderkey)
references orders (o_orderkey),
constraint lineitem_l_partkey_fkey foreign key (l_partkey,
l_suppkey) references partsupp (ps_partkey, ps_suppkey),
```

```
check ( l_shipdate>'1996-02-10' and l_shipdate<='1997-05-19' ) )
inherits(lineitem);
```

```
create table lineitem_p5 ( constraint lineitem_p5_pkey primary
key (l_linenumber, l_orderkey),
constraint lineitem_l_orderkey_fkey foreign key (l_orderkey)
references orders (o_orderkey),
constraint lineitem_l_partkey_fkey foreign key (l_partkey,
l_suppkey) references partsupp (ps_partkey, ps_suppkey),
check ( l_shipdate>'1997-05-19' ) ) inherits(lineitem);
```

A tabela contém um *trigger* para redirecionar *inserts*, *updates* e *deletes* para as partições da tabela correspondentes às faixas de valores de *l_shipdate*. Abaixo, a definição do *trigger* e o código da função para inserção de dados.

```
create trigger lineitem_insert_trg before insert on lineitem for
each row execute procedure lineitem_insert();
```

```
create function lineitem_insert()
returns trigger as $$
begin
if (new.l_shipdate<='1993-05-26') then insert into lineitem_p1
values (new.*);
elseif (new.l_shipdate>'1993-05-26' and new.l_shipdate<='1994-10-
07') then insert into lineitem_p2 values (new.*);
elseif (new.l_shipdate>'1994-10-07' and new.l_shipdate<='1996-02-
10') then insert into lineitem_p3 values (new.*);
elseif (new.l_shipdate>'1996-02-10' and new.l_shipdate<='1997-05-
19') then insert into lineitem_p4 values (new.*);
elseif (new.l_shipdate>'1997-05-19') then insert into lineitem_p5
values (new.*);
end if;
return null;
end;
$$
language plpgsql;
```

C.2 Cenário 2

O particionamento horizontal da tabela *orders* é definido com base na coluna *o_orderdate*. Cada uma das cinco partições da tabela contém a definição das faixas de valores para esta coluna.

```
create table orders_p1 ( constraint orders_p1_pkey primary key
(o_orderkey), constraint orders_p1_o_custkey_fkey foreign key
(o_custkey) references customer (c_custkey), check
(o_orderdate<='1993-04-03') ) inherits(orders);

create table orders_p2 ( constraint orders_p2_pkey primary key
(o_orderkey), constraint orders_p2_o_custkey_fkey foreign key
(o_custkey) references customer (c_custkey), check
(o_orderdate>'1993-04-03' and o_orderdate<='1994-07-07') )
inherits(orders);

create table orders_p3 ( constraint orders_p3_pkey primary key
(o_orderkey), constraint orders_p3_o_custkey_fkey foreign key
(o_custkey) references customer (c_custkey), check
(o_orderdate>'1994-07-07' and o_orderdate<='1995-11-23') )
inherits(orders);

create table orders_p4 ( constraint orders_p4_pkey primary key
(o_orderkey), constraint orders_p4_o_custkey_fkey foreign key
(o_custkey) references customer (c_custkey), check
(o_orderdate>'1995-11-23' and o_orderdate<='1997-03-07') )
inherits(orders);

create table orders_p5 ( constraint orders_p5_pkey primary key
(o_orderkey), constraint orders_p5_o_custkey_fkey foreign key
(o_custkey) references customer (c_custkey), check
(o_orderdate>'1997-03-07') ) inherits(orders);
```

A tabela contém um *trigger* para redirecionar *inserts*, *updates* e *deletes* para as partições da tabela correspondentes às faixas de valores de *o_orderdate*. Abaixo, a definição do *trigger* e o código da função para inserção de dados.

```
create trigger orders_insert_trg before insert on orders for each
row execute procedure orders_insert();
```

```
create function orders_insert()
returns trigger as $$
begin
      if (new.o_orderdate<='1993-04-03') then insert into
orders_p1 values (new.*);
      elseif (new.o_orderdate>'1993-04-03' and
new.o_orderdate<='1994-07-07') then insert into orders_p2 values
(new.*);
      elseif (new.o_orderdate>'1994-07-07' and
new.o_orderdate<='1995-11-23') then insert into orders_p3 values
(new.*);
      elseif (new.o_orderdate>'1995-11-23' and
new.o_orderdate<='1997-03-07') then insert into orders_p4 values
(new.*);
      elseif (new.o_orderdate>'1997-03-07') then insert into
orders_p5 values (new.*);
end if;
return null;
end;
$$
language plpgsql;
```

O particionamento horizontal da tabela *lineitem* é definido por referência à tabela *orders*. É criada uma função de verificação *is_orders_p** para cada partição da tabela e esta função é chamada tanto na *constraint check* quanto na função chamada pelos *triggers*. Abaixo, o código das funções de verificação de cada partição da tabela.

```
create function is_orders_p1(int) returns boolean as $$
select exists ( select 1 from "orders" where o_orderkey=$1 and
o_orderdate<='1993-04-03' );
$$
language sql;
```

```
create function is_orders_p2(int) returns boolean as $$
select exists ( select 1 from "orders" where o_orderkey=$1 and
o_orderdate>'1993-04-03' and o_orderdate<='1994-07-07' );
$$
language sql;
```

```
create function is_orders_p3(int) returns boolean as $$
select exists ( select 1 from "orders" where o_orderkey=$1 and
o_orderdate>'1994-07-07' and o_orderdate<='1995-11-23' );
$$
language sql;
```

```
create function is_orders_p4(int) returns boolean as $$
select exists ( select 1 from "orders" where o_orderkey=$1 and
o_orderdate>'1995-11-23' and o_orderdate<='1997-03-07' );
$$
language sql;
```

```
create function is_orders_p5(int) returns boolean as $$
select exists ( select 1 from "orders" where o_orderkey=$1 and
o_orderdate>'1997-03-07' );
$$
language sql;
```

Para garantir bom desempenho na execução das funções acima, é importante a criação de índices nas partições da tabela *orders*, conforme abaixo:

```
create index i_orders_p1 on orders_p1 (o_orderkey, o_orderdate);
create index i_orders_p2 on orders_p2 (o_orderkey, o_orderdate);
create index i_orders_p3 on orders_p3 (o_orderkey, o_orderdate);
create index i_orders_p4 on orders_p4 (o_orderkey, o_orderdate);
create index i_orders_p5 on orders_p5 (o_orderkey, o_orderdate);
```

A criação das tabelas particionadas utiliza chamada às funções de verificação:

```
create table lineitem_p1 ( constraint lineitem_p1_pkey primary
key (l_linenumber, l_orderkey), constraint lineitem_p1_l_
orderkey_fkey foreign key (l_orderkey) references orders_p1
(o_orderkey), constraint lineitem_p1_l_partkey_fkey foreign key
(l_partkey, l_suppkey) references partsupp (ps_partkey,
ps_suppkey), check ( is_orders_p1 ( cast (l_orderkey as int) ) )
) inherits(lineitem);

create table lineitem_p2 ( constraint lineitem_p2_pkey primary
key (l_linenumber, l_orderkey), constraint lineitem_p2_l_
```

```
orderkey_fkey foreign key (l_orderkey) references orders_p2
(o_orderkey), constraint lineitem_p2_l_partkey_fkey foreign key
(l_partkey, l_suppkey) references partsupp (ps_partkey,
ps_suppkey), check ( is_orders_p2 ( cast (l_orderkey as int) ) )
) inherits(lineitem);

create table lineitem_p3 ( constraint lineitem_p3_pkey primary
key (l_linenumber, l_orderkey), constraint lineitem_p3_l_
orderkey_fkey foreign key (l_orderkey) references orders_p3
(o_orderkey), constraint lineitem_p3_l_partkey_fkey foreign key
(l_partkey, l_suppkey) references partsupp (ps_partkey,
ps_suppkey), check ( is_orders_p3 ( cast (l_orderkey as int) ) )
) inherits(lineitem);

create table lineitem_p4 (constraint lineitem_p4_pkey primary key
(l_linenumber, l_orderkey), constraint lineitem_p4_l_orderkey_
fkey foreign key (l_orderkey) references orders_p4 (o_orderkey),
constraint lineitem_p4_l_partkey_fkey foreign key (l_partkey,
l_suppkey) references partsupp (ps_partkey, ps_suppkey), check (
is_orders_p4 ( cast (l_orderkey as int) ) ) ) inherits(lineitem);

create table lineitem_p5 ( constraint lineitem_p5_pkey primary
key (l_linenumber, l_orderkey), constraint lineitem_p5_l_
orderkey_fkey foreign key (l_orderkey) references orders_p5
(o_orderkey), constraint lineitem_p5_l_partkey_fkey foreign key
(l_partkey, l_suppkey) references partsupp (ps_partkey,
ps_suppkey), check ( is_orders_p5 ( cast (l_orderkey as int) ) )
) inherits(lineitem);
```

Abaixo, a criação do *trigger* e da função por ele chamada para a inserção dos dados:

```
create trigger lineitem_insert_trg before insert on lineitem for
each row execute procedure lineitem_insert();
```

```
create or replace function lineitem_insert()
returns trigger as $$
begin
if (is_orders_p1(cast(new.l_orderkey as int))) then insert into
lineitem_p1 values (new.*);
```

```
elseif (is_orders_p2(cast(new.l_orderkey as int))) then insert
into lineitem_p2 values (new.*);
elseif (is_orders_p3(cast(new.l_orderkey as int))) then insert
into lineitem_p3 values (new.*);
elseif (is_orders_p4(cast(new.l_orderkey as int))) then insert
into lineitem_p4 values (new.*);
elseif (is_orders_p5(cast(new.l_orderkey as int))) then insert
into lineitem_p5 values (new.*);
end if;
return null;
end;
$$
language plpgsql;
```

C.3 Cenário 3

O particionamento horizontal da tabela *lineitem* é definido com base na coluna *l_receiptdate*. Cada uma das cinco partições da tabela contém a definição das faixas de valores para esta coluna.

```
create table lineitem_p1 ( constraint lineitem_p1_pkey primary
key (l_linenumber, l_orderkey),
constraint lineitem_l_orderkey_fkey foreign key (l_orderkey)
references orders_p1 (o_orderkey),
constraint lineitem_l_partkey_fkey foreign key (l_partkey,
l_suppkey) references partsupp (ps_partkey, ps_suppkey),
check ( l_shipmode='AIR' ) ) inherits(lineitem);
```

```
create table lineitem_p2 ( constraint lineitem_p2_pkey primary
key (l_linenumber, l_orderkey),
constraint lineitem_l_orderkey_fkey foreign key (l_orderkey)
references orders_p2 (o_orderkey),
constraint lineitem_l_partkey_fkey foreign key (l_partkey,
l_suppkey) references partsupp (ps_partkey, ps_suppkey),
check ( l_shipmode='FOB' ) ) inherits(lineitem);
```

```
create table lineitem_p3 ( constraint lineitem_p3_pkey primary
key (l_linenumber, l_orderkey),
constraint lineitem_l_orderkey_fkey foreign key (l_orderkey)
references orders_p3 (o_orderkey),
constraint lineitem_l_partkey_fkey foreign key (l_partkey,
l_suppkey) references partsupp (ps_partkey, ps_suppkey),
check ( l_shipmode='MAIL' or l_shipmode='SHIP' ) )
inherits(lineitem);
```

```
create table lineitem_p4 ( constraint lineitem_p4_pkey primary
key (l_linenumber, l_orderkey),
constraint lineitem_l_orderkey_fkey foreign key (l_orderkey)
references orders_p4 (o_orderkey),
constraint lineitem_l_partkey_fkey foreign key (l_partkey,
l_suppkey) references partsupp (ps_partkey, ps_suppkey),
check ( l_shipmode='RAIL' or l_shipmode='REG AIR' ) )
inherits(lineitem);
```

```
create table lineitem_p5 ( constraint lineitem_p5_pkey primary
key (l_linenumber, l_orderkey),
constraint lineitem_l_orderkey_fkey foreign key (l_orderkey)
references orders_p5 (o_orderkey),
constraint lineitem_l_partkey_fkey foreign key (l_partkey,
l_suppkey) references partsupp (ps_partkey, ps_suppkey),
check ( l_shipmode='TRUCK' ) ) inherits(lineitem);
```

A tabela contém um *trigger* para redirecionar *inserts*, *updates* e *deletes* para as partições da tabela correspondentes às faixas de valores de *o_orderdate*. Abaixo, a definição do *trigger* e o código da função para inserção de dados.

```
create trigger lineitem_insert_trg before insert on lineitem for
each row execute procedure lineitem_insert();
```

```
create function lineitem_insert()
returns trigger as $$
begin
if (new.l_shipmode='AIR') then insert into lineitem_p1 values
(new.*);
elseif (new.l_shipmode='FOB') then insert into lineitem_p2 values
(new.*);
elseif (new.l_shipmode='MAIL' or new.l_shipmode='SHIP') then
insert into lineitem_p3 values (new.*);
elseif (new.l_shipmode='RAIL' or new.l_shipmode='REG AIR') then
insert into lineitem_p4 values (new.*);
elseif (new.l_shipmode='TRUCK') then insert into lineitem_p5
values (new.*);
end if;
return null;
end;
$$
language plpgsql;
```

Apêndice D – Planos de execução

D.1 Cenário 1 – Q1 – TPCH


```
[
  {
    "Execution Time": 3567.668,
    "Planning Time": 0.15,
    "Plan": {
      "Node Type": "Sort",
      "Sort Key": ["l_returnflag", "l_linestatus"],
      "Plans": [
        {
          "Node Type": "Aggregate",
          "Group Key": ["l_returnflag", "l_linestatus"],
          "Plans": [
            {
              "Node Type": "Seq Scan",
              "Relation Name": "lineitem",
              "Filter": "(l_shipdate <= '1993-01-01')",
              "Parallel Aware": false,
              "Total Cost": 197220.14
            }
          ],
          "Total Cost": 227539.78
        }
      ],
      "Total Cost": 227539.88
    },
  }
]
```

D.2 Cenário 1 – Q1 – TPCH-I


```
[
  {
    "Execution Time": 3362.235,
    "Planning Time": 2.527,
    "Plan": {
      "Node Type": "Sort",
      "Sort Key": ["l_returnflag", "l_linestatus"],
      "Plans": [
        {
          "Node Type": "Aggregate",
          "Group Key": ["l_returnflag", "l_linestatus"],
          "Plans": [
            {
              "Node Type": "Bitmap Heap Scan",
              "Plans": [
                {
                  "Node Type": "Bitmap Index Scan",
                  "Index Name": "i_lineitem_shipdate_returnflag_linestatus",
                  "Index Cond": "(l_shipdate <= '1993-01-01')",
                  "Total Cost": 14042.38
                }
              ],
              "Relation Name": "lineitem",
              "Total Cost": 146003.7
            }
          ],
          "Total Cost": 176414.26
        }
      ],
      "Total Cost": 176414.36
    },
    "Triggers": []
  }
]
```

D.3 Cenário 1 – Q1 – TPCH-P


```
[
  {
    "Execution Time": 3221.806,
    "Planning Time": 29.579,
    "Plan": {
      "Node Type": "Aggregate",
      "Group Key": ["lineitem.l_returnflag", "lineitem.l_linestatus"],
      "Plans": [
        {
          "Node Type": "Sort",
          "Sort Key": ["lineitem.l_returnflag", “lineitem.l_linestatus"],
          "Plans": [
            {
              "Node Type": "Append",
              "Plans": [
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "lineitem",
                  "Filter": "(l_shipdate <= '1993-01-01')",
                  "Total Cost": 0
                },
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "lineitem_p1",
                  "Filter": "(l_shipdate <= '1993-01-01')",
                  "Total Cost": 36787.31
                }
              ],
              "Total Cost": 36787.31
            }
          ],
          "Total Cost": 130596.35
        }
      ],
      "Total Cost": 161940.67
    },
    "Triggers": []
  }
]
```

D.4 Cenário 2 – Q3 – TPCH


```
[
  {
    "Execution Time": 1526.891,
    "Planning Time": 0.778,
    "Plan": {
      "Node Type": "Limit",
      "Plans": [
        {
          "Node Type": "Sort",
          "Sort Key": ["(sum((lineitem.l_extendedprice * ('1' -
lineitem.l_discount)))) DESC", "orders.o_orderdate"],
          "Plans": [
            {
              "Node Type": "Aggregate",
              "Group Key": ["lineitem.l_orderkey", "orders.o_orderdate",
                "orders.o_shippriority"],
              "Plans": [
                {
                  "Node Type": "Sort",
                  "Sort Key": ["lineitem.l_orderkey", "orders.o_orderdate",
                    "orders.o_shippriority"],
                  "Plans": [
                    {
                      "Node Type": "Nested Loop",
                      "Plans": [
                        {
                          "Node Type": "Nested Loop",
                          "Plans": [
                            {
                              "Node Type": "Seq Scan",
                              "Relation Name": "lineitem",
                              "Filter": "(l_shipdate > '1999-03-15')",
                              "Total Cost": 197283.04
                            },
                            {
                              "Node Type": "Index Scan",
                              "Relation Name": "orders",
                              "Index Name": "orders_pkey",
                              "Index Cond": "(o_orderkey = lineitem.l_orderkey)",
                              "Filter": "(o_orderdate < '1994-03-15')",
                              "Total Cost": 8.16
                            }
                          ],
                          "Total Cost": 201850.78
                        },
                        {
                          "Node Type": "Index Scan",
                          "Relation Name": "customer",
                          "Index Name": "customer_pkey",
```

```
                        "Index Cond": "(c_custkey = orders.o_custkey)",
                        "Filter": "(c_mktsegment = 'BUILDING')",
                        "Total Cost": 0.47
                      }
                    ],
                    "Total Cost": 201942
                  }
                ],
                "Total Cost": 201943.1
              }
            ],
            "Total Cost": 201944.14
          }
        ],
        "Total Cost": 201945.06
      }
    ],
    "Total Cost": 201944.99
  },
  "Triggers": []
}
]
```

D.5 Cenário 2 – Q3 – TPCH-I


```
[
  {
    "Execution Time": 0.092,
    "Planning Time": 0.763,
    "Plan": {
      "Node Type": "Limit",
      "Plans": [
        {
          "Node Type": "Sort",
          "Sort Key": ["(sum((lineitem.l_extendedprice * ('1' -
lineitem.l_discount)))) DESC", "orders.o_orderdate"],
          "Plans": [
            {
              "Node Type": "Aggregate",
              "Group Key": ["lineitem.l_orderkey", "orders.o_orderdate",
                "orders.o_shippriority"],
              "Plans": [
                {
                  "Node Type": "Sort",
                  "Sort Key": ["lineitem.l_orderkey", "orders.o_orderdate",
                    "orders.o_shippriority"],
                  "Plans": [
                    {
                      "Node Type": "Nested Loop",
                      "Plans": [
                        {
                          "Node Type": "Nested Loop",
                          "Plans": [
                            {
                              "Node Type": "Index Scan",
                              "Relation Name": "lineitem",
                              "Index Name": "i_lineitem_shipdate"
                              "Index Cond": "(l_shipdate > '1999-03-15')",
                              "Total Cost": 8.45
                            },
                            {
                              "Node Type": "Index Scan",
                              "Relation Name": "orders",
                              "Index Name": "orders_pkey"
                              "Index Cond": "(o_orderkey = lineitem.l_orderkey)",
                              "Filter": "(o_orderdate < '1994-03-15')",
                              "Total Cost": 8.45
                            }
                          ],
                          "Total Cost": 16.91
                        },
                        {
                          "Node Type": "Index Scan",
                          "Relation Name": "customer",
```

```
                        "Index Name": "customer_pkey",
                        "Index Cond": "(c_custkey = orders.o_custkey)",
                        "Filter": "(c_mktsegment = 'BUILDING')",
                        "Total Cost": 0.47
                      }
                    ],
                    "Total Cost": 17.39
                  }
                ],
                "Total Cost": 17.41
              }
            ],
            "Total Cost": 17.43
          }
        ],
        "Total Cost": 17.45
      }
    ],
    "Total Cost": 17.45
  },
  "Triggers": []
 }
]
```

D.6 Cenário 2 – Q3 – TPCH-P


```
[
  {
    "Execution Time": 1809.465,
    "Planning Time": 0.95,
    "Plan": {
      "Node Type": "Limit",
      "Plans": [
        {
          "Node Type": "Sort",
          "Sort Key": ["(sum((lineitem.l_extendedprice * ('1' -
lineitem.l_discount)))) DESC", "orders.o_orderdate"],
          "Plans": [
            {
              "Node Type": "Aggregate",
              "Group Key": ["lineitem.l_orderkey", "orders.o_orderdate",
                "orders.o_shippriority"],
              "Plans": [
                {
                  "Node Type": "Sort",
                  "Sort Key": ["lineitem.l_orderkey", "orders.o_orderdate",
                    "orders.o_shippriority"],
                  "Plans": [
                    {
                      "Node Type": "Nested Loop",
                      "Plans": [
                        {
                          "Node Type": "Nested Loop",
                          "Plans": [
                            {
                              "Node Type": "Append",
                              "Plans": [
                                {
                                  "Node Type": "Seq Scan",
                                  "Relation Name": "lineitem",
                                  "Filter": "(l_shipdate > '1999-03-15')",
                                  "Total Cost": 0
                                },
                                {
                                  "Node Type": "Seq Scan",
                                  "Relation Name": "lineitem_p1",
                                  "Filter": "(l_shipdate > '1999-03-15')",
                                  "Total Cost": 37441.74
                                },
                                {
                                  "Node Type": "Seq Scan",
                                  "Relation Name": "lineitem_p2",
                                  "Filter": "(l_shipdate > '1999-03-15')",
                                  "Total Cost": 37687.6
                                },
```

```
                                {
                                  "Node Type": "Seq Scan",
                                  "Relation Name": "lineitem_p3",
                                  "Filter": "(l_shipdate > '1999-03-15')",
                                  "Total Cost": 41420.76
                                },
                                {
                                  "Node Type": "Seq Scan",
                                  "Relation Name": "lineitem_p4",
                                  "Filter": "(l_shipdate > '1999-03-15')",
                                  "Total Cost": 38573.03
                                },
                                {
                                  "Node Type": "Seq Scan",
                                  "Relation Name": "lineitem_p5",
                                  "Filter": "(l_shipdate > '1999-03-15')",
                                  "Total Cost": 42095.06
                                }
                              ],
                              "Total Cost": 197218.19
                            },
                            {
                              "Node Type": "Append",
                              "Plans": [
                                {
                                  "Node Type": "Seq Scan",
                                  "Relation Name": "orders",
                                  "Filter": "((o_orderdate < '1994-03-15') AND
(lineitem.l_orderkey = o_orderkey))",
                                  "Total Cost": 0
                                },
                                {
                                  "Node Type": "Index Scan",
                                  "Relation Name": "orders_p1",
                                  "Index Name": "i_orders_p1",
                                  "Index Cond": "((o_orderkey =
lineitem.l_orderkey) AND (o_orderdate < '1994-03-15'))",
                                  "Total Cost": 7.38
                                },
                                {
                                  "Node Type": "Index Scan",
                                  "Relation Name": "orders_p2",
                                  "Index Name": "i_orders_p2",
                                  "Index Cond": "((o_orderkey =
lineitem.l_orderkey) AND (o_orderdate < '1994-03-15'))",
                                  "Total Cost": 7.39
                                }
                              ],
                              "Total Cost": 14.78
                            }
                          ],
```

```
                        "Total Cost": 203984.34
                      },
                      {
                        "Node Type": "Index Scan",
                        "Relation Name": "customer",
                        "Index Name": "customer_pkey",
                        "Index Cond": "(c_custkey = orders.o_custkey)",
                        "Filter": "(c_mktsegment = 'BUILDING')",
                        "Total Cost": 0.47
                      }
                    ],
                    "Total Cost": 204205.09
                  }
                ],
                "Total Cost": 204208.45
              }
            ],
            "Total Cost": 204211.06
          }
        ],
        "Total Cost": 204213.35
      }
    ],
    "Total Cost": 204213.14
  },
  "Triggers": []
 }
]
```

D.7 Cenário 2 – Q8 – TPCH


```
[
  {
    "Execution Time": 3502.742,
    "Planning Time": 15.051,
    "Plan": {
      "Node Type": "Aggregate",
      "Group Key": ["orders.o_orderdate"],
      "Plans": [
        {
          "Node Type": "Sort",
          "Sort Key": ["orders.o_orderdate"],
          "Plans": [
            {
              "Node Type": "Nested Loop",
              "Plans": [
                {
                  "Node Type": "Nested Loop",
                  "Plans": [
                    {
                      "Node Type": "Nested Loop",
                      "Plans": [
                        {
                          "Node Type": "Nested Loop",
                          "Plans": [
                            {
                              "Node Type": "Hash Join",
                              "Hash Cond": "(orders.o_custkey =
customer.c_custkey)",
                              "Plans": [
                                {
                                  "Node Type": "Seq Scan",
                                  "Relation Name": "orders",
                                  "Filter": "((o_orderdate >= '1995-01-01') AND
(o_orderdate <= '1996-12-31'))",
                                  "Total Cost": 50326
                                },
                                {
                                  "Plans": [
                                    {
                                      "Node Type": "Hash Join",
                                      "Hash Cond": "(customer.c_nationkey =
n1.n_nationkey)",
                                      "Plans": [
                                        {
                                          "Node Type": "Seq Scan",
                                          "Relation Name": "customer",
                                          "Total Cost": 5226
                                        },
```

```
                                        {
                                          "Node Type": "Hash",
                                          "Plans": [
                                            {
                                              "Node Type": "Hash Join",
                                              "Hash Cond": "(n1.n_regionkey =
region.r_regionkey)",
                                              "Plans": [
                                                {
                                                  "Node Type": "Seq Scan",
                                                  "Relation Name": "nation",
                                                  "Total Cost": 11.7
                                                },
                                                {
                                                  "Plans": [
                                                    {
                                                      "Node Type": "Seq Scan",
                                                      "Filter": "(r_name =
'AMERICA')",
                                                      "Relation Name": "region",
                                                      "Total Cost": 12.13
                                                    }
                                                  ],
                                                  "Total Cost": 12.13
                                                }
                                              ],
                                              "Total Cost": 24.48
                                            }
                                          ],
                                          "Total Cost": 24.48
                                        }
                                      ],
                                      "Total Cost": 5821.82
                                    }
                                  ],
                                  "Total Cost": 5821.82
                                }
                              ],
                              "Total Cost": 57880.87
                            },
                            {
                              "Node Type": "Index Scan",
                              "Relation Name": "lineitem",
                              "Index Name": "lineitem_pkey"
                              "Index Cond": "(l_orderkey = orders.o_orderkey)",
                              "Total Cost": 2.02
                            }
                          ],
                          "Total Cost": 63696.12
                        },
                        {
```

```
                        "Node Type": "Index Scan",
                        "Relation Name": "part",
                        "Index Name": "part_pkey"
                        "Index Cond": "(p_partkey = lineitem.l_partkey)",
                        "Filter": "((p_type) = 'ECONOMY ANODIZED STEEL')",
                        "Total Cost": 0.44
                      }
                    ],
                    "Total Cost": 68517.34
                  },
                  {
                    "Node Type": "Index Scan",
                    "Relation Name": "supplier",
                    "Index Name": "supplier_pkey"
                    "Index Cond": "(s_suppkey = lineitem.l_suppkey)",
                    "Total Cost": 0.3
                  }
                ],
                "Total Cost": 68539.54
              },
              {
                "Node Type": "Index Scan",
                "Relation Name": "nation",
                "Index Name": "nation_pkey"
                "Index Cond": "(n_nationkey = supplier.s_nationkey)",
                "Total Cost": 0.17
              }
            ],
            "Total Cost": 68552.12
          }
        ],
        "Total Cost": 68554.48
      }
    ],
    "Total Cost": 68557.15
  },
  "Triggers": []
 }
]
```

D.8 Cenário 2 – Q8 – TPCH-I


```
[
  {
    "Execution Time": 3292.593,
    "Planning Time": 1.862,
    "Plan": {
      "Node Type": "Aggregate",
      "Group Key": ["orders.o_orderdate"],
      "Plans": [
        {
          "Node Type": "Sort",
          "Sort Key": ["orders.o_orderdate"],
          "Plans": [
            {
              "Node Type": "Nested Loop",
              "Plans": [
                {
                  "Node Type": "Nested Loop",
                  "Plans": [
                    {
                      "Node Type": "Nested Loop",
                      "Plans": [
                        {
                          "Node Type": "Nested Loop",
                          "Plans": [
                            {
                              "Node Type": "Hash Join",
                              "Hash Cond": "(orders.o_custkey =
customer.c_custkey)",
                              "Plans": [
                                {
                                  "Node Type": "Bitmap Heap Scan",
                                  "Relation Name": "orders",
                                  "Plans": [
                                    {
                                      "Node Type": "Bitmap Index Scan",
                                      "Index Name":
"i_orders_orderdate_shippriority"
                                      "Index Cond": "((o_orderdate >= '1995-01-
01') AND (o_orderdate <= '1996-12-31'))",
                                      "Total Cost": 9485.62
                                    }
                                  ],
                                  "Total Cost": 44206.43
                                },
                                {
                                  "Node Type": "Hash",
                                  "Plans": [
                                    {
                                      "Node Type": "Hash Join",
```

```
                                    "Hash Cond": "(customer.c_nationkey =
n1.n_nationkey)",
                                    "Plans": [
                                      {
                                        "Node Type": "Seq Scan",
                                        "Relation Name": "customer",
                                        "Total Cost": 5226
                                      },
                                      {
                                        "Node Type": "Hash",
                                        "Plans": [
                                          {
                                            "Node Type": "Hash Join",
                                            "Hash Cond": "(n1.n_regionkey =
region.r_regionkey)",
                                            "Plans": [
                                              {
                                                "Node Type": "Seq Scan",
                                                "Relation Name": "nation",
                                                "Total Cost": 11.7
                                              },
                                              {
                                                "Node Type": "Hash",
                                                "Plans": [
                                                  {
                                                    "Node Type": "Seq Scan",
                                                    "Relation Name": "region",
                                                    "Filter": "(r_name =
'AMERICA')",
                                                    "Total Cost": 12.13
                                                  }
                                                ],
                                                "Total Cost": 12.13
                                              }
                                            ],
                                            "Total Cost": 24.48
                                          }
                                        ],
                                        "Total Cost": 24.48
                                      }
                                    ],
                                    "Total Cost": 5821.82
                                  }
                                ],
                                "Total Cost": 5821.82
                              }
                            ],
                            "Total Cost": 51761.3
                          },
                          {
                            "Node Type": "Index Scan",
```

```
                            "Relation Name": "lineitem",
                            "Index Name": "lineitem_pkey"
                            "Index Cond": "(l_orderkey = orders.o_orderkey)",
                            "Total Cost": 2.02
                          }
                        ],
                        "Total Cost": 57576.55
                      },
                      {
                        "Node Type": "Index Scan",
                        "Relation Name": "part",
                        "Index Name": "part_pkey"
                        "Index Cond": "(p_partkey = lineitem.l_partkey)",
                        "Filter": "((p_type) = 'ECONOMY ANODIZED STEEL')",
                        "Total Cost": 0.44
                      }
                    ],
                    "Total Cost": 62397.77
                  },
                  {
                    "Node Type": "Index Scan",
                    "Relation Name": "supplier",
                    "Index Name": "supplier_pkey"
                    "Index Cond": "(s_suppkey = lineitem.l_suppkey)",
                    "Total Cost": 0.3
                  }
                ],
                "Total Cost": 62419.97
              },
              {
                "Node Type": "Index Scan",
                "Relation Name": "nation",
                "Index Name": "nation_pkey"
                "Index Cond": "(n_nationkey = supplier.s_nationkey)",
                "Total Cost": 0.17
              }
            ],
            "Total Cost": 62432.56
          }
        ],
        "Total Cost": 62434.92
      }
    ],
    "Total Cost": 62437.58
  },
  "Triggers": []
 }
]
```

D.9 Cenário 2 – Q8 – TPCH-P


```
[
  {
    "Execution Time": 2610.737,
    "Planning Time": 1.493,
    "Plan": {
      "Node Type": "Aggregate",
      "Group Key": ["orders.o_orderdate"],
      "Plans": [
        {
          "Node Type": "Sort",
          "Sort Key": [“orders.o_orderdate"],
          "Plans": [
            {
              "Node Type": "Hash Join",
              "Plans": [
                {
                  "Node Type": "Nested Loop",
                  "Plans": [
                    {
                      "Node Type": "Merge Join",
                      "Merge Cond": "(orders.o_orderkey = lineitem.l_orderkey)",
                      "Plans": [
                        {
                          "Node Type": "Sort",
                          "Sort Key": ["orders.o_orderkey"],
                          "Plans": [
                            {
                              "Node Type": "Hash Join",
                              "Hash Cond": "(orders.o_custkey =
customer.c_custkey)",
                              "Plans": [
                                {
                                  "Node Type": "Append",
                                  "Plans": [
                                    {
                                      "Node Type": "Seq Scan",
                                      "Relation Name": "orders",
                                      "Filter": "((o_orderdate >= '1995-01-01')
AND (o_orderdate <= '1996-12-31'))",
                                      "Total Cost": 0
                                    },
                                    {
                                      "Node Type": "Seq Scan",
                                      "Relation Name": "orders_p3",
                                      "Filter": "((o_orderdate >= '1995-01-01')
AND (o_orderdate <= '1996-12-31'))",
                                      "Total Cost": 10566.7
                                    },
                                    {
```

```
                                    "Node Type": "Seq Scan",
                                    "Relation Name": "orders_p4",
                                    "Filter": "((o_orderdate >= '1995-01-01')
AND (o_orderdate <= '1996-12-31'))",
                                    "Total Cost": 9827.03
                                  }
                                ],
                                "Total Cost": 20393.73
                              },
                              {
                                "Node Type": "Hash",
                                "Plans": [
                                  {
                                    "Node Type": "Hash Join",
                                    "Hash Cond": "(customer.c_nationkey =
n1.n_nationkey)",
                                    "Plans": [
                                      {
                                        "Node Type": "Seq Scan",
                                        "Relation Name": "customer",
                                        "Total Cost": 5213
                                      },
                                      {
                                        "Node Type": "Hash",
                                        "Plans": [
                                          {
                                            "Node Type": "Hash Join",
                                            "Hash Cond": "(n1.n_regionkey =
region.r_regionkey)",
                                            "Plans": [
                                              {
                                                "Node Type": "Seq Scan",
                                                "Relation Name": "nation",
                                                "Total Cost": 11.7
                                              },
                                              {
                                                "Node Type": "Hash",
                                                "Plans": [
                                                  {
                                                    "Node Type": "Seq Scan",
                                                    "Relation Name": "region",
                                                    "Filter": "(r_name =
'AMERICA')",
                                                    "Total Cost": 12.13
                                                  }
                                                ],
                                                "Total Cost": 12.13
                                              }
                                            ],
                                            "Total Cost": 24.48
                                          }
```

```
                                        ],
                                        "Total Cost": 24.48
                                      }
                                    ],
                                    "Total Cost": 5808.82
                                  }
                                ],
                                "Total Cost": 5808.82
                              }
                            ],
                            "Total Cost": 27955.48
                          }
                        ],
                        "Total Cost": 28115.45
                      },
                      {
                        "Node Type": "Sort",
                        "Sort Key": ["lineitem.l_orderkey"],
                        "Plans": [
                          {
                            "Node Type": "Hash Join",
                            "Hash Cond": "(lineitem.l_partkey =
part.p_partkey)",
                            "Plans": [
                              {
                                "Node Type": "Append",
                                "Plans": [
                                  {
                                    "Node Type": "Seq Scan",
                                    "Relation Name": "lineitem",
                                    "Total Cost": 0
                                  },
                                  {
                                    "Node Type": "Seq Scan",
                                    "Relation Name": "lineitem_p1",
                                    "Total Cost": 34593.59
                                  },
                                  {
                                    "Node Type": "Seq Scan",
                                    "Relation Name": "lineitem_p2",
                                    "Total Cost": 34820.48
                                  },
                                  {
                                    "Node Type": "Seq Scan",
                                    "Relation Name": "lineitem_p3",
                                    "Total Cost": 38269.61
                                  },
                                  {
                                    "Node Type": "Seq Scan",
                                    "Relation Name": "lineitem_p4",
                                    "Total Cost": 35638.62
```

```
                                    },
                                    {
                                      "Node Type": "Seq Scan",
                                      "Relation Name": "lineitem_p5",
                                      "Total Cost": 38892.85
                                    }
                                  ],
                                  "Total Cost": 182215.15
                                },
                                {
                                  "Node Type": "Hash",
                                  "Plans": [
                                    {
                                      "Node Type": "Seq Scan",
                                      "Relation Name": "part",
                                      "Filter": "((p_type) = 'ECONOMY ANODIZED
STEEL')",
                                      "Total Cost": 6630
                                    }
                                  ],
                                  "Total Cost": 6630
                                }
                              ],
                              "Total Cost": 211766.35
                            }
                          ],
                          "Total Cost": 214923.72
                        }
                      ],
                      "Total Cost": 251015.49
                    },
                    {
                      "Node Type": "Index Scan",
                      "Relation Name": "supplier",
                      "Index Name": "supplier_pkey"
                      "Index Cond": "(s_suppkey = lineitem.l_suppkey)",
                      "Total Cost": 0.3
                    }
                  ],
                  "Total Cost": 251088.97
                },
                {
                  "Node Type": "Hash",
                  "Plans": [
                    {
                      "Node Type": "Seq Scan",
                      "Relation Name": "nation",
                      "Total Cost": 11.7
                    }
                  ],
                  "Total Cost": 11.7
```

```
                }
              ],
              "Hash Cond": "(supplier.s_nationkey = n2.n_nationkey)",
              "Total Cost": 251106.03
            }
          ],
          "Total Cost": 251115.87
        }
      ],
      "Total Cost": 251124.07
    },
    "Triggers": []
  }
]
```

D.10 Cenário 3 – Q5 – TPCH


```
[
  {
    "Execution Time": 1342.671,
    "Planning Time": 1.656,
    "Plan": {
      "Node Type": "Sort",
      "Sort Key": ["(sum((lineitem.l_extendedprice * ('1' - lineitem.l_discount
)))) DESC"],
      "Plans": [
        {
          "Node Type": "Aggregate",
          "Group Key": ["nation.n_name"],
          "Plans": [
            {
              "Node Type": "Sort",
              "Sort Key": ["nation.n_name"],
              "Plans": [
                {
                  "Node Type": "Hash Join",
                  "Hash Cond": "((lineitem.l_suppkey = supplier.s_suppkey) AND
(customer.c_nationkey = supplier.s_nationkey))",
                  "Plans": [
                    {
                      "Node Type": "Nested Loop",
                      "Plans": [
                        {
                          "Node Type": "Hash Join",
                          "Hash Cond": "(orders.o_custkey = customer.c_custkey)",
                          "Plans": [
                            {
                              "Node Type": "Seq Scan",
                              "Filter": "((o_orderdate >= '1994-01-01') AND
(o_orderdate < '1995-01-01'))",
                              "Relation Name": "orders",
                              "Total Cost": 50326
                            },
                            {
                              "Node Type": "Hash",
                              "Plans": [
                                {
                                  "Node Type": "Hash Join",
                                  "Hash Cond": "(customer.c_nationkey =
nation.n_nationkey)",
                                  "Plans": [
                                    {
                                      "Node Type": "Seq Scan",
                                      "Relation Name": "customer",
                                      "Total Cost": 5226
                                    },
```

```
                                    {
                                      "Node Type": "Hash",
                                      "Plans": [
                                        {
                                          "Node Type": "Hash Join",
                                          "Hash Cond": "(nation.n_regionkey =
region.r_regionkey)",
                                          "Plans": [
                                            {
                                              "Node Type": "Seq Scan",
                                              "Relation Name": "nation",
                                              "Total Cost": 11.7
                                            },
                                            {
                                              "Node Type": "Hash",
                                              "Plans": [
                                                {
                                                  "Node Type": "Seq Scan",
                                                  "Filter": "(r_name = 'ASIA')",
                                                  "Relation Name": "region",
                                                  "Total Cost": 12.13
                                                }
                                              ],
                                              "Total Cost": 12.13
                                            }
                                          ],
                                          "Total Cost": 24.48
                                        }
                                      ],
                                      "Total Cost": 24.48
                                    }
                                  ],
                                  "Total Cost": 5821.82
                                }
                              ],
                              "Total Cost": 5821.82
                            }
                          ],
                          "Total Cost": 57014.43
                        },
                        {
                          "Node Type": "Index Scan",
                          "Relation Name": "lineitem",
                          "Index Name": "lineitem_pkey"
                          "Index Cond": "(l_orderkey = orders.o_orderkey)",
                          "Total Cost": 3.27
                        }
                      ],
                      "Total Cost": 61554.36
                    },
                    {
```

```
                    "Node Type": "Hash",
                    "Plans": [
                      {
                        "Node Type": "Seq Scan",
                        "Relation Name": "supplier",
                        "Total Cost": 332
                      }
                    ],
                    "Total Cost": 332
                  }
                ],
                "Total Cost": 62078.12
              }
            ],
            "Total Cost": 62086.8
          }
        ],
        "Total Cost": 62091.03
      }
    ],
    "Total Cost": 62097.75
  },
  "Triggers": []
}
]
```

D.11 Cenário 3 – Q5 – TPCH-I


```
[
  {
    "Execution Time": 1398.552,
    "Planning Time": 1.759,
    "Plan": {
      "Node Type": "Sort",
      "Sort Key": ["(sum((lineitem.l_extendedprice * ('1' - lineitem.l_discount
)))) DESC"],
      "Plans": [
        {
          "Node Type": "Aggregate",
          "Group Key": ["nation.n_name"],
          "Plans": [
            {
              "Node Type": "Sort",
              "Sort Key": ["nation.n_name"],
              "Plans": [
                {
                  "Node Type": "Hash Join",
                  "Hash Cond": "((lineitem.l_suppkey = supplier.s_suppkey) AND
(customer.c_nationkey = supplier.s_nationkey))",
                  "Plans": [
                    {
                      "Node Type": "Nested Loop",
                      "Plans": [
                        {
                          "Node Type": "Hash Join",
                          "Hash Cond": "(orders.o_custkey = customer.c_custkey)",
                          "Plans": [
                            {
                              "Node Type": "Bitmap Heap Scan",
                              "Relation Name": "orders",
                              "Plans": [
                                {
                                  "Node Type": "Bitmap Index Scan",
                                  "Index Name": "i_orders_orderdate",
                                  "Index Cond": "((o_orderdate >= '1994-01-01')
AND (o_orderdate < '1995-01-01'))",
                                  "Total Cost": 4714.78
                                }
                              ],
                              "Total Cost": 35966.46
                            },
                            {
                              "Node Type": "Hash",
                              "Plans": [
                                {
                                  "Node Type": "Hash Join",
```

```
                                "Hash Cond": "(customer.c_nationkey =
nation.n_nationkey)",
                                "Plans": [
                                  {
                                    "Node Type": "Seq Scan",
                                    "Relation Name": "customer",
                                    "Total Cost": 5226
                                  },
                                  {
                                    "Node Type": "Hash",
                                    "Plans": [
                                      {
                                        "Node Type": "Hash Join",
                                        "Hash Cond": "(nation.n_regionkey =
region.r_regionkey)",
                                        "Plans": [
                                          {
                                            "Node Type": "Seq Scan",
                                            "Relation Name": "nation",
                                            "Total Cost": 11.7
                                          },
                                          {
                                            "Node Type": "Hash",
                                            "Plans": [
                                              {
                                                "Node Type": "Seq Scan",
                                                "Relation Name": "region",
                                                "Filter": "(r_name = 'ASIA')",
                                                "Total Cost": 12.13
                                              }
                                            ],
                                            "Total Cost": 12.13
                                          }
                                        ],
                                        "Total Cost": 24.48
                                      }
                                    ],
                                    "Total Cost": 24.48
                                  }
                                ],
                                "Total Cost": 5821.82
                              }
                            ],
                            "Total Cost": 5821.82
                          }
                        ],
                        "Total Cost": 42654.9
                      },
                      {
                        "Node Type": "Index Scan",
                        "Relation Name": "lineitem",
```

```
                        "Index Name": "lineitem_pkey",
                        "Index Cond": "(l_orderkey = orders.o_orderkey)",
                        "Total Cost": 3.27
                      }
                    ],
                    "Total Cost": 47194.82
                  },
                  {
                    "Node Type": "Hash",
                    "Plans": [
                      {
                        "Node Type": "Seq Scan",
                        "Relation Name": "supplier",
                        "Total Cost": 332
                      }
                    ],
                    "Total Cost": 332
                  }
                ],
                "Total Cost": 47718.58
              }
            ],
            "Total Cost": 47727.26
          }
        ],
        "Total Cost": 47731.49
      }
    ],
    "Total Cost": 47738.22
  },
  "Triggers": []
 }
]
```

D.12 Cenário 3 – Q5 – TPCH-P


```
[
  {
    "Execution Time": 8916.988,
    "Planning Time": 1.652,
    "Plan": {
      "Node Type": "Sort",
      "Sort Key": ["(sum((lineitem.l_extendedprice * ('1'-l_discount)))) DESC"],
      "Plans": [
        {
          "Node Type": "Aggregate", "Group Key": ["nation.n_name"],
          "Plans": [
            {
              "Node Type": "Sort",
              "Sort Key": ["nation.n_name"],
              "Plans": [
                {
                  "Node Type": "Nested Loop",
                  "Join Filter": "(supplier.s_nationkey = customer.c_nationkey)",
                  "Plans": [
                    {
                      "Node Type": "Nested Loop",
                      "Plans": [
                        {
                          "Node Type": "Hash Join",
                          "Hash Cond": "(lineitem.l_suppkey = s_suppkey)",
                          "Plans": [
                            {
                              "Node Type": "Append",
                              "Plans": [
                                {
                                  "Node Type": "Seq Scan",
                                  "Relation Name": "lineitem",
                                  "Total Cost": 0
                                },
                                {
                                  "Node Type": "Seq Scan",
                                  "Relation Name": "lineitem_p1",
                                  "Total Cost": 35149.36
                                },
                                {
                                  "Node Type": "Seq Scan",
                                  "Relation Name": "lineitem_p2",
                                  "Total Cost": 36797.92
                                },
                                {
                                  "Node Type": "Seq Scan",
                                  "Relation Name": "lineitem_p3",
                                  "Total Cost": 36308.16
                                },
```

```
                                {
                                  "Node Type": "Seq Scan",
                                  "Relation Name": "lineitem_p4",
                                  "Total Cost": 36093.5
                                },
                                {
                                  "Node Type": "Seq Scan",
                                  "Relation Name": "lineitem_p5",
                                  "Total Cost": 37865.21
                                }
                              ],
                              "Total Cost": 182214.15
                            },
                            {
                              "Node Type": "Hash",
                              "Plans": [
                                {
                                  "Node Type": "Hash Join",
                                  "Hash Cond": "(supplier.s_nationkey =
nation.n_nationkey)",
                                  "Plans": [
                                    {
                                      "Node Type": "Seq Scan",
                                      "Relation Name": "supplier",
                                      "Total Cost": 331
                                    },
                                    {
                                      "Node Type": "Hash",
                                      "Plans": [
                                        {
                                          "Node Type": "Hash Join",
                                          "Hash Cond": "(nation.n_regionkey =
region.r_regionkey)",
                                          "Plans": [
                                            {
                                              "Node Type": "Seq Scan",
                                              "Relation Name": "nation",
                                              "Total Cost": 11.7
                                            },
                                            {
                                              "Node Type": "Hash",
                                              "Plans": [
                                                {
                                                  "Node Type": "Seq Scan",
                                                  "Relation Name": "region",
                                                  "Filter": "(r_name = 'ASIA')",
                                                  "Total Cost": 12.13
                                                }
                                              ],
                                              "Total Cost": 12.13
                                            }
```

```
                                        ],
                                        "Total Cost": 24.48
                                      }
                                    ],
                                    "Total Cost": 24.48
                                  }
                                ],
                                "Total Cost": 393.59
                              }
                            ],
                            "Total Cost": 393.59
                          }
                        ],
                        "Total Cost": 205467.11
                      },
                      {
                        "Node Type": "Index Scan",
                        "Relation Name": "orders",
                        "Index Name": "orders_pkey"
                        "Index Cond": "(o_orderkey = lineitem.l_orderkey)",
                        "Filter": "((o_orderdate >= '1994-01-01') AND
(o_orderdate < '1995-01-01'))",
                        "Total Cost": 0.47,
                      }
                    ],
                    "Total Cost": 222455.77
                  },
                  {
                    "Node Type": "Index Scan",
                    "Relation Name": "customer",
                    "Index Name": "customer_pkey"
                    "Index Cond": "(c_custkey = orders.o_custkey)",
                    "Total Cost": 0.51,
                  }
                ],
                "Total Cost": 225248.65
              }
            ],
            "Total Cost": 225257.47
          }
        ],
        "Total Cost": 225261.73
      }
    ],
    "Total Cost": 225268.45
  },
  "Triggers": []
 }
]
```

D.13 Cenário 3 – Q12 – TPCH


```
[
  {
    "Execution Time": 3276.287,
    "Planning Time": 0.463,
    "Plan": {
      "Node Type": "Aggregate",
      "Group Key": ["lineitem.l_shipmode"],
      "Plans": [
        {
          "Node Type": "Sort", "Sort Key": ["lineitem.l_shipmode"],
          "Plans": [
            {
              "Node Type": "Merge Join",
              "Merge Cond": "(orders.o_orderkey = lineitem.l_orderkey)",
              "Plans": [
                {
                  "Node Type": "Index Scan",
                  "Relation Name": "orders",
                  "Index Name": "orders_pkey",
                  "Total Cost": 66789.43
                },
                {
                  "Node Type": "Sort",
                  "Sort Key": ["lineitem.l_orderkey"],
                  "Plans": [
                    {
                      "Node Type": "Seq Scan",
                      "Relation Name": "lineitem",
                      "Filter": "((l_shipmode = ANY ('{MAIL,SHIP}')) AND
(l_commitdate < l_receiptdate) AND (l_shipdate < l_commitdate) AND (l_receiptdate
>= '1994-01-01') AND (l_receiptdate < '1995-01-01'))",
                      "Total Cost": 257295.34
                    }
                  ],
                  "Total Cost": 259461.67
                }
              ],
              "Total Cost": 330351.82
            }
          ],
          "Total Cost": 332518.16
        }
      ],
      "Total Cost": 333014.11
    },
    "Triggers": []
  }
]
```

D.14 Cenário 3 – Q12 – TPCH-I


```
[
  {
    "Execution Time": 2141.209,
    "Planning Time": 1.821,
    "Plan": {
      "Node Type": "Sort",
      "Sort Key": ["lineitem.l_shipmode"],
      "Plans": [
        {
          "Node Type": "Aggregate",
          "Group Key": ["lineitem.l_shipmode"],
          "Plans": [
            {
              "Node Type": "Merge Join",
              "Merge Cond": "(orders.o_orderkey = lineitem.l_orderkey)",
              "Plans": [
                {
                  "Node Type": "Index Scan",
                  "Index Name": "orders_pkey",
                  "Total Cost": 66789.43
                },
                {
                  "Node Type": "Sort",
                  "Sort Key": ["lineitem.l_orderkey"],
                  "Plans": [
                    {
                      "Node Type": "Bitmap Heap Scan",
                      "Relation Name": "lineitem",
                      "Filter": "((l_commitdate < l_receiptdate) AND (l_shipdate
< l_commitdate))",
                      "Plans": [
                        {
                          "Node Type": "Bitmap Index Scan",
                          "Index Name": "i_lineitem_shipmode_receiptdate",
                          "Index Cond": "((l_shipmode = ANY ('{MAIL,SHIP}')) AND
(l_receiptdate >= '1994-01-01') AND (l_receiptdate < '1995-01-01'))",
                          "Total Cost": 7044.67
                        }
                      ],
                      "Total Cost": 135057.79
                    }
                  ],
                  "Total Cost": 137224.13
                }
              ],
              "Total Cost": 208114.27
            }
          ],
```

```
          "Total Cost": 208610.22
        }
      ],
      "Total Cost": 208610.34
    },
    "Triggers": []
  }
]
```

D.15 Cenário 3 – Q12 – TPCH-P


```
[
  {
    "Execution Time": 1259.206,
    "Planning Time": 0.543,
    "Plan": {
      "Node Type": "Sort",
      "Sort Key": ["lineitem.l_shipmode"],
      "Plans": [
        {
          "Node Type": "Aggregate",
          "Group Key": ["lineitem.l_shipmode"],
          "Plans": [
            {
              "Node Type": "Merge Join",
              "Merge Cond": "(orders.o_orderkey = lineitem.l_orderkey)",
              "Plans": [
                {
                  "Node Type": "Index Scan",
                  "Relation Name": "orders",
                  "Index Name": "orders_pkey"
                },
                {
                  "Node Type": "Sort",
                  "Sort Key": ["lineitem.l_orderkey"],
                  "Plans": [
                    {
                      "Node Type": "Append",
                      "Plans": [
                        {
                          "Node Type": "Seq Scan",
                          "Relation Name": "lineitem",
                          "Filter": "((l_shipmode = ANY ('{MAIL,SHIP}')) AND
(l_commitdate < l_receiptdate) AND (l_shipdate < l_commitdate) AND (l_receiptdate
>= '1994-01-01') AND (l_receiptdate < '1995-01-01'))",
                          "Total Cost": 0
                        },
                        {
                          "Node Type": "Seq Scan",
                          "Relation Name": "lineitem_p3",
                          "Filter": "((l_shipmode = ANY ('{MAIL,SHIP}')) AND
(l_commitdate < l_receiptdate) AND (l_shipdate < l_commitdate) AND (l_receiptdate
>= '1994-01-01') AND (l_receiptdate < '1995-01-01'))",
                          "Total Cost": 73527.22
                        }
                      ],
                      "Total Cost": 73527.22
                    }
                  ],
                  "Total Cost": 75802.11
```

```
              }
            ],
            "Total Cost": 146654.93
          }
        ],
        "Total Cost": 147175.47
      }
    ],
    "Total Cost": 147183.61
  },
  "Triggers": []
 }
]
```

D.16 Atualizações – U1 – TPCH / TPCH-P


```
[
  {
    "Execution Time": 30335.708,
    "Planning Time": 0.119,
    "Plan": {
      "Node Type": "ModifyTable",
      "Relation Name": "orders",
      "Operation": "Update",
      "Plans": [
        {
          "Node Type": "Seq Scan",
          "Relation Name": "orders",
          "Filter": "((o_orderdate >= '1994-06-01') AND

(o_orderdate <= '1997-10-01'))",
          "Total Cost": 76791.2
        }
      ],
      "Total Cost": 76791.2
    },
    "Triggers": []
  }
]
```

TPCH-P


```
[
  {
    "Execution Time": 33085.086,
    "Planning Time": 0.577,
    "Plan": {
      "Node Type": "ModifyTable",
      "Relation Name": "orders",
      "Operation": "Update",
      "Plans": [
        {
          "Node Type": "Seq Scan",
          "Relation Name": "orders",
          "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <= '1997-10-
01'))",
          "Total Cost": 0
        },
        {
          "Node Type": "Seq Scan",
          "Relation Name": "orders_p2",
          "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <= '1997-10-
01'))",
          "Total Cost": 10926
        },
        {
          "Node Type": "Seq Scan",
          "Relation Name": "orders_p3",
          "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <= '1997-10-
01'))",
          "Total Cost": 24962.22
        },{
          "Node Type": "Seq Scan",
          "Relation Name": "orders_p4",
          "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <= '1997-10-
01'))",
          "Total Cost": 21318.04
        },
        {
          "Node Type": "Seq Scan",
          "Relation Name": "orders_p5",
          "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <= '1997-10-
01'))",
          "Total Cost": 15852.28
        }
      ],
      "Total Cost": 73058.54
    },
    "Triggers": []
  }
]
```

D.17 Atualizações – U2 – TPCH / TPCH-P


```
[
  {
    "Execution Time": 155143.24,
    "Planning Time": 0.171,
    "Plan": {
      "Node Type": "ModifyTable",
      "Relation Name": "lineitem",
      "Operation": "Update",
      "Plans": [
        {
          "Filter": "((l_receiptdate >= '1994-06-01'::date) AND (l_receiptdate <=
'1997-10-01'::date))",
          "Node Type": "Seq Scan",
          "Relation Name": "lineitem",
          "Total Cost": 319651.37
        }
      ],
      "Total Cost": 319651.37
    },
    "Triggers": []
  }
]
```

TPCH-P


```
[
 {
   "Execution Time": 61540.602,
   "Planning Time": 5.336,
   "Plan": {
     "Node Type": "ModifyTable",
     "Relation Name": "lineitem",
     "Operation": "Update",
     "Plans": [
       {
         "Node Type": "Seq Scan",
         "Relation Name": "lineitem",
         "Filter": "(l_shipmode = ANY ('{MAIL,SHIP}'))",
         "Total Cost": 0
       },
       {
         "Node Type": "Seq Scan",
         "Relation Name": "lineitem_p3",
         "Filter": "(l_shipmode = ANY ('{MAIL,SHIP}'))",
         "Total Cost": 112873.3
       }
     "Total Cost": 112873.3
     ],
   },
   "Triggers": []
 }
]
```

D.18 Atualizações – U3 – TPCH / TPCH-P cenario 2 / cenario 3


```
[
  {
    "Execution Time": 2552907.847,
    "Planning Time": 33.02,
    "Plan": {
      "Node Type": "ModifyTable",
      "Operation": "Update",
      "Relation Name": "lineitem",
      "Plans": [
        {
          "Node Type": "Hash Join",
          "Hash Cond": "(l.l_orderkey = o.o_orderkey)",
          "Plans": [
            {
              "Node Type": "Seq Scan",
              "Relation Name": "lineitem",
              "Total Cost": 334546.97
            },
            {
              "Node Type": "Hash",
              "Plans": [
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 82129.9
                }
              ],
              "Total Cost": 82129.9
            }
          ],
          "Total Cost": 687922.91
        }
      ],
      "Total Cost": 687922.91
    },
    "Triggers": []
  }
]
```

TPCH-P cenário 2


```
[
  {
    "Execution Time": 589905.029,
    "Planning Time": 1144.641,
    "Plan": {
      "Node Type": "ModifyTable",
      "Relation Name": "lineitem",
      "Operation": "Update",
      "Plans": [
        {
          "Node Type": "Nested Loop",
          "Plans": [
            {
              "Node Type": "Seq Scan",
              "Relation Name": "lineitem",
              "Total Cost": 0
            },
            {
              "Node Type": "Append",
              "Plans": [
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01') AND (l.l_orderkey = o_orderkey))",
                  "Total Cost": 0
                },
                {
                  "Node Type": "Index Scan",
                  "Relation Name": "orders_p2",
                  "Index Name": "i_orders_p2"
                  "Index Cond": "((o_orderkey = l.l_orderkey) AND (o_orderdate >=
'1994-06-01') AND (o_orderdate <= '1997-10-01'))",
                  "Total Cost": 8.44
                },
                {
                  "Node Type": "Index Scan",
                  "Relation Name": "orders_p3",
                  "Index Name": "i_orders_p3"
                  "Index Cond": "((o_orderkey = l.l_orderkey) AND (o_orderdate >=
'1994-06-01') AND (o_orderdate <= '1997-10-01'))",
                  "Total Cost": 8.44
                },
                {
                  "Node Type": "Index Scan",
                  "Relation Name": "orders_p4",
                  "Index Name": "i_orders_p4"
                  "Index Cond": "((o_orderkey = l.l_orderkey) AND (o_orderdate >=
'1994-06-01') AND (o_orderdate <= '1997-10-01'))",
```

```
                  "Total Cost": 8.44
                },
                {
                  "Node Type": "Index Scan",
                  "Relation Name": "orders_p5",
                  "Index Name": "i_orders_p5"
                  "Index Cond": "((o_orderkey = l.l_orderkey) AND (o_orderdate >=
'1994-06-01') AND (o_orderdate <= '1997-10-01'))",
                  "Total Cost": 8.44
                }
              ],
              "Total Cost": 33.78
            }
          ],
          "Total Cost": 33.83
        },
        {
          "Node Type": "Hash Join",
          "Hash Cond": "(o.o_orderkey = l_1.l_orderkey)",
          "Plans": [
            {
              "Node Type": "Append",
              "Plans": [
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 0
                },
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders_p2",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 10926
                },
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders_p3",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 24962.22
                },
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders_p4",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 21318.04
                },
```

```
              {
                "Node Type": "Seq Scan",
                "Relation Name": "orders_p5",
                "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                "Total Cost": 15852.28
              }
            ],
            "Total Cost": 73058.54
          },
          {
            "Node Type": "Hash",
            "Plans": [
              {
                "Node Type": "Seq Scan",
                "Relation Name": "lineitem_p1",
                "Total Cost": 34593.59
              }
            ],
            "Total Cost": 34593.59
          }
        ],
        "Total Cost": 384835.08
      },
      {
        "Node Type": "Hash Join",
        "Hash Cond": "(o.o_orderkey = l_2.l_orderkey)",
        "Plans": [
          {
            "Node Type": "Append",
            "Plans": [
              {
                "Node Type": "Seq Scan",
                "Relation Name": "orders",
                "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                "Total Cost": 0
              },
              {
                "Node Type": "Seq Scan",
                "Relation Name": "orders_p2",
                "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                "Total Cost": 10926
              },
              {
                "Node Type": "Seq Scan",
                "Relation Name": "orders_p3",
                "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                "Total Cost": 24962.22
```

```
              },
              {
                "Node Type": "Seq Scan",
                "Relation Name": "orders_p4",
                "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                "Total Cost": 21318.04
              },
              {
                "Node Type": "Seq Scan",
                "Relation Name": "orders_p5",
                "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                "Total Cost": 15852.28
              }
            ],
            "Total Cost": 73058.54
          },
          {
            "Node Type": "Hash",
            "Plans": [
              {
                "Node Type": "Seq Scan",
                "Relation Name": "lineitem_p2",
                "Total Cost": 40452.48
              }
            ],
            "Total Cost": 40452.48
          }
        ],
        "Total Cost": 392323.92
      },
      {
        "Node Type": "Hash Join",
        "Hash Cond": "(o.o_orderkey = l_3.l_orderkey)",
        "Plans": [
          {
            "Node Type": "Append",
            "Plans": [
              {
                "Node Type": "Seq Scan",
                "Relation Name": "orders",
                "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                "Total Cost": 0
              },
              {
                "Node Type": "Seq Scan",
                "Relation Name": "orders_p2",
                "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
```

```
                  "Total Cost": 10926
                },
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders_p3",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 24962.22
                },
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders_p4",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 21318.04
                },
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders_p5",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 15852.28
                }
              ],
              "Total Cost": 73058.54
            },
            {
              "Node Type": "Hash",
              "Plans": [
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "lineitem_p3",
                  "Total Cost": 63999.49
                }
              ],
              "Total Cost": 63999.49
            }
          ],
          "Total Cost": 462819.86
        },
        {
          "Node Type": "Hash Join",
          "Hash Cond": "(o.o_orderkey = l_4.l_orderkey)",
          "Plans": [
            {
              "Node Type": "Append",
              "Plans": [
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders",
```

```
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 0
                },
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders_p2",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 10926
                },
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders_p3",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 24962.22
                },
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders_p4",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 21318.04
                },
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders_p5",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 15852.28
                }
              ],
              "Total Cost": 73058.54
            },
            {
              "Node Type": "Hash",
              "Plans": [
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "lineitem_p4",
                  "Total Cost": 59594.52
                }
              ],
              "Total Cost": 59594.52
            }
          ],
          "Total Cost": 392993.75
        },
        {
          "Node Type": "Hash Join",
```

```
          "Hash Cond": "(o.o_orderkey = l_5.l_orderkey)",
          "Plans": [
            {
              "Node Type": "Append",
              "Plans": [
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 0
                },
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders_p2",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 10926
                },
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders_p3",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 24962.22
                },
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders_p4",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 21318.04
                },
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders_p5",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 15852.28
                }
              ],
              "Total Cost": 73058.54
            },
            {
              "Node Type": "Hash",
              "Plans": [
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "lineitem_p5",
                  "Total Cost": 49466.51
                }
```

```
              ],
              "Total Cost": 49466.51
            }
          ],
          "Total Cost": 435864.63
        }
      ],
      "Total Cost": 2068871.06
    },
    "Triggers": []
  }
]
```

TPCH-P cenário 3


```
[
  {
    "Execution Time": 1135734.336,
    "Planning Time": 1487.278,
    "Plan": {
      "Node Type": "ModifyTable",
      "Relation Name": "lineitem",
      "Operation": "Update",
      "Plans": [
        {
          "Node Type": "Nested Loop",
          "Plans": [
            {
              "Node Type": "Seq Scan",
              "Relation Name": "lineitem",
              "Total Cost": 0
            },
            {
              "Node Type": "Index Scan",
              "Relation Name": "orders",
              "Index Name": "orders_pkey"
              "Index Cond": "(o_orderkey = l.l_orderkey)",
              "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
              "Total Cost": 8.45
            }
          ],
          "Total Cost": 8.46
        },
        {
          "Node Type": "Hash Join",
          "Hash Cond": "(l_1.l_orderkey = o.o_orderkey)",
          "Plans": [
            {
              "Node Type": "Seq Scan",
              "Relation Name": "lineitem_p1",
              "Total Cost": 35149.36
            },
            {
              "Node Type": "Hash",
              "Plans": [
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 50269
                }
              ],
```

```
              "Total Cost": 50269
            }
          ],
          "Total Cost": 151030.31
        },
        {
          "Node Type": "Hash Join",
          "Hash Cond": "(l_2.l_orderkey = o.o_orderkey)",
          "Plans": [
            {
              "Node Type": "Seq Scan",
              "Relation Name": "lineitem_p2",
              "Total Cost": 51582.96
            },
            {
              "Node Type": "Hash",
              "Plans": [
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 50269
                }
              ],
              "Total Cost": 50269
            }
          ],
          "Total Cost": 169378.04
        },
        {
          "Node Type": "Hash Join",
          "Hash Cond": "(l_3.l_orderkey = o.o_orderkey)",
          "Plans": [
            {
              "Node Type": "Seq Scan",
              "Relation Name": "lineitem_p3",
              "Total Cost": 84842.59
            },
            {
              "Node Type": "Hash",
              "Plans": [
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 50269
                }
              ],
              "Total Cost": 50269
```

```
            }
          ],
          "Total Cost": 202333.79
        },
        {
          "Node Type": "Hash Join",
          "Hash Cond": "(l_4.l_orderkey = o.o_orderkey)",
          "Plans": [
            {
              "Node Type": "Seq Scan",
              "Relation Name": "lineitem_p4",
              "Total Cost": 84352.81
            },
            {
              "Node Type": "Hash",
              "Plans": [
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 50269
                }
              ],
              "Total Cost": 50269
            }
          ],
          "Total Cost": 201610.78
        },
        {
          "Node Type": "Hash Join",
          "Hash Cond": "(l_5.l_orderkey = o.o_orderkey)",
          "Plans": [
            {
              "Node Type": "Seq Scan",
              "Relation Name": "lineitem_p5",
              "Total Cost": 49858.69
            },
            {
              "Node Type": "Hash",
              "Plans": [
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "orders",
                  "Filter": "((o_orderdate >= '1994-06-01') AND (o_orderdate <=
'1997-10-01'))",
                  "Total Cost": 50269
                }
              ],
              "Total Cost": 50269
            }
```

```
          ],
          "Total Cost": 169536.95
        }
      ],
      "Total Cost": 893898.32
    },
    "Triggers": []
  }
]
```