UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

STB-INDEX: UM ÍNDICE BASEADO EM BITMAP PARA DATA WAREHOUSE ESPAÇO-TEMPORAL

RENATA MIWA TSURUDA

ORIENTADOR: PROF. DR. RICARDO RODRIGUES CIFERRI

CO-ORIENTADORA: PROF.ª DR.ª VALÉRIA CESÁRIO TIMES

São Carlos - SP
Dezembro/2012

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

STB-INDEX: UM ÍNDICE BASEADO EM BITMAP PARA DATA WAREHOUSE ESPAÇO-TEMPORAL

RENATA MIWA TSURUDA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software e Banco de Dados.
Orientador: Prof. Dr. Ricardo Rodrigues Ciferri.
Co-Orientadora: Prof.ª Dr.ª Valéria Cesário Times.

São Carlos - SP
Dezembro/2012

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

T882si

Tsuruda, Renata Miwa.
STB-index : um índice baseado em bitmap para data warehouse espaço-temporal / Renata Miwa Tsuruda. -- São Carlos : UFSCar, 2013.
129 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2012.

1. Banco de dados. 2. *Data warehouse*. 3. Índice *bitmap*. 4. Indexação. I. Título.

CDD: 005.74 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"STB-index: Um Índice Baseado em Bitmap para Data Warehouse Espaço-Temporal"

Renata Miwa Tsuruda

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação

Membros da Banca:

Ricardo Rodrigues Ciferri

Prof. Dr. Ricardo Rodrigues Ciferri
(Orientador - DC/UFSCar)

Marilde Santos

Profa. Dra. Marilde Terezinha Prado Santos
(DC/UFSCar)

Ana Carolina Salgado

Profa. Dra. Ana Carolina Brandão Salgado
(UFPE)

São Carlos
Dezembro/2012

RESUMO

A crescente preocupação com o suporte ao processo de tomada de decisão estratégica fez com que as empresas buscassem tecnologias que apoiassem as suas decisões. A tecnologia mais utilizada atualmente é a de *Data Warehouse* (DW), que permite armazenar dados de forma que seja possível produzir informação útil e confiável para auxiliar na tomada de decisão estratégica. Aliando-se os conceitos de *Data Warehouse* Espacial (DWE), que permite o armazenamento e o gerenciamento de geometrias, e de *Data Warehouse* Temporal (DWT), que possibilita representar as mudanças nos dados que ocorrem no mundo real, surgiu o tema de pesquisa conhecido por *Data Warehouse* Espaço-Temporal (DWET), que é próprio para o tratamento de geometrias que se alteram ao longo do tempo. Essas tecnologias, aliadas ao constante crescimento no volume de dados armazenados, evidenciam a necessidade de estruturas de indexação que melhorem o desempenho do processamento de consultas analíticas com predicados espaciais e com variação das geometrias no tempo. Nesse sentido, este trabalho se concentrou na proposta de um índice para DWET denominado *Spatio-Temporal Bitmap Index*, ou STB-index. O índice proposto foi projetado para o processamento de consultas do tipo *drill-down* e *roll-up* considerando a existência de hierarquias espaciais predefinidas, sendo que os atributos espaciais podem variar sua posição e sua forma ao longo do tempo. A validação do STB-index ocorreu por meio da realização de testes experimentais utilizando um DWET criado a partir de dados sintéticos. Os testes avaliaram o tempo e o número de acessos a disco para a construção do índice, a quantidade de espaço para armazenamento do índice e o tempo e número de acessos a disco para o processamento de consultas analíticas. Os resultados obtidos foram comparados com o processamento de consultas utilizando os recursos disponíveis dos sistemas gerenciadores de banco de dados, sendo que o STB-index apresentou um ganho de desempenho entre 98,12% e 99,22% no tempo de resposta das consultas se comparado ao uso de visões materializadas.

Palavras-chave: Data warehouse, Data warehouse Espaço-Temporal, Índice Bitmap, Indexação.

ABSTRACT

The growing concern with the support of the decision-making process has made companies to search technologies that support their decisions. The technology most widely used presently is the Data Warehouse (DW), which allows storing data so it is possible to produce useful and reliable information to assist in strategic decisions. Combining the concepts of Spatial Data Warehouse (SDW), that allows geometry storage and managing, and Temporal Data Warehouse (TDW), which allows storing data changes that occur in the real-world, a research topic known as Spatio-Temporal Data Warehouse (STDW) has emerged. STDW are suitable for the treatment of geometries that change over time. These technologies, combined with the steady growth volume of data, show the necessity of index structures to improve the performance of analytical query processing with spatial predicates and also with geometries that may vary over time. In this sense, this work focused on proposing an index for STDW called Spatio-Temporal Bitmap Index, or STB-index. The proposed index was designed to processing drill-down and roll-up queries considering the existence of predefined spatial hierarchies and with spatial attributes that can vary its position and shape over time. The validation of STB-index was performed by conducting experimental tests using a DWET created from synthetic data. Tests evaluated the elapsed time and the number of disk accesses to construct the index, the amount of storage space of the index and the elapsed time and the number of disk accesses for query processing. Results were compared with query processing using database management system resources and STB-index improved the query performance by 98.12% up to 99.22% in response time compared to materialized views.

Keywords: Data Warehouse, Spatio-Temporal Data Warehouse, Bitmap Index, Indexing.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

a3DR-tree – *aggregate 3DR-tree*

aRB-tree – *aggregate RB-tree*

aR-tree – *aggregate R-tree*

BD – Banco de Dados

DW – *Data warehouse*

DWE – *Data Warehouse* Espacial

DWET – *Data Warehouse* Espaço-Temporal

DWT – *Data Warehouse* Temporal

IBJ – Índice Bitmap de Junção

ETL – *Extract, Transform and Load*

HSB-index – *Hierarchical Spatial Bitmap Index*

MAM – Método de Acesso Multidimensional

MBR – *Minimum Bounding Rectangle*

OLAP – *On-Line Analytical Processing*

OLTP – *On-Line Transactional Processing*

SB-index – *Spatial Bitmap Index*

SGBD – Sistema Gerenciador de Banco de Dados

SOLAP – *Spatial On-Line Analytical Processing*

SQL – *Structured Query Language*

SSB – *Star Schema Benchmark*

STB-index – *Spatial-Temporal Bitmap Index*

TIGER/Line – *Topologically Integrated Geographic Encoding and Referencing system*

SUMÁRIO

Capítulo 1

INTRODUÇÃO

Este capítulo apresenta o contexto em que este trabalho de pesquisa está inserido e a motivação que deu origem a ele. Em seguida são discutidos os objetivos, finalizando com a descrição da organização da dissertação.

1.1 Contexto do Trabalho

Na atual conjuntura de um mercado globalizado, é essencial para uma empresa reagir rapidamente às mudanças nas condições do negócio. Dessa forma, a descoberta de tendências que auxiliem no processo de tomada de decisão de uma empresa é de vital importância para a elaboração de estratégias de mercado para os seus produtos e serviços (CIFERRI, C. D. A., 2002).

Inicialmente, as informações estratégicas eram obtidas diretamente do ambiente operacional com o auxílio de programas de extração que selecionavam dados de arquivos ou de tabelas de bancos de dados e armazenavam-nos separadamente em arquivos conhecidos como extratos de dados. Esses extratos permitiam a manipulação dos dados sem afetar a integridade desses em relação às aplicações já existentes. Porém, a proliferação de extratos de dados fez com que as análises produzidas a partir de diferentes extratos perdessem credibilidade, uma vez que os dados existentes eram heterogêneos e inconsistentes (INMON, 2005).

Diante desses problemas, houve a necessidade de criação de uma base de dados integrada, o *Data Warehouse* (DW), separando o ambiente informacional do ambiente operacional (CIFERRI, C. D. A., 2002; KIMBALL; ROSS, 2002; INMON, 2005; RIZZI, 2007; MALINOWSKI; ZIMÁNYI, 2008a). O ambiente para o suporte aos

processos de gerência e tomada de decisão (i.e., o ambiente informacional) é fundamentalmente diferente do ambiente convencional de processamento de transações (i.e., o ambiente operacional). Operações no ambiente informacional são feitas por aplicações OLAP (*On-Line Analytical Processing*) e são caracterizadas por leituras sobre dados históricos, resumidos e consolidados, podendo acessar milhões de registros por vez e realizando muitas varreduras e junções. Por sua vez, OLTP (*On-Line Transaction Processing*) refere-se a um ambiente no qual um grande volume de operações, principalmente operações de inserção, remoção e atualização, é feito concorrentemente. Essas operações acessam poucos registros por vez e são simples, atômicas e isoladas.

O DW surgiu como solução para armazenar os dados de um ambiente informacional de uma empresa e prover suporte para aplicações OLAP. A informação obtida de cada fonte de dados deve passar por um processo de extração, limpeza, filtragem, tradução e integração (i.e., processo de ETL), para finalmente ser armazenada no DW, garantindo a consistência e homogeneidade dos dados. Assim, as consultas são realizadas de forma mais rápida e eficiente, pois são feitas diretamente no DW, sem que seja necessário acessar os provedores de informação originais.

Em muitos casos, a representação de dados espaciais pode ajudar a revelar tendências que seriam difíceis de descobrir somente com dados convencionais (e.g. números, *strings* e datas). Por exemplo, representar a localização (endereço) de clientes em um mapa pode auxiliar na descoberta de que muitos deles não necessariamente compram produtos em lojas próximas às suas residências. No entanto, apesar de muitos DW incluírem uma dimensão de localização com atributos como endereço, cidade e estado, usualmente essa informação é armazenada como um dado alfanumérico (*string*).

Com o objetivo de viabilizar o processamento de consultas analíticas com predicados espaciais, foram criados os *Data warehouses* Espaciais (DWE) (STEFANOVIC, 1997; BIMONTE et al., 2005; MALINOWSKI; ZIMÁNYI, 2005, 2006a, 2008a; SAMPAIO et al., 2006). Um DWE permite o armazenamento de mapas na forma de geometrias (i.e., conjunto de coordenadas), permitindo a realização de consultas do tipo: “*qual foi a quantidade vendida do produto P por mês do ano de 2011 nos estados cortados por rios?*”.

Por sua vez, a dimensão temporal é uma das mais relevantes, é sempre encontrada em DW e permite a comparação das medidas ao longo de diferentes períodos. Entretanto, alterações nos dados das demais dimensões não podem ser representadas usando a dimensão temporal. Uma vez que essas alterações podem influenciar as medidas (por exemplo, o acréscimo de ingredientes na composição de um biscoito pode elevar a quantidade vendida do produto), é essencial que haja uma tecnologia que permita lidar com essas mudanças. Para isso, extensões foram feitas aos modelos multidimensionais e deram origem aos *Data Warehouses* Temporais (DWT) (EDER et al., 2001, 2006; MALINOWSKI; ZIMÁNYI, 2006b, 2008a, 2008b; GOLFARELLI; RIZZI, 2009).

Assim como alterações nos dados convencionais das dimensões podem afetar as medidas, mudanças em dados espaciais de dimensões também podem impactar nas medidas numéricas ou espaciais. Por exemplo, a mudança de uma loja para um bairro mais nobre pode elevar a quantidade de produtos vendidos de melhor qualidade. Para tratar essas alterações, foram criados os *Data Warehouses* Espaço-Temporais (DWET) (PESTANA; SILVA, DA, 2005; GÓMEZ et al., 2009; VAISMAN; ZIMÁNYI, 2009; CASTRO, 2010). O DWET possibilita representar objetos espaciais que evoluem ao longo do tempo, ou seja, objetos cujas posições ou formatos se alteram com o tempo, viabilizando análises que levam em consideração aspectos espaciais e temporais simultaneamente.

1.2 Motivação

O modelo lógico relacional de um DW é geralmente um esquema estrela, o qual é composto por uma tabela de fatos central que referencia várias tabelas de dimensão (CIFERRI, C. D. A., 2002; KIMBALL; ROSS, 2002; INMON, 2005; RIZZI, 2007; MALINOWSKI; ZIMÁNYI, 2008a). No DWET, além das dimensões convencionais, há as dimensões espaço-temporais, que contêm geometrias que evoluem ao longo do tempo (PESTANA; SILVA, DA, 2005; GÓMEZ et al., 2009; VAISMAN; ZIMÁNYI, 2009; CASTRO, 2010).

O processamento de consultas sobre esse esquema é extremamente custoso, uma vez que envolve operações de junção e agrupamento sobre uma

enorme quantidade de dados, volume esse que tem aumentado substancialmente nos últimos anos. Somado a esse custo, há ainda o cômputo dos predicados espaciais e temporais. Dessa forma, métodos que melhorem o desempenho dessas consultas, tais como índices, são fundamentais.

Na literatura existem diversas propostas de estruturas de indexação. Considerando apenas bancos de dados espaciais, pode-se citar mais de cinquenta propostas, desde propostas originais até propostas que estendem estruturas previamente existentes (GAEDE; GÜNTHER, 1998; CIFERRI, R. R., 2002). Algumas estruturas de indexação são mais conhecidas pelos usuários e mais utilizadas em Sistemas Gerenciadores de Banco de Dados (SGBD), como exemplo pode-se citar B-tree, R-tree, quadtree e slim-tree (BECKMANN et al., 1990; TRAINA JR. et al., 2000).

Apesar da grande quantidade de estruturas de indexação existentes na literatura, apenas algumas foram projetadas e estão aptas para indexar dados armazenados em um DW. Essa quantidade é ainda mais reduzida quando se tratam de DWE, DWT e DWET, motivando o estudo de estruturas de indexação que obtenham melhores tempos de resposta no processamento de consultas OLAP.

1.3 Objetivo

Neste trabalho de pesquisa, é proposto o *Spatio-Temporal Bitmap Index*, ou STB-index, para a indexação de DWET. O STB-index é uma extensão do SB-index, índice para DWE proposto por Siqueira (2009). O índice proposto nesta dissertação foi projetado para o processamento de consultas do tipo *drill-down* e *roll-up* considerando a existência de hierarquias espaciais predefinidas, sendo que os atributos espaciais podem variar a sua posição e a sua forma ao longo do tempo.

A validação do STB-index foi realizada por meio de testes de desempenho experimentais utilizando um DWET criado a partir de dados sintéticos. Os testes avaliaram o tempo e o número de acessos a disco para a construção do índice, a quantidade de espaço para o armazenamento do índice e o tempo e o número de acessos a disco para o processamento de consultas OLAP. Os resultados obtidos foram comparados com o processamento de consultas utilizando os recursos

disponíveis dos SGBD, sendo que o STB-index apresentou um ganho de desempenho no processamento de consultas OLAP entre 98,12% e 99,22% no tempo de resposta se comparado ao uso de visões materializadas.

1.4 Organização da Dissertação

Esta dissertação está organizada da seguinte forma:

- Capítulo 2: são descritos os conceitos e fundamentos teóricos essenciais para a compreensão deste trabalho de pesquisa.
- Capítulo 3: é dedicado a um estudo específico de índices com o objetivo de exibir uma discussão sobre o estado da arte.
- Capítulo 4: é proposto o STB-index.
- Capítulo 5: são descritos os experimentos realizados para testar e comparar o desempenho do STB-index com recursos dos SGBD.
- Capítulo 6: são apresentadas as contribuições do trabalho e as sugestões para trabalhos futuros.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

Este capítulo descreve os principais conceitos relacionados à proposta da dissertação de mestrado. Primeiramente, são definidos os conceitos relacionados a DW, DWE, DWT e DWET, essenciais para se entender os dados e a estrutura do DWET que será indexado na validação da proposta do STB-index. Em seguida, o conceito de indexação é apresentado, uma vez que este trabalho de pesquisa é voltado à proposição de um índice para DWET. Por fim, considerando-se que a validação da proposta será por meio do uso de benchmarks de banco de dados, faz-se necessário a explicação dessa técnica experimental de avaliação de desempenho.

2.1 Data Warehouse

Aplicações OLAP são projetadas para responder consultas analíticas multidimensionais sobre dados armazenados em um DW. Em geral, essas consultas são complexas e demoradas, pois requerem o acesso a um grande volume de dados e a junção de várias tabelas. Dessa forma, a modelagem lógica de um DW é feita de maneira a facilitar o processamento dessas consultas. Na modelagem, o DW pode ser visto como um cubo de dados multidimensional cujas células representam fatos, ou seja, eventos que ocorreram no domínio do negócio (POURABBAS; RAFANELLI, 1999; CIFERRI, C. D. A., 2002; RIZZI, 2007; MALINOWSKI; ZIMÁNYI, 2008a). Os fatos são quantificados por medidas e descritos por dimensões. As dimensões, por sua vez, representam perspectivas do DW e definem o contexto sobre o qual os dados serão analisados enquanto as medidas são definidas pela intersecção das dimensões.

Um exemplo de cubo no domínio de uma aplicação de varejo é mostrado na Figura 2.1. O fato representado é a venda, a qual é descrita pela medida quantidade vendida e analisada de acordo com as dimensões `Data`, `Item` e `Localização`.

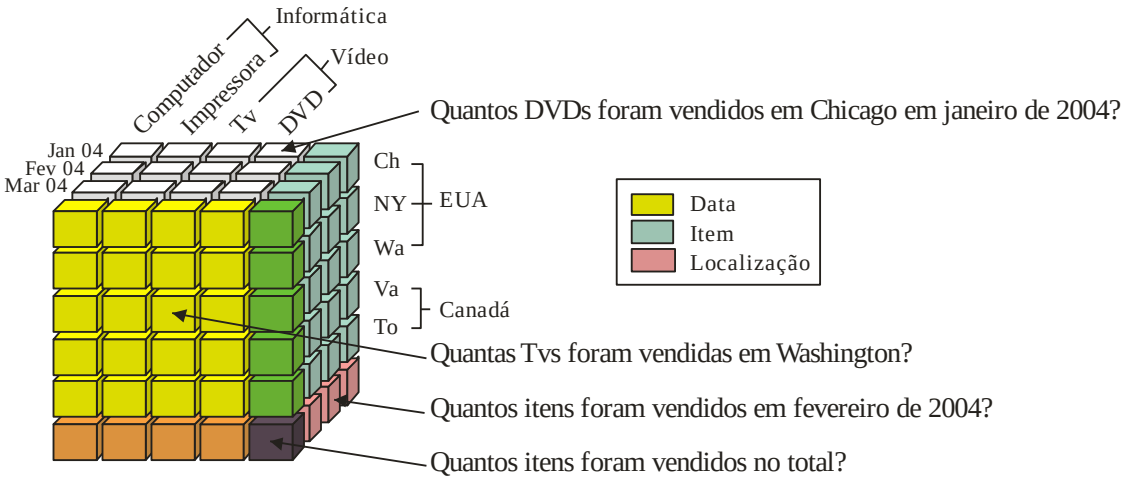


Figura 2.1. Exemplo de cubo de dados multidimensional de uma aplicação de varejo e exemplo de consultas analíticas. Adaptada de Siqueira (2009).

No exemplo em questão, pode-se observar a existência da hierarquia de atributos país $\preceq$ cidade, sendo cidade (`Ch`, `NY`, `Wa`, `Va`, `To`) um nível de granularidade menor que país (`EUA`, `Canadá`). Essas hierarquias de atributos permitem a realização de agregações para responder consultas de forma mais detalhada ou mais resumida. Existem duas operações para a realização desse tipo de consulta: *drill-down* e *roll-up*. A operação de *drill-down* consiste em obter resultados progressivamente mais detalhados. Já a operação de *roll-up* objetiva obter resultados progressivamente mais resumidos. A Figura 2.2 apresenta exemplos de operações de *drill-down* e *roll-up* com base no cubo da Figura 2.1. Outras operações bastante conhecidas são *pivoting*, *slice-and-dice*, *drill-across* e *drill-through* (POURABBAS; RAFANELLI, 1999; CIFERRI, C. D. A., 2002; MALINOWSKI; ZIMÁNYI, 2008a).

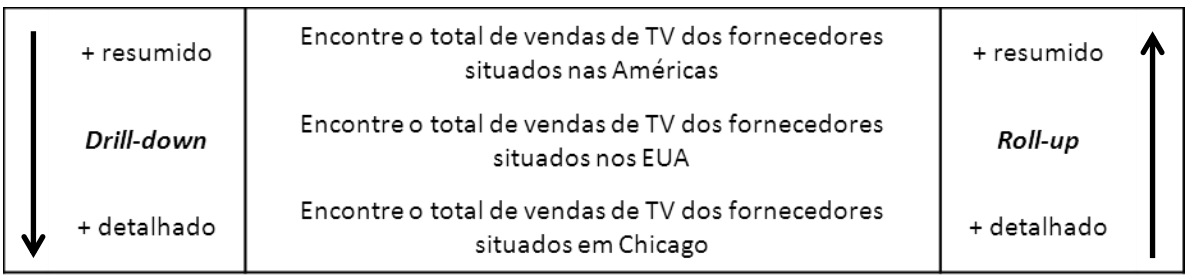


Figura 2.2. Exemplos de operações de *drill-down* e *roll-up* (SIQUEIRA, 2009).

O modelo lógico relacional de um DW é geralmente um esquema estrela, o qual é composto por uma tabela central de fatos e várias tabelas de dimensão (CIFERRI, C. D. A., 2002; KIMBALL; ROSS, 2002; MALINOWSKI; ZIMÁNYI, 2008a). Toda tupla na tabela de fatos contém uma chave estrangeira para cada tabela de dimensão e valores de medidas do objeto de análise. Na Figura 2.3, é apresentado um exemplo de esquema estrela para a aplicação de varejo da Figura 2.1. No centro tem-se a tabela de fatos de `Vendas` com chaves estrangeiras para as dimensões `Item`, `Data` e `Localização` e a medida de interesse `Quantidade` para análise do negócio.

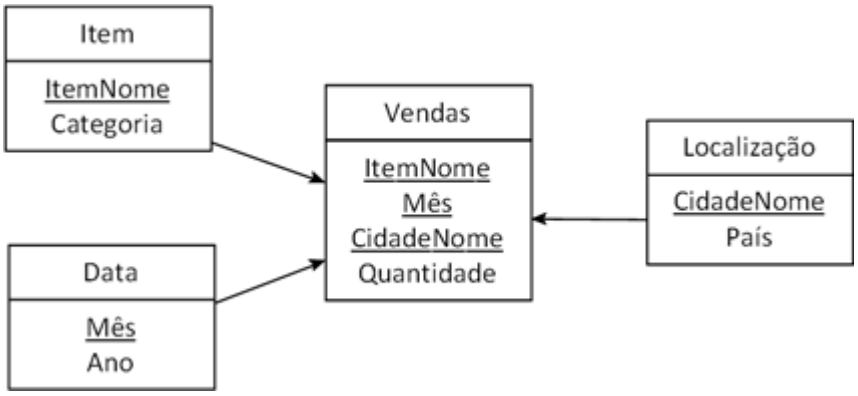


Figura 2.3. Exemplo de esquema estrela para a aplicação de varejo.

Uma forma alternativa para modelar um DW é utilizando o esquema floco de neve (CIFERRI, C. D. A., 2002; KIMBALL; ROSS, 2002; MALINOWSKI; ZIMÁNYI, 2008a). Ele permite representar explicitamente uma hierarquia de atributos por meio da normalização das tabelas de dimensão. Porém, o esquema floco de neve aumenta o custo do processamento de consultas OLAP ao introduzir novas junções entre as tabelas. Uma vez que o desempenho no processamento de consultas é prioridade quando se trata de um DW, o esquema estrela é preferível ao esquema floco de neve.

2.1.1 Visões Materializadas

Uma visão convencional, ou apenas visão, é uma especificação de programa que gera dados cada vez que é referenciada. Quando se trabalha com DW, torna-se interessante armazenar os dados obtidos pela execução da visão. A isso se denomina visão materializada (CIFERRI, C. D. A., 2002; MALINOWSKI; ZIMÁNYI, 2008a).

Visões materializadas são essenciais quando se deseja um rápido acesso a uma parte dos dados de um DW, ou quando recalcular a visão sobre as relações base é muito caro ou inviável, seja pelo tamanho das relações base, pela localização das mesmas ou pela complexidade da consulta. Dessa forma, visões são muito utilizadas para melhorar o desempenho no processamento de consultas OLAP.

Ao mesmo tempo em que melhora o desempenho, um maior número de visões materializadas pode comprometer a escalabilidade do mesmo. Nesse sentido, pode haver certas restrições, tais como espaço de armazenamento disponível e os custos relacionados à atualização dessas visões. Por isso, um bom projeto de DW deve levar em consideração estes três fatores: desempenho, escalabilidade e tempo de atualização (HARINARAYAN et al., 1996).

2.2 Data Warehouse Espacial

Um *data warehouse* espacial é um DW que armazena dados espaciais (i.e. dados descritos por uma geometria) em uma ou mais dimensões ou em pelo menos uma medida espacial (STEFANOVIC, 1997; BIMONTE et al., 2005; MALINOWSKI; ZIMÁNYI, 2005, 2006a, 2008a; SAMPAIO et al., 2006).

Dados espaciais podem ser de diferentes tipos (Figura 2.4). Um ponto é a menor unidade para representar um objeto espacial que não possui extensão. Em geral, pontos são usados para representar localizações discretas, como uma cidade em um mapa. Uma linha é uma sequência de pontos conectados de forma retilínea, enquanto que em uma linha poligonal, essa última característica mencionada não se observa. Em ambas, cada par de pontos conectados corresponde a um segmento de linha. Linhas e linhas poligonais são utilizadas para representar objetos espaciais lineares, como rios, estradas, ferrovias e redes de infraestrutura. Um polígono é formado por uma sequência fechada de linhas ou linhas poligonais, isto é, o primeiro e o último ponto coincidem. Polígonos são usados para representar objetos espaciais bidimensionais, como a área ocupada por um bairro. Um poliedro é formado por quatro ou mais polígonos, sendo utilizado para representar sólidos no espaço, como um membro humano ou uma peça mecânica (GÜTING, 1994; CÂMARA et al., 1996; RIGAUX et al., 2001; CIFERRI, R. R., 2002).

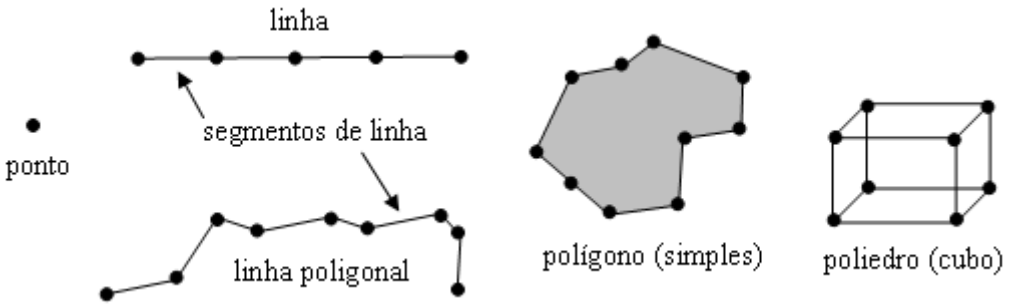


Figura 2.4. Tipo de dados espaciais. Adaptada de Ciferri, R. R. (2002).

Para a modelagem de um DWE utiliza-se o esquema estrela com algumas extensões: as tabelas de dimensão e as medidas podem ser espaciais ou não-espaciais (i.e. convencionais). Uma tabela de dimensão espacial não-geométrica contém dados descritivos sobre a localização do objeto espacial, mas não possui qualquer geometria que o represente (Figura 2.5.a). Por sua vez, em uma tabela de dimensão estritamente espacial, os objetos são representados por feições geométricas (Figura 2.5.b).

Figura 2.5. Exemplo de dados (a) não-geométricos e (b) geométricos.

A Figura 2.6 apresenta um exemplo de esquema estrela para um DWE definido a partir de uma adaptação do esquema do SSB (*Star Schema Benchmark*) (O'NEIL, P. E. et al., 2009). Ele é formado pela tabela de fatos `Lineorder` e pelas tabelas de dimensão `Customer`, `Supplier`, `Part` e `Date`. Nas tabelas de `Customer` e `Supplier`, todos os atributos terminados com a expressão "_geo" armazenam geometrias relacionadas à localização: `address_geo` armazena pontos e os demais atributos espaciais, `city_geo`, `nation_geo` e `region_geo` armazenam polígonos.

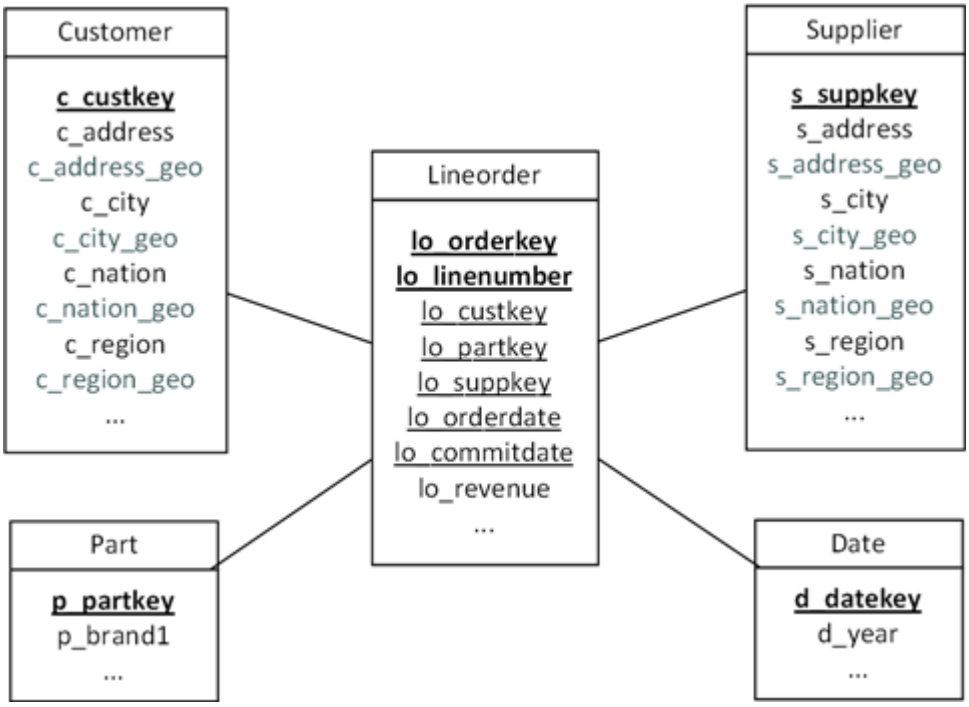


Figura 2.6. Exemplo de esquema estrela de um DWE. Adaptada de Siqueira et al. (2008).

É possível realizar diversos tipos de consultas sobre objetos espaciais dispostos no espaço multidimensional. Um exemplo de consulta em SQL é mostrado na Figura 2.7 considerando-se o esquema de DWE da Figura 2.6, em que o predicado convencional `s_region = 'EUROPE'` é substituído pelo predicado espacial está contido (*within*) ou intersecta (*intersects*) uma determinada janela de consulta (JC).

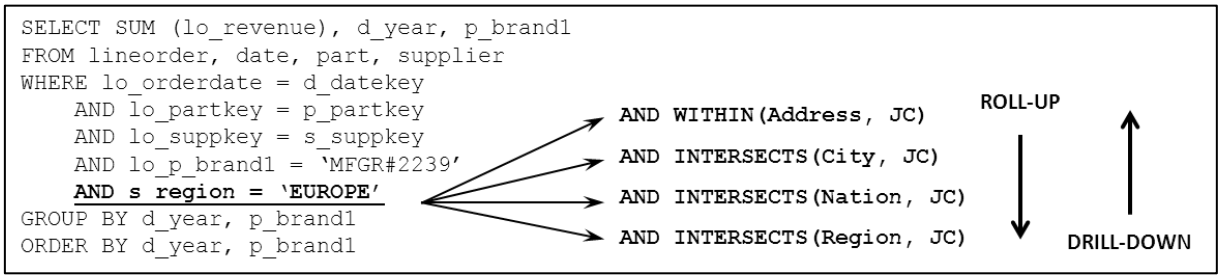
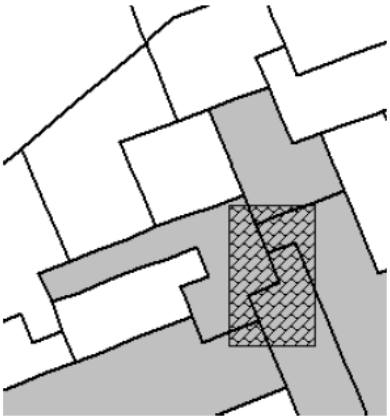
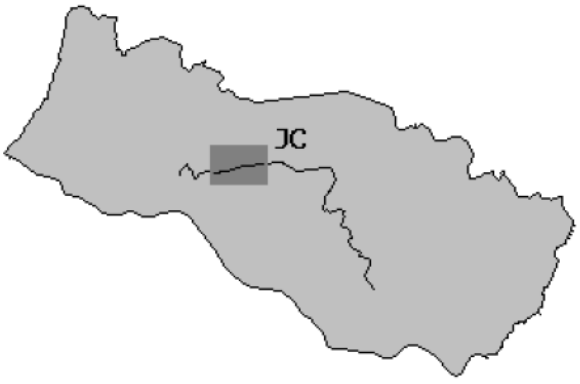


Figura 2.7. Exemplo de consulta contendo um predicado espacial. Adaptada de Siqueira et al. (2009).

A Tabela 2.1 apresenta os principais tipos de consultas envolvendo predicados espaciais. Para mais detalhes sobre os tipos e subtipos de consultas, consultar Ciferri, R. R. (2002) e Siqueira (2009).

Tabela 2.1. Exemplo de consultas espaciais (SIQUEIRA, 2009).

Consulta / Exemplo	Figura
Intersection range query Fornecida uma região retangular com a finalidade de construir um parque, identifique todos os bairros que terão terras desapropriadas. → A região retangular entrelaçada (janela de consulta) é onde se pretende construir o parque. Os polígonos são os bairros, sendo que os polígonos na cor cinza intersectam a janela de consulta e fazem parte da resposta. Os polígonos na cor branca não intersectam a janela de consulta, portanto não são parte da resposta.	
Containment range query Fornecida uma região retangular sobre uma cidade, recupere as instituições de ensino nela contidas, bem como os bairros. → A região retangular cinza tem coloridas em preto as instituições nela contidas. O único bairro contido por ela está destacado com listras. As instituições que não fazem parte da resposta têm cor cinza, e os bairros não recebem cor de destaque (exceto onde a janela de consulta os sobrepõe).	
Enclosure range query Fornecida uma região retangular que intersecta um rio, encontre a bacia hidrográfica a que ele pertence. → A janela de consulta (JC) retangular (cinza escuro) intersecta um rio (linha), e por isso toda a bacia hidrográfica é colorida em um tom de cinza mais claro.	

2.3 Data Warehouse Temporal

Tempo é uma das dimensões usualmente encontradas em DW e permite a comparação das medidas em diferentes períodos. No entanto, a dimensão Tempo não possibilita representar alterações nas demais dimensões. Isso ocorre porque, na modelagem convencional de um DW, considera-se que os dados das dimensões são invariantes ao tempo. Os *Data Warehouses* Temporais (DWT) surgiram com a finalidade de permitir a representação dessas mudanças nas dimensões (EDER et al., 2001, 2006; MALINOWSKI; ZIMÁNYI, 2006b, 2008a, 2008b; GOLFARELLI; RIZZI, 2009).

No mundo real, mudanças nas regras de negócio de uma empresa são inevitáveis para sua sobrevivência e isso pode ter um grande impacto nas medidas utilizadas para tomadas de decisão. Kimball e Ross (2002) propuseram três soluções para representar dimensões que mudam lentamente, ou seja, dimensões cujos atributos não se alteram com muita frequência. As soluções propostas foram chamadas de Tipo 1, Tipo 2 e Tipo 3.

No Tipo 1, o valor antigo do atributo é substituído pelo valor corrente na linha da dimensão. Desse modo, o atributo sempre conterá o valor mais recente e o histórico de mudanças não é armazenado. Portanto, o Tipo 1 não atende à contento a questão temporal.

Como exemplo, suponha um fragmento da tabela de dimensão `Produto` apresentado na Tabela 2.2. Uma chave artificial (*surrogate key*) é utilizada como chave primária da tabela (`Chave do Produto`) ao invés da chave natural (`SKU`). Ainda que o `SKU` seja tratado como um atributo descritivo qualquer, ele continua sendo chave natural, ou seja, ao contrário dos outros atributos descritivos, o `SKU` deve permanecer inalterado.

Tabela 2.2. Fragmento da tabela de dimensão Produto.

Chave do Produto	SKU (Chave Natural)	Nome	Peso
12345	ABC123-Z	Caixa de Bombom	500g

Suponha que o `Peso` do produto `Caixa de Bombom` seja alterado para `400g`. Nesse caso, o resultado da alteração utilizando o Tipo 1 é mostrado na Tabela 2.3. O valor `500g` é sobrescrito pelo novo valor, `400g`.

Tabela 2.3. Fragmento da tabela de dimensão Produto após alteração usando Tipo 1.

Chave do Produto	SKU (Chave Natural)	Nome	Peso
12345	ABC123-Z	Caixa de Bombom	400g

Apesar de ser bastante fácil e rápida de ser executada, a solução Tipo 1 pode induzir a erros durante a análise, pois o histórico de alterações não é mantido. Por exemplo, a mudança no peso da `Caixa de Bombom` pode reduzir a quantidade vendida do mesmo. Se essa informação não é armazenada, não haverá explicação para a redução das vendas ou ela será erroneamente atribuída a outro fator. Dessa forma, recomenda-se o uso da solução Tipo 1 apenas quando a alteração é referente a uma correção do valor, ou seja, o valor antigo estava incorreto, ou se ainda não há nenhum valor para o atributo.

O Tipo 2 é a solução predominante para tratar dimensões que mudam lentamente. Cada vez que há alteração no valor de algum atributo, uma nova linha é adicionada à tabela de dimensão.

A Tabela 2.4 mostra o resultado da alteração do peso da `Caixa de Bombom` utilizando o Tipo 2. Uma nova chave artificial é gerada e a chave natural permanece a mesma. Dessa forma, é possível armazenar o histórico de quantas alterações ocorreram e, caso seja necessário efetuar alguma consulta que envolva a quantidade de produtos distintos, pode-se utilizar a chave natural.

Tabela 2.4. Fragmento da tabela de dimensão Produto após alteração usando Tipo 2.

Chave do Produto	SKU (Chave Natural)	Nome	Peso
12345	ABC123-Z	Caixa de Bombom	500g
34567	ABC123-Z	Caixa de Bombom	400g

Para análises mais avançadas, pode ser necessário armazenar o tempo de validade de cada produto, sendo possível saber precisamente quando houve a alteração. Kimball e Ross (2002) recomendam cuidado para que não haja confusão entre o tempo de validade e o tempo armazenado na tabela de dimensão Tempo.

No Tipo 3, uma nova coluna é adicionada para armazenar a alteração. A Tabela 2.5 mostra o resultado da alteração do peso da `Caixa de Bombom`. Uma nova coluna (`Peso Anterior`) é adicionada para armazenar o valor antigo e a coluna `Peso` é atualizada com o valor atual.

Tabela 2.5. Fragmento da Tabela de Dimensão Produto. Adaptada de Kimball e Ross (2002).

Chave do Produto	SKU (Chave Natural)	Nome	Peso	Peso Anterior
12345	ABC123-Z	Caixa de Bombom	400g	500g

Essa solução se distingue do Tipo 2 porque tanto o valor antigo quanto o atual podem ser considerados verdadeiros ao mesmo tempo. Ela é usada com pouca frequência e sua principal desvantagem é quando há a necessidade de controlar muitas alterações imprevistas.

Há ainda abordagens híbridas, que mesclam as soluções Tipo 1, Tipo 2 e Tipo 3. Para mais detalhes, consultar Kimball e Ross (2002).

Dois tipos de tempo são normalmente armazenados nos DWT, tempo válido e tempo de transação (DYRESON et al., 1994; GOLFARELLI; RIZZI, 2009). Tempo válido expressa o tempo em que o fato é verdadeiro no domínio do negócio e normalmente é fornecido pelo usuário. Já o tempo de transação expressa o tempo em que o fato foi registrado no banco de dados e pode ser recuperado em uma consulta, sendo fornecido pelo SGBD.

2.4 Data Warehouse Espaço-Temporal

Data Warehouses Espaço-Temporais (DWET) possibilitam trabalhar com conceitos espaciais e temporais simultaneamente. Nesse sentido, há duas abordagens distintas relacionadas a DWET na literatura. Umas delas está relacionada a objetos móveis, em que o objetivo é obter dados resumidos sobre a trajetória de um conjunto de objetos para auxiliar na tomada de decisão (GÓMEZ et al., 2009; TAO; PAPADIAS, 2009). Como exemplo pode-se citar um sistema de controle de tráfego que monitora a quantidade de carros em determinadas áreas de

interesse para evitar congestionamentos. A outra abordagem trata de objetos espaciais que se alteram aleatoriamente ao longo do tempo e está relacionada com o conceito de dimensões que mudam lentamente, introduzido por Kimball e Ross (2002). Como exemplo tem-se a representação da divisão de um estado, a alteração do endereço de uma loja, etc. Esta dissertação se concentra apenas nessa segunda abordagem.

Segundo Castro (2010), DWET contêm pelo menos uma dimensão espaço-temporal, ou seja, uma dimensão com um ou mais atributos espaciais que variam ao longo do tempo, sendo que esses atributos podem formar uma hierarquia espacial também variável ao longo do tempo.

Em seu trabalho, Castro (2010) propõe um modelo conceitual para DWET denominado *Conceptual SpatioTemporal Modeling* (CSTM). O CSTM é baseado no modelo MultiDimER (MALINOWSKI; ZIMÁNYI, 2008a) e permite definir níveis, hierarquias e dimensões tanto com características espaciais quanto temporais ou com ambas simultaneamente (i.e. espaço-temporais).

2.5 Estruturas de Indexação

Índices são estruturas de acesso auxiliares utilizados para tornar mais rápida a recuperação de registros na resposta a certas condições de busca. Eles permitem acessar os registros por caminhos alternativos, sem que seja necessário afetar a disposição física dos dados, e possibilitam selecionar e recuperar os registros que satisfazem às condições de consulta segundo uma chave de busca, a qual consiste num conjunto arbitrário de atributos. Além disso, um índice possibilita encontrar um registro consultando apenas uma pequena parte de todos os registros possíveis pela imposição de uma ordem de acesso, sem de fato reordenar os registros (FOLK; ZOELLICK, 1991).

Um índice pode ser criado com quaisquer atributos de uma relação. Quando o arquivo de dados está fisicamente ordenado na mesma ordem que as chaves de busca no índice, tem-se um índice primário; caso contrário, tem-se um índice secundário (MANOLOPOULOS et al., 2009). Quando não existem registros duplicados para os valores da chave de um índice, tem-se um índice único.

Comumente, as estruturas de indexação crescem em volume, exigindo que seu armazenamento seja feito em memória secundária, o que é feito, geralmente, utilizando-se discos magnéticos. Os custos no armazenamento secundário estão relacionados à busca de uma determinada localização em disco. Uma vez posicionada, a cabeça de leitura pode processar um fluxo contíguo de bytes rapidamente. É uma combinação de busca lenta e transferência de dados rápida.

Em geral, os registros de um índice são menores que aqueles do arquivo de dados porque possuem menos atributos (são normalmente compostos por um valor de chave, um ponteiro e nenhum ou poucos atributos relevantes para a pesquisa), mas podem também ocupar diversos blocos. Sendo assim, mesmo com estratégias de busca eficientes, podem ser necessários vários acessos a disco para obter o registro desejado. A modificação dos dados por meio de inserções, eliminações e atualizações implica, em certos casos, na custosa reorganização do índice e no gerenciamento do espaço extra concedido ao armazenamento (blocos de *overflow*). Em geral, índices são baseados em esquemas multi-níveis, árvores e *hashing* (GARCIA-MOLINA et al., 2008).

As estruturas de indexação convencionais, como B-tree e hash (FOLK; ZOELLICK, 1991; GARCIA-MOLINA et al., 2008), auxiliam na recuperação de registros de acordo com os valores de uma única chave de busca. Índices dessa forma possuem apenas uma dimensão, seja a chave de busca um único atributo ou uma concatenação deles. Aplicações OLAP e espaciais exigem a visualização de dados em espaços com duas ou mais dimensões, se diferenciando dos SGBD tradicionais no suporte a certos tipos de consulta.

O espaço multidimensional é o domínio utilizado pelos métodos de acesso multidimensionais (MAM) (GAEDE; GÜNTHER, 1998) para indexar elementos com dimensionalidade. O armazenamento de todos os pares de coordenadas de um objeto espacial é custoso, por isso existem aproximações (i.e., abstrações) dos objetos espaciais para representá-los por formas geométricas mais simples. No entanto, uma aproximação deve ser conservativa, ou seja, cada ponto do objeto espacial deve estar contido na geometria da aproximação. Isso garante que nenhum dos objetos espaciais que satisfaz certo relacionamento espacial seja desconsiderado na resposta a uma consulta.

A aproximação mais utilizada pelos MAM é o retângulo envolvente mínimo (MBR – *Minimum Bounding Rectangle*), que consiste no menor retângulo

d-dimensional com lados paralelos aos eixos que contém completamente o objeto espacial. Essa aproximação consiste em quatro pares de coordenadas: ($X_{mín}$, $Y_{mín}$), ($X_{mín}$, $Y_{máx}$), ($X_{máx}$, $Y_{mín}$) e ($X_{máx}$, $Y_{máx}$), nas quais $X_{mín}$ e $Y_{mín}$ são os menores valores das coordenadas X e Y do retângulo, respectivamente, e $X_{máx}$ e $Y_{máx}$ são os maiores valores.

A utilização de aproximações, tais como o MBR, implica na perda de precisão na representação do objeto espacial, ocasionando a formação de uma área vazia, ou seja, um espaço contido na aproximação que não faz parte da geometria do objeto espacial. Essa área vazia é denominada *dead space*. Se essa área for considerada no teste de um relacionamento espacial, o objeto espacial correspondente pode ser um falso candidato, porque pode não satisfazer de fato ao referido relacionamento espacial.

Devido a essa imprecisão que pode ocorrer no conjunto de respostas, o processamento de consultas espaciais nos MAM exige a filtragem e o posterior refinamento das respostas (Figura 2.8). Na primeira, objetos que não satisfazem a um relacionamento espacial são descartados e, devido ao uso de aproximações, surgem falsos candidatos. Essa fase é pouco custosa graças à manipulação de geometrias simples, o que diminui o custo de transferência de dados para a memória principal e o custo para se determinar a satisfação de um relacionamento espacial. No refinamento, cada objeto candidato passa pela verificação, que testa se o mesmo obedece ao relacionamento espacial, descartando assim os falsos candidatos. Como trata das geometrias exatas dos objetos espaciais, essa fase é mais custosa, pois embora o volume de dados seja menor, os cálculos são mais complexos e requerem a recuperação de dados armazenados em disco (BRINKHOFF et al., 1994).

Portanto, a fase de filtragem deve reduzir drasticamente a quantidade de objetos a serem analisados na fase de refinamento para que se obtenham ganhos expressivos no desempenho. Aproximações progressivas, que consistem em subconjuntos de pontos do objeto espacial, podem ser usadas para validar candidatos numa fase intermediária entre a filtragem e o refinamento, dispensando que alguns candidatos verdadeiros sigam para o refinamento.

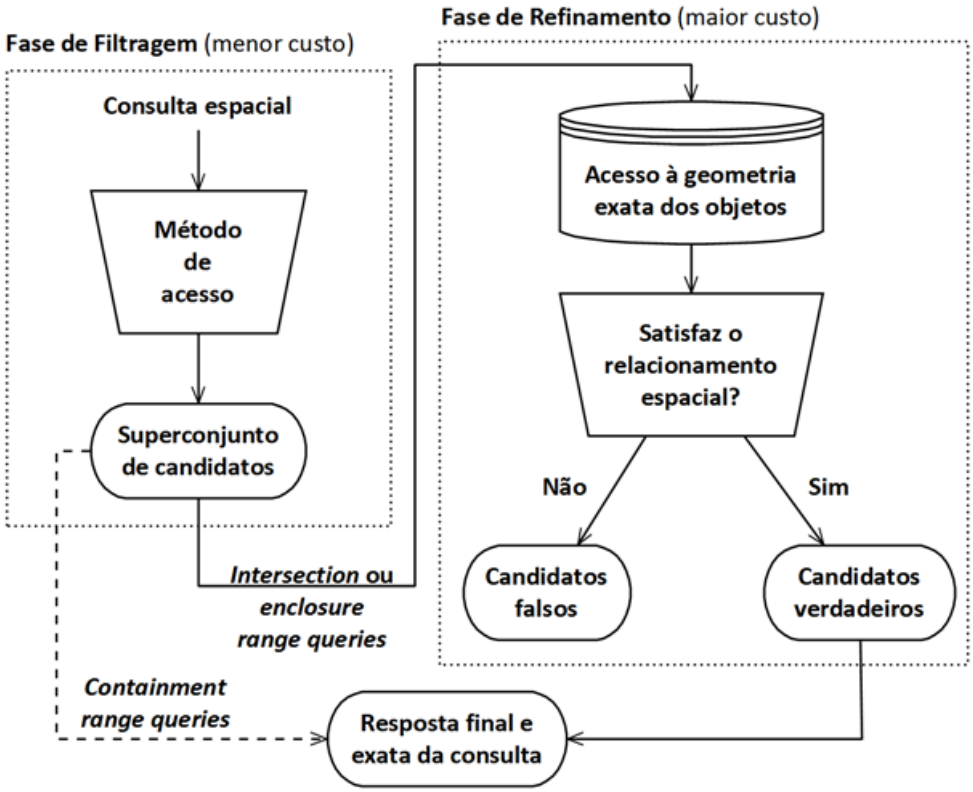


Figura 2.8. Etapas do processamento de consultas espaciais. Adaptada de Ciferri, R. R. (2002).

2.5.1 R-tree

A R-tree (GUTTMAN, 1984) é um método de acesso multidimensional baseado na B-tree que provê suporte eficiente para *range queries* por meio da recuperação de objetos de acordo com suas localizações espaciais. Ela foi proposta devido à limitação dos índices existentes até então com relação à dinâmica dos dados, ao tratamento de paginação em memória secundária, à indexação em dimensões iguais ou superiores a dois e ao desempenho na manipulação de grandes quantidades de dados.

A estrutura da R-tree é baseada em nó folha e nó interno. Cada nó possui espaço para M entradas, sendo que cada entrada é formada por uma localização espacial e a localização dos dados espaciais na memória, relativos à localização espacial representada. Além disso, cada nó possui um número mínimo m de entradas com $m \leq M/2$, com exceção do nó raiz que, quando nó interno, deve ter pelo menos duas entradas e, quando nó folha, não possui restrições. Uma vez que a

R-tree é usada comumente para indexar objetos espaciais armazenados em memória secundária, cada nó corresponde a uma página de disco.

Apesar da estrutura do nó folha e do nó interno ser idêntica, há uma diferença entre eles: enquanto os nós internos armazenam informações sobre nós de nível imediatamente inferior, também chamados de nós filhos, nós folhas armazenam exclusivamente informações sobre objetos espaciais. Por ser uma estrutura balanceada, todos os nós folhas estão sempre no mesmo nível.

Um nó folha contém entradas da forma (`I`, `id`), em que `I` corresponde ao MBR n-dimensional do objeto espacial identificado por `id` e esse, por sua vez, é o identificador do objeto espacial, o qual indiretamente corresponde a um endereço de memória que possui os dados do objeto.

Um nó interno possui entradas da forma (`I`, `p`), em que `I` corresponde ao MBR n-dimensional do nó filho referenciado pelo ponteiro `p`, o qual engloba o MBR de todas as entradas do nó inferior e, consequentemente, engloba o MBR de todos os objetos espaciais contidos em nós folhas que podem ser alcançados a partir dessa entrada, e `p` é o endereço de um nó filho, o qual determina uma sub-árvore hierarquicamente subordinada a essa entrada.

A Figura 2.9 e a Figura 2.10 ilustram um exemplo de R-tree que indexa 18 objetos espaciais, que podem ser pontos, linhas e polígonos, sendo visível, na primeira figura, a geometria de cada objeto no espaço e, na segunda, a estrutura de dados da R-tree, sendo essa composta de três níveis com o valor de `M=4` e `m=50%*M`.

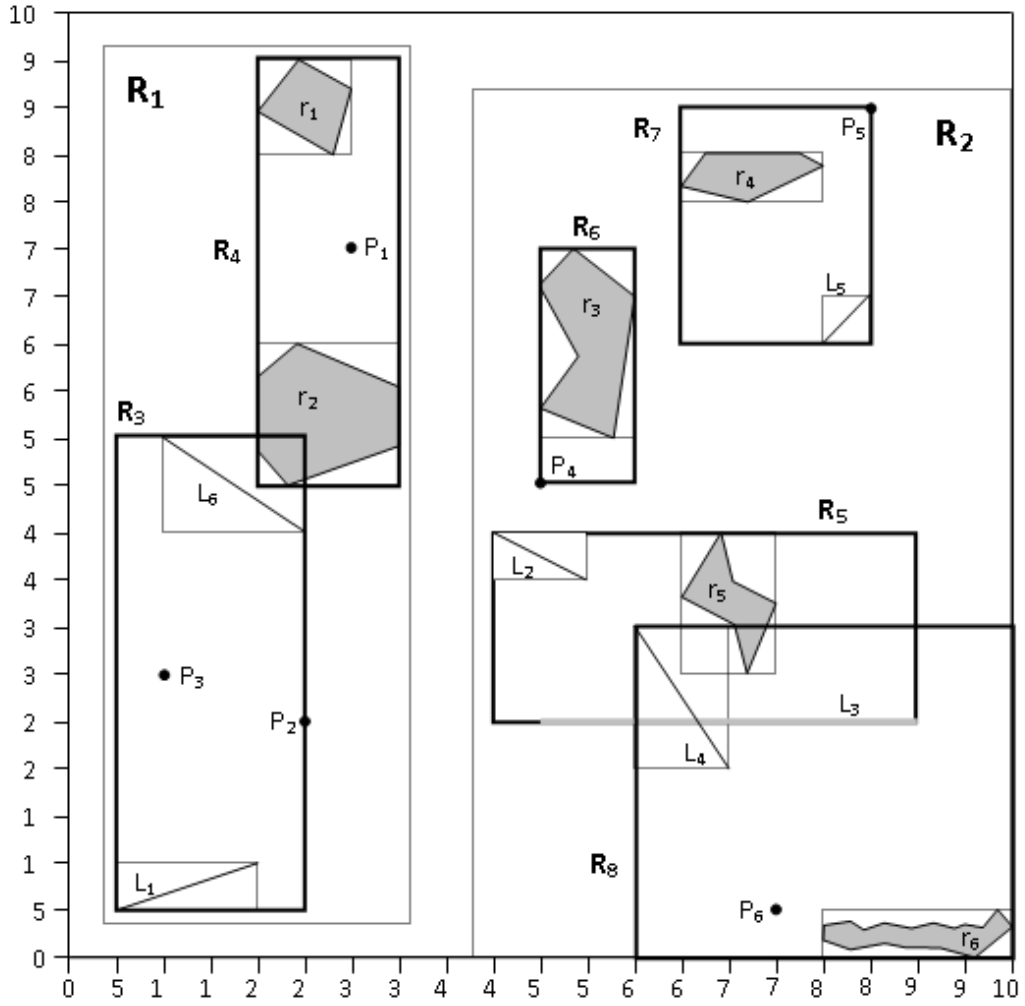


Figura 2.9. Exemplo: extensão e localização de objetos no espaço (CIFERRI, R. R., 2002).

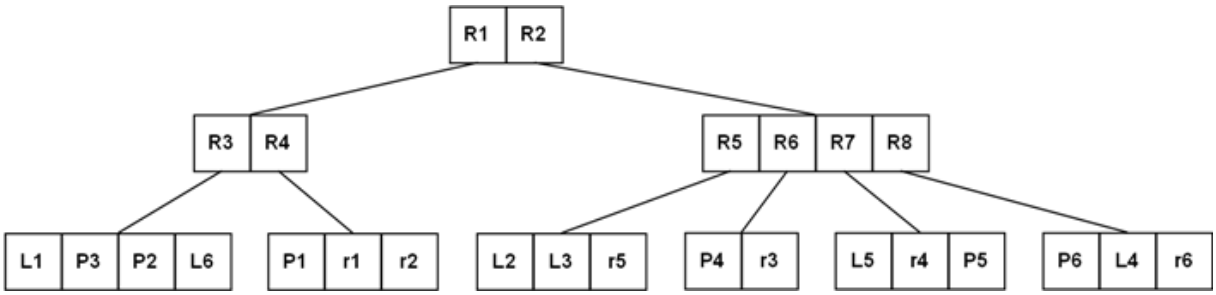


Figura 2.10. Estrutura de dados da R-tree exemplificada na Figura 2.9. Adaptada de Ciferri, R. R. (2002).

2.6 Técnica de Benchmark de Banco de Dados

Um benchmark é o resultado da execução de um conjunto de testes padrão em um componente ou sistema com a finalidade de comparar seu desempenho com outros componentes ou sistemas (BARBOSA et al., 2009).

Ele pode ser projetado para representar sistemas do mundo real para obtenção de medidas de desempenho dos mesmos ou pode ser um benchmark sintético, permitindo a configuração do conjunto de dados e das operações na execução para testar componentes individuais, tais como estruturas de indexação.

Benchmarks são compostos de especificações da carga de trabalho, dos dados e das medidas de desempenho (CIFERRI, R. R., 2002). Na área de banco de dados, a carga de trabalho especifica o conjunto de consultas que simulam as atividades ou transações sobre os BD. O conjunto de dados pode ser composto de dados de aplicações do mundo real (dados reais) ou ser composto de dados sintéticos, produzidos a partir de geradores de dados. A especificação das medidas de desempenho definem como essas devem ser coletadas e de que forma serão avaliadas.

O *Transaction Processing Performance Council* (TPC) é uma organização reconhecida na definição e publicação de benchmarks para sistemas de bancos de dados (BARBOSA et al., 2009). Na área de aplicações de suporte à tomada de decisão, existe o TPC-H, um benchmark para DW relacionado a pedidos e vendas de uma companhia. Porém, seu esquema difere do tradicional esquema estrela comumente utilizado na modelagem de DW.

Outro benchmark consolidado para DW convencionais é o *Star Schema Benchmark* (SSB) (O'NEIL, P. E. et al., 2009). Projetado a partir de modificações no TPC-H, o SSB possui um esquema estrela puro (Figura 2.11) composto por uma tabela de fatos de pedidos (`Lineorder`) e de quatro tabelas de dimensão (`Customer`, `Supplier`, `Part`, e `Date`). Nas tabelas de dimensão `Customer` e `Supplier` pode-se perceber a presença de uma hierarquia predefinida entre os atributos que descrevem a localização, i.e., `region` $\preceq$ `nation` $\preceq$ `city` $\preceq$ `address`. Entretanto, o SSB não possui atributos espaciais que possibilitem a execução de consultas multidimensionais com predicados espaciais. Ou seja, as

tabelas `Customer` e `Supplier` são tabelas de dimensão espacial não-geométricas.

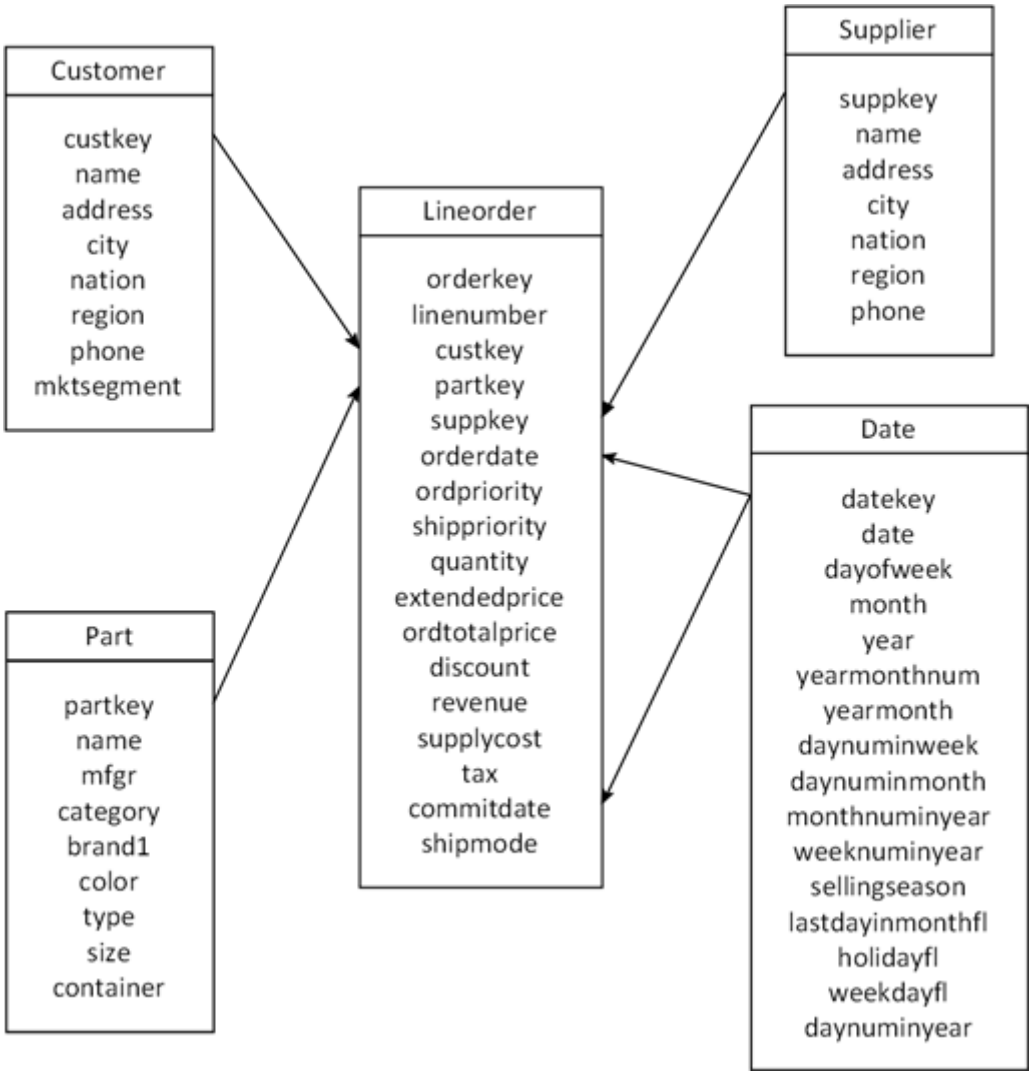


Figura 2.11. Esquema do SSB. Adaptada de O'Neil, P. E. et al. (2009).

Um novo benchmark chamado Spadawan (*Spatial Data Warehouse Benchmark*) foi proposto por Siqueira et al. (2010) a fim de analisar o desempenho do processamento de consultas OLAP em DWE. Baseado no SSB, o Spadawan possui dois esquemas, o redundante e o híbrido (Figura 2.12). Ambos mantêm a mesma estrutura do esquema do SSB, com o diferencial de conter atributos espaciais para a representação da localização nas tabelas de `Customer` e `Supplier`. Nos esquemas redundante e híbrido, o atributo `address_geo` é associado a um ponto e os demais atributos espaciais referentes à localização (`city_geo`, `nation_geo` e `region_geo`) são representados por polígonos.

Como pode ser visto na Figura 2.12.a, o esquema redundante possui dados espaciais redundantes, por exemplo, o mapa da Ásia é armazenado em todas as

tuplas cujo fornecedor está localizado na Ásia. Por sua vez, o esquema híbrido (Figura 2.12.b) foi construído considerando-se que clientes e fornecedores compartilham cidade, nação e região. Enquanto o primeiro esquema foi projetado com o objetivo de investigar quais consultas espaciais são afetadas pela redundância dos dados, o segundo permite analisar o custo da introdução de junções espaciais no processamento de consultas, uma vez que essas junções são necessárias para evitar a redundância dos dados.

É importante ressaltar que o esquema híbrido não se trata de um esquema floco de neve, uma vez que esse último consiste na normalização de hierarquias de atributos.

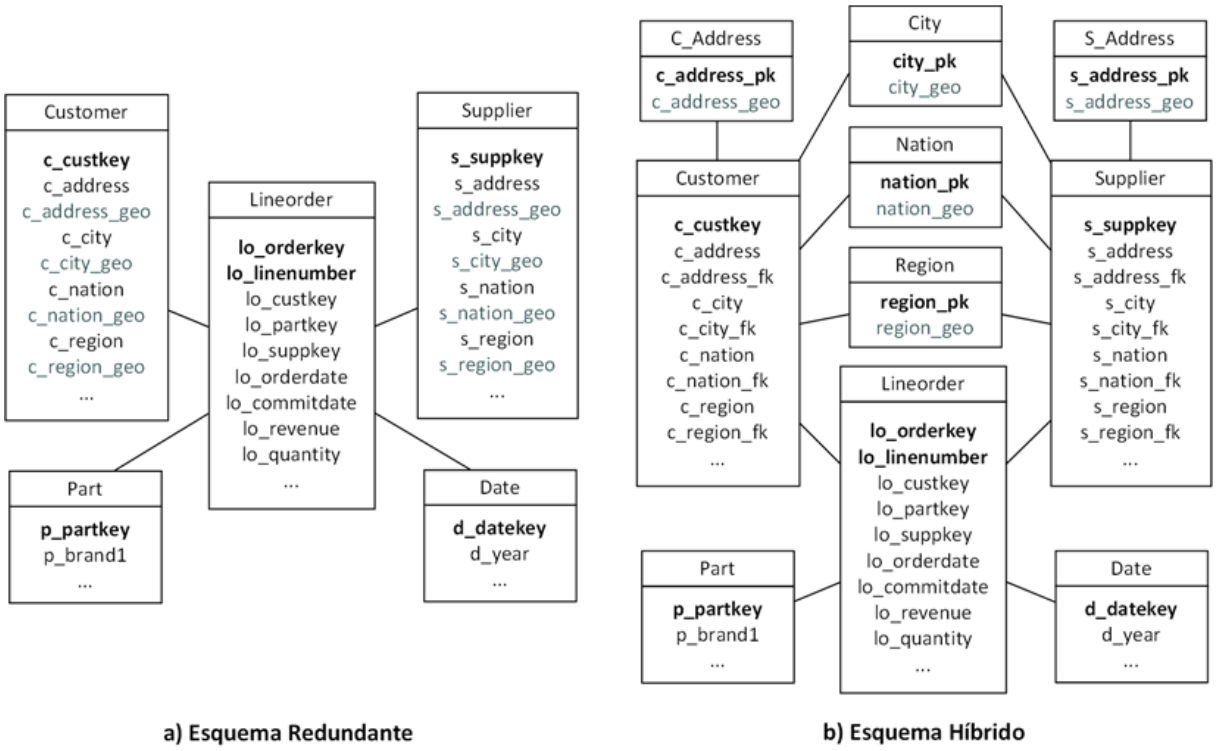


Figura 2.12. Esquema do Spadawan. Adaptada de Siqueira et al. (2010).

Recentemente, Nascimento et al. (2011) propuseram outro benchmark para avaliar o desempenho de aplicações para DWE denominado *Spatial Star Schema Benchmark* (Spatial SSB). O esquema do Spatial SSB (Figura 2.13) é baseado no SSB, sendo bastante similar ao esquema híbrido do Spadawan. A diferença está no acréscimo da tabela `Street`, cujo atributo `street_geo` é representado por uma linha, e não houve a criação das tabelas de dimensão espacial para os endereços de clientes e fornecedores. Eles foram colocados nas próprias tabelas de `Customer` e `Supplier`, já que todos são distintos e há um relacionamento de 1:1

entre o endereço e a chave primária da tabela de dimensão, tanto para `Customer` quanto para `Supplier`.

A principal diferença entre o Spatial SSB e o Spadawan está na natureza dos dados espaciais. Enquanto o Spadawan utiliza dados espaciais adaptados do TIGER/Line (U.S. CENSUS BUREAU, 2012), um banco de dados real, o Spatial SSB utiliza dados sintéticos, o que permite controlar com acurácia algumas características dos dados, como a quantidade de pontos por tipo de objeto espacial e o volume de dados a ser gerado.

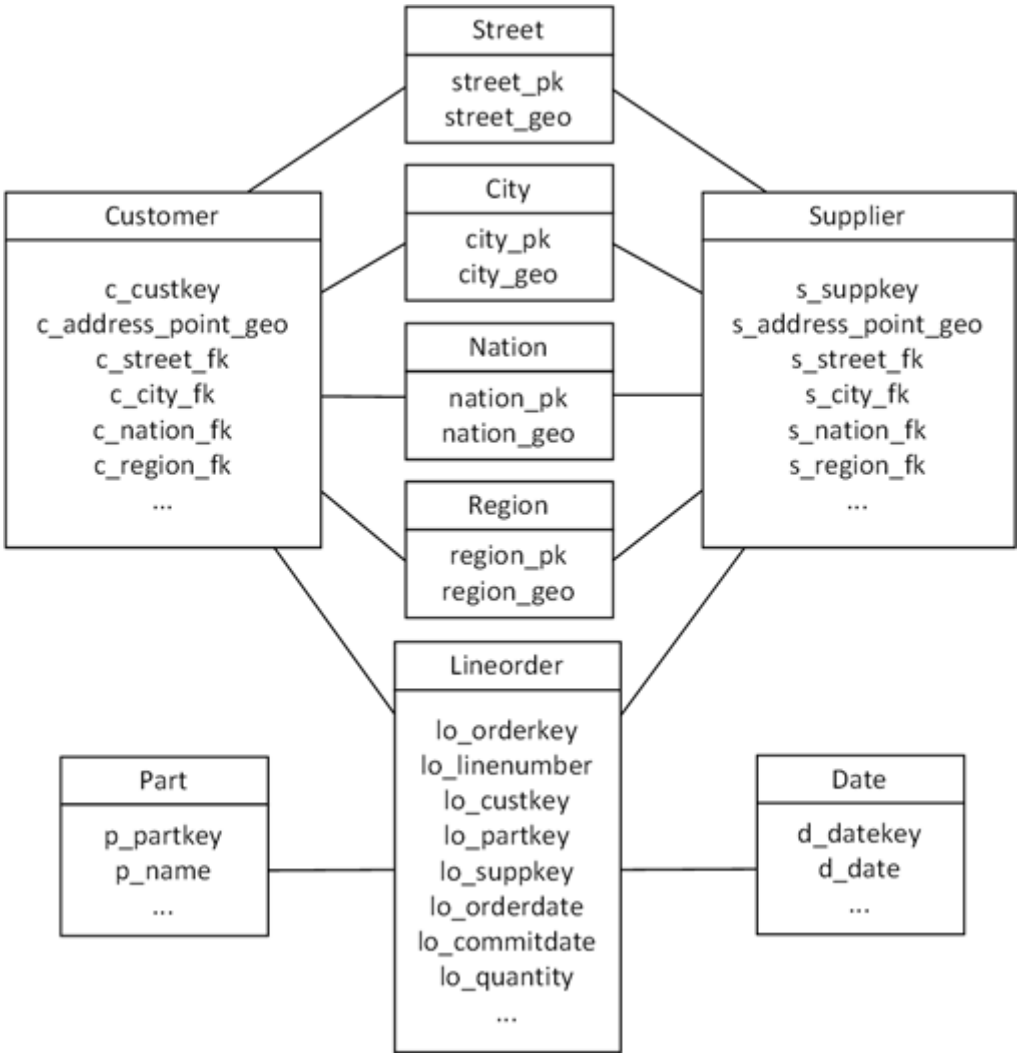


Figura 2.13. Esquema do Spatial SSB. Adaptada de Nascimento et al. (2011).

Capítulo 3

REVISÃO DA LITERATURA

Este capítulo apresenta o SB-index, o HSB-index e a aR-tree, eficientes estruturas de indexação para DWE, os quais foram estudados para a criação de um índice para DWET. Além disso, são descritos dois índices para DWET, a a3DR-tree e a aRB-tree.

3.1 aR-tree

A aR-tree (*aggregate R-tree*) é um índice para DWE baseado na R-tree que mantém em suas entradas os valores de uma função de agregação sobre uma medida da tabela de fatos (PAPADIAS et al., 2001).

Em aplicações OLAP é possível definir hierarquias de atributos em diversas dimensões. Um exemplo de hierarquia é `ano` $\preceq$ `trimestre` $\preceq$ `mês`, em que `mês` constitui o nível de menor granularidade. O usuário pode também submeter consultas com agrupamento por `trimestre` ou por `ano`. Para tornar ágil a resposta a consultas que envolvem agregação, os resultados das consultas podem ser pré-calculados e materializados.

Há várias técnicas que promovem a pré-agregação de medidas em DW convencionais com o propósito de melhorar o desempenho no processamento de consultas OLAP. Porém, diferentemente do que ocorre em um DW convencional, em um DWE pode haver pouco ou nenhum conhecimento prévio sobre a hierarquia de atributos espaciais, a qual permite o agrupamento dos objetos espaciais. As hierarquias de atributos não definidas a priori durante o projeto são chamadas na literatura de hierarquias *ad-hoc*. Além disso, as consultas espaciais podem requerer agrupamentos baseados em mapas criados arbitrariamente, ao invés de apenas

solicitar as regiões espaciais predefinidas nas tabelas de dimensão. Dessa forma, os métodos de agregação prévia utilizados para melhorar o desempenho de operações OLAP são inadequados em alguns casos para dados espaciais.

Motivados por esse fato, Papadias et al. (2001) propuseram um método que combina indexação espacial com resultados pré-agregados. Um índice é construído sobre os objetos de menor granularidade da dimensão espacial, o qual consiste em uma R-tree que armazena, para cada MBR, o valor da função de agregação para todos os objetos nele incluídos. Uma vez que os agrupamentos de objetos espaciais realizados pelo índice criam uma hierarquia implícita, tal hierarquia foi incorporada à treliça de Harinarayan et al. (1996), possibilitando selecionar as agregações apropriadas para materialização.

Há diversas aplicações que podem se beneficiar desse método. Papadias et al. (2001) utilizam o exemplo de um sistema de supervisão de tráfego que monitora as posições dos veículos numa cidade e o tráfego nas ruas. Nesse sistema, algumas consultas que podem ser respondidas eficientemente são: "*encontre o total de carros dentro de cada bairro*" ou "*encontre o segmento de rua com o maior tráfego em um raio de 2 km ao redor de cada hospital*".

3.1.1 Estrutura de Dados

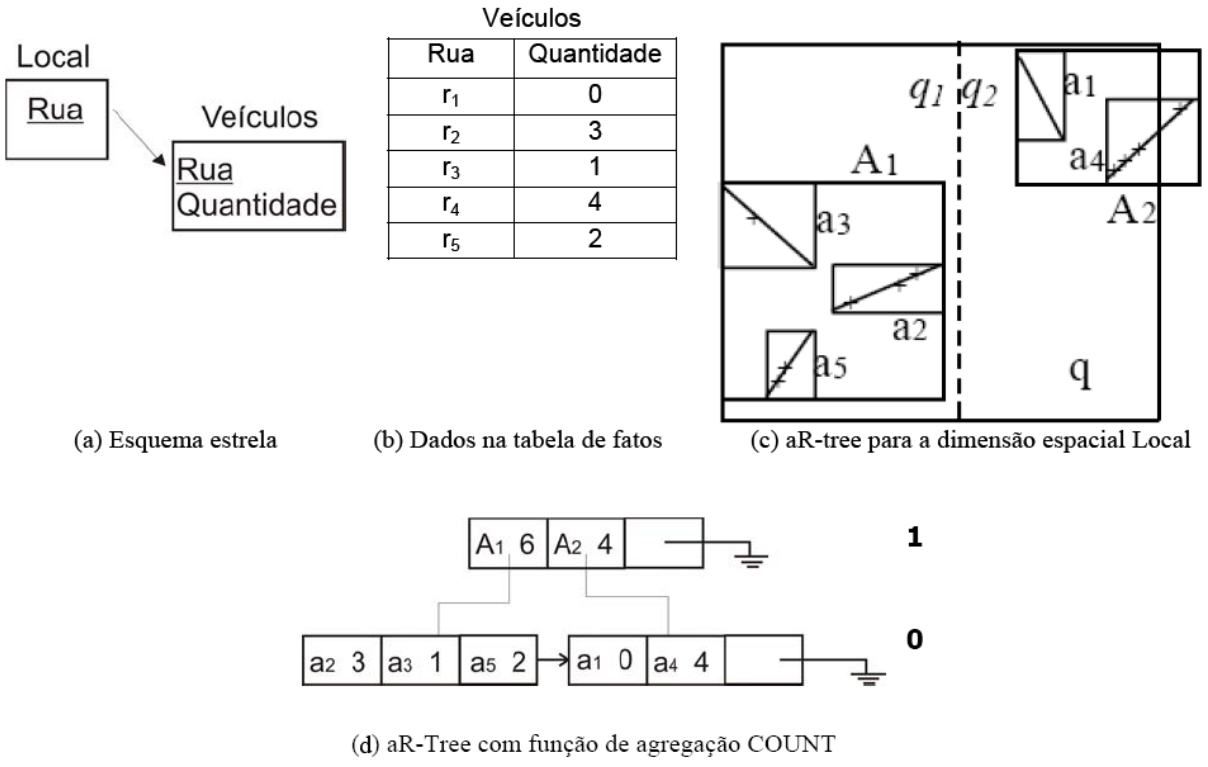
A aR-tree é uma R-tree que armazena, para cada MBR, o valor da função de agregação para todos os objetos nele contidos, sendo construída a partir do objeto espacial de menor granularidade da dimensão espacial.

A Figura 3.1 apresenta uma aR-tree que indexa um conjunto de cinco segmentos de rua r_1, ..., r_5 representados pelos seus respectivos MBR a_1, ..., a_5. O DWE do exemplo para a supervisão de tráfego é representado por um esquema estrela cuja tabela de fatos deve conter os resultados agregados para a dimensão espacial no menor nível de granularidade. Esse nível define a quantidade de carros por segmento de rua. A Figura 3.1.a exibe um DWE com apenas uma tabela de dimensão espacial chamada `Local`, para simplificar a compreensão da aR-tree. Na Figura 3.1.b, um fragmento da tabela de fatos é mostrado. Já a Figura 3.1.c consiste em uma representação gráfica da aR-tree.

Na estrutura de dados (Figura 3.1.d), cada nó X_i da aR-tree possui as entradas $X_{i,1}$ ... $X_{i,CPi}$, sendo que CP é a capacidade de X_i e $CP_i \leq CP$. Em cada

entrada da forma <$X_{i,k}$.MBR, $X_{i,k}$.ref, $X_{i,k}$.agr>, o ponteiro $X_{i,k}$.ref referencia o nó X_k do próximo nível inferior. Logo, a raiz constitui o maior nível da árvore. O resultado pré-agregado para cada entrada é denotado por $X_{i,k}$.agr, mantendo todos os valores para cada resultado da função de agregação (COUNT, MAX, MIN, AVG, etc.). Cada nó X_i possui um ponteiro denominado X_i.next para o próximo nó X_{i+1} do mesmo nível. Cada objeto no nível $n-1$ pertence a exatamente um MBR do nível n, permitindo o armazenamento de resultados parciais e a agregação deles num passo posterior. Com isso, são obtidos resultados mais resumidos, ou seja, efetua-se uma operação de *roll-up*.

Ainda na Figura 3.1.d, como existem 3 carros na rua r_2, há uma entrada (a_2,3) em um nó folha da aR-tree. No nível superior, o MBR A_1 contém três ruas, r_2, r_3 e r_5. O número total de carros nessas ruas é 6, por isso há uma entrada (A_1,6) no nível 1 da árvore.



(a) Esquema estrela (b) Dados na tabela de fatos (c) aR-tree para a dimensão espacial Local

(d) aR-Tree com função de agregação COUNT

Figura 3.1. aR-tree correspondente ao DWE com a dimensão espacial Local. Adaptada de Siqueira (2009).

Ao invés de armazenar um resultado em cada entrada da árvore, é possível armazenar uma lista de resultados para todas as funções de agregação de interesse. A Figura 3.2 mostra um exemplo com as funções de agregação COUNT e MAX, em que a entrada (A_1,6,3) denota que a soma dos carros presentes nas ruas

r_2, r_3 e r_5, contidas em A_1, é igual a 6, e o número máximo de carros em uma rua contida em A_1 é 3, no caso a rua r_2. Dessa forma, é possível trabalhar com funções de agregação algébricas, como a média de carros por rua, por exemplo, em que seria necessário armazenar o total de carros presentes nas ruas de um determinado nó e a quantidade de ruas contidas nesse nó.

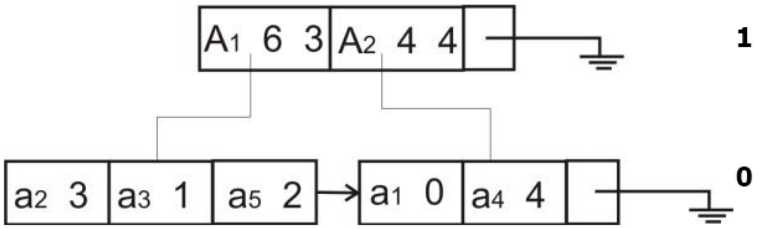


Figura 3.2. aR-tree com função de agregação COUNT e MAX. Adaptada de Siqueira (2009).

Como citado anteriormente, uma vez que os agrupamentos de objetos espaciais realizados pelo índice criam uma hierarquia implícita, tal hierarquia pode ser incorporada à treliça de Harinarayan et al. (1996), permitindo que a aR-tree desfrute de todas as vantagens conferidas aos DW convencionais. Abaixo seguem descritas algumas dessas vantagens.

A primeira vantagem é o suporte a múltiplas dimensões. Quando houver mais de uma tabela de dimensão no esquema estrela, deve-se manter em cada entrada da árvore um ponteiro para um vetor multidimensional. Esse vetor indica os valores agregados para todo o domínio das outras dimensões. Por exemplo, seja o esquema estrela da Figura 3.3.a e um fragmento dos dados de sua tabela de fatos na Figura 3.3.b. A dimensão `TipoVeículo` é convencional.

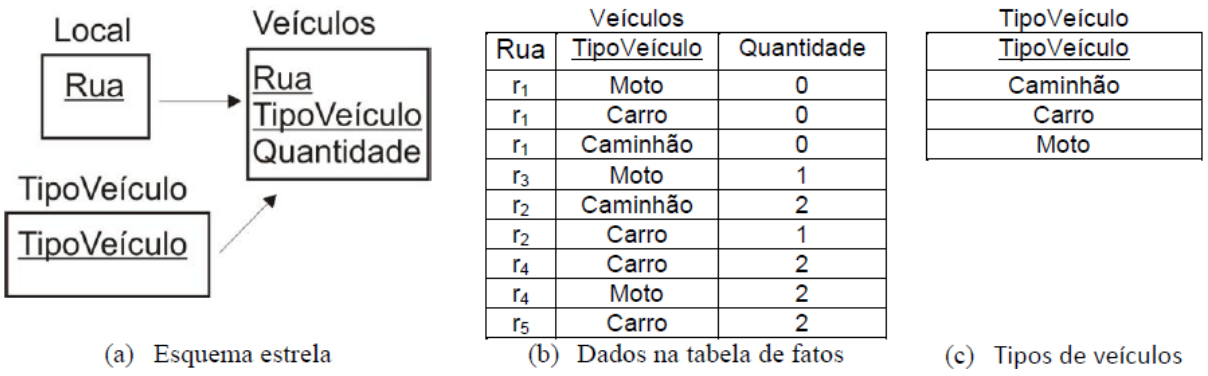


Figura 3.3. aR-tree com uma dimensão espacial (Local) e uma dimensão não espacial (Veículos) (SIQUEIRA, 2009).

A aR-tree correspondente ao fragmento exibido pela Figura 3.3 é mostrada na Figura 3.4. Presume-se que a ordem dos dados mantidos na tabela de dimensão `TipoVeículo` (Figura 3.3.c) é respeitada na disposição dos dados em cada vetor

unidimensional que acompanha cada entrada dos nós da árvore. Essa suposição é feita porque Papadias et al. (2001) apenas mencionam que deve haver um vetor multidimensional para os valores das demais dimensões, mas não explicam sua implementação. No exemplo, a entrada que mantém o MBR A_1 tem o vetor {2, 1, 3}. Isto é, no interior de A_1 há 2 caminhões, 1 carro e 3 motos.

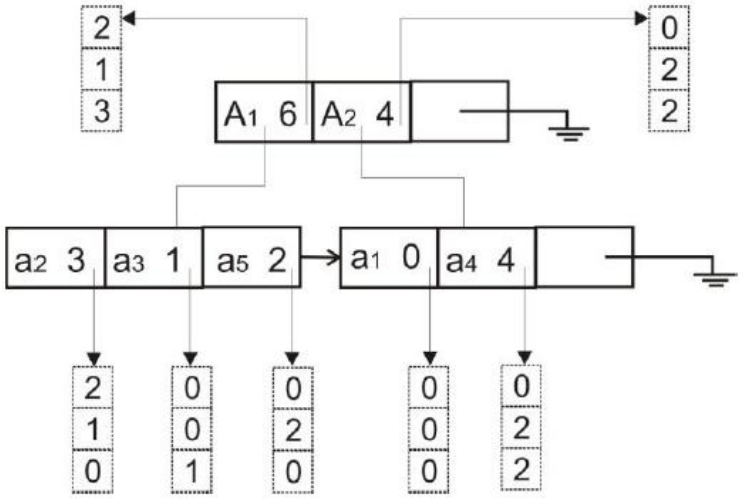


Figura 3.4. aR-tree com mais de uma dimensão (SIQUEIRA, 2009).

Se houver no esquema estrela mais que uma dimensão espacial, então uma aR-tree distinta deve ser criada para cada uma delas.

Outra vantagem diz respeito à materialização de visões, que consiste na seleção e materialização de determinados níveis da árvore a fim de acelerar o processamento de consultas OLAP. Isso é realizado criando-se uma visão da árvore por meio de redefinições de ponteiros, interligando alguns de seus níveis e dispensando outros intermediários. A construção de uma visão materializada não modifica o índice, pois cria uma estrutura à parte. Um esboço de uma aR-tree parcialmente materializada é mostrada na Figura 3.5, em que apenas os níveis dos nós raiz e folha foram materializados.

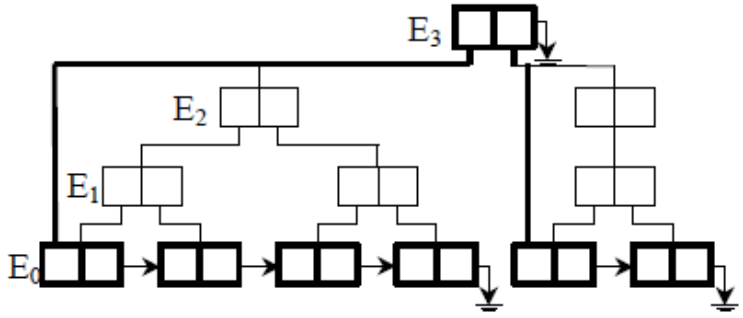


Figura 3.5. Materialização parcial da aR-tree (PAPADIAS et al., 2001).

Para hierarquias conhecidas em tempo de projeto, a materialização é preferível à indexação usando a aR-tree. Logo, o processamento de consultas que envolvam uma dimensão espacial `Local` com a hierarquia de relacionamento de atributos `região` $\preceq$ `nação` $\preceq$ `cidade` $\preceq$ `endereço` pode ser feito com maior agilidade usando visões materializadas do que usando a indexação com a aR-tree (PAPADIAS et al., 2001). Contudo, as dimensões com hierarquias *ad-hoc* (sistema de supervisão de tráfego exemplificado) ainda necessitam desse índice para obter um bom desempenho no processamento de consultas.

Papadias et al. (2001) citam também que a estrutura da aR-tree permite medidas espaciais. Nesse caso, o valor da função de agregação é substituído por um ponteiro para o objeto agregado. Além disso, medidas numéricas e espaciais podem coexistir na mesma árvore.

3.1.2 Processamento de Consultas

Na aR-tree, o processamento de consultas se inicia a partir do nó raiz e prossegue recursivamente em direção aos nós folha. Suponha o exemplo apresentado anteriormente na Figura 3.1. Para todas as entradas dos nós visitados, uma das seguintes condições ocorre:

- O MBR da entrada e a JC (janela de consulta) são disjuntos, então os nós inferiores que pertencem à sub-árvore dessa entrada não contém carros que contribuem para a resposta, não precisando ser acessados.
- O MBR da entrada está contido na JC. Como a informação agregada é armazenada na entrada, recupera-se essa informação, evitando acesso ao nó filho.
- O MBR da entrada sobrepõe parcialmente a JC. Nesse caso, o nó filho dessa entrada deve ser percorrido. Para entradas pertencentes a nós folha, a resposta pode ser estimada ou computada acessando os dados originais.

O algoritmo de pesquisa da aR-tree é similar ao da R-tree, mas há uma diferença fundamental entre eles. Na R-tree, se a JC for muito grande, utiliza-se a busca sequencial nos MBR dos objetos ao invés do índice. Já na aR-tree, para consultas que promovem agregação, existem dois casos:

- A JC é grande. Logo, muitos nós intermediários da aR-tree estarão contidos na JC. Então, os resultados pré-calculados são utilizados, evitando-se visitar os objetos individualmente.
- A JC é pequena. Então, a aR-tree age como um MAM, permitindo a seleção dos objetos candidatos.

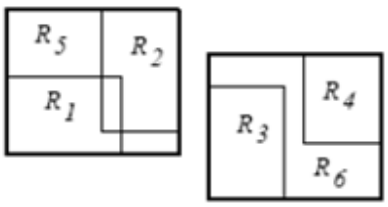
3.2 Índices a3DR-tree e aRB-tree

Papadias et al. (2002) motivados pelo acúmulo de grandes volumes de dados multidimensionais envolvendo tempo em aplicações diversas, como supervisão de tráfego e sistemas de telefonia celular, propuseram estruturas de indexação para DWET. Sua abordagem modela as dimensões espacial e temporal em uma mesma dimensão combinada no hipercubo, integrando indexação espaço-temporal com pré-agregação de medidas.

São dois os casos tratados em seu trabalho. Um deles considera que as dimensões espaciais são dinâmicas, ou seja, evoluem no tempo. Por exemplo, a área de uma célula de cobertura de rádio que, exposta a determinadas condições climáticas, pode ter seu raio alterado. O outro caso considera dimensões espaciais estáticas, como a análise do tráfego nas ruas de uma cidade, e o objetivo é manter informações agregadas referentes a cada objeto espacial conforme a evolução do tempo. Essa segunda abordagem é tratada por Papadias et al. (2002), em que duas propostas de índices são apresentadas, a a3DR-tree e a aRB-tree, que são descritas abaixo.

A primeira proposta é uma generalização da aR-tree para o espaço tridimensional, denominada a3DR-tree (*aggregate 3DR-tree*). Cada entrada `r` tem a forma <`r.MBR`, `r.pointer`, `r.lifespan`, `r.aggr[]`>, referente ao MBR do objeto espacial indexado, ao ponteiro para um nó de nível inferior (ou para o endereço efetivo da tupla), ao intervalo durante o qual o valor da medida é válido e à medida agregada, respectivamente. Sempre que a medida agregada se altera, uma nova entrada é criada. A Figura 3.6.c exibe as entradas requeridas para uma região R_1 de acordo com os valores agregados (Figura 3.6.b), supondo a R-tree da Figura 3.6.a.

Embora a a3DR-tree possibilite a integração das dimensões espacial e temporal na mesma estrutura, ela desperdiça espaço em disco porque armazena o MBR a cada vez que há mudança na medida agregada. Por exemplo, na Figura 3.6, o MBR de R_1 é armazenado 4 vezes . Além disso, o grande tamanho da estrutura e o pequeno *fan-out* dos nós comprometem a eficiência do processamento de consultas OLAP em disco magnético.



(a) Exemplo de R-tree

Região	Tempo	Quantidade
R_1	1	150
R_1	2	150
R_1	3	145
R_1	4	135
R_1	5	130

(b) Fragmento da tabela de fatos

<R_1.MBR, R_1.ptr, [1, 2], 150>
<R_1.MBR, R_1.ptr, [3, 3], 145>
<R_1.MBR, R_1.ptr, [4, 4], 135>
<R_1.MBR, R_1.ptr, [5, 5], 130>

(c) Entradas para R_1

Figura 3.6. Exemplo da a3DR-tree. Adaptada de Papadias et al. (2002).

A segunda proposta, a aRB-tree (*aggregate RB-tree*), é baseada no seguinte conceito: as regiões que constituem a hierarquia espacial são armazenadas uma única vez e são indexadas por uma R-tree. Para cada entrada da R-tree, há um ponteiro para uma B-tree que armazena os valores históricos agregados referentes à entrada.

Cada entrada `r` da R-tree tem a forma <`r.MBR`, `r.pointer`, `r.btree`, `r.aggr[]`>, em que `r.MBR` e `r.pointer` tem seus significados usuais, `r.aggr[]` mantém dados resumidos sobre `r` acumulados por todos os intervalos e `r.btree` é um ponteiro para a B-tree que mantém dados históricos sobre `r`. Cada entrada `b` da B-tree tem a forma <`b.time`, `b.pointer`, `b.aggr[]`>, em que `b.aggr[]` corresponde ao dado agregado no tempo `b.time`. Se o valor de `b.aggr[]` não se alterar em intervalos consecutivos, ele não é replicado.

A Figura 3.7 ilustra uma aRB-tree para as regiões da Figura 3.6.a. Por exemplo, o valor 1130 armazenado na entrada R_5 da R-tree denota que o número total de objetos em R_5 é 1130. Já a primeira entrada folha na B-tree para R_5 (1, 225) informa que o número de objetos em R_5 durante o intervalo [1, 1] é 225. De modo similar, a primeira entrada no nó raiz da B-tree (1, 685) denota que o número de objetos durante o intervalo [1, 3] é 685. A B-tree mais ao topo corresponde ao nó raiz da R-tree e armazena informações referentes ao espaço todo.

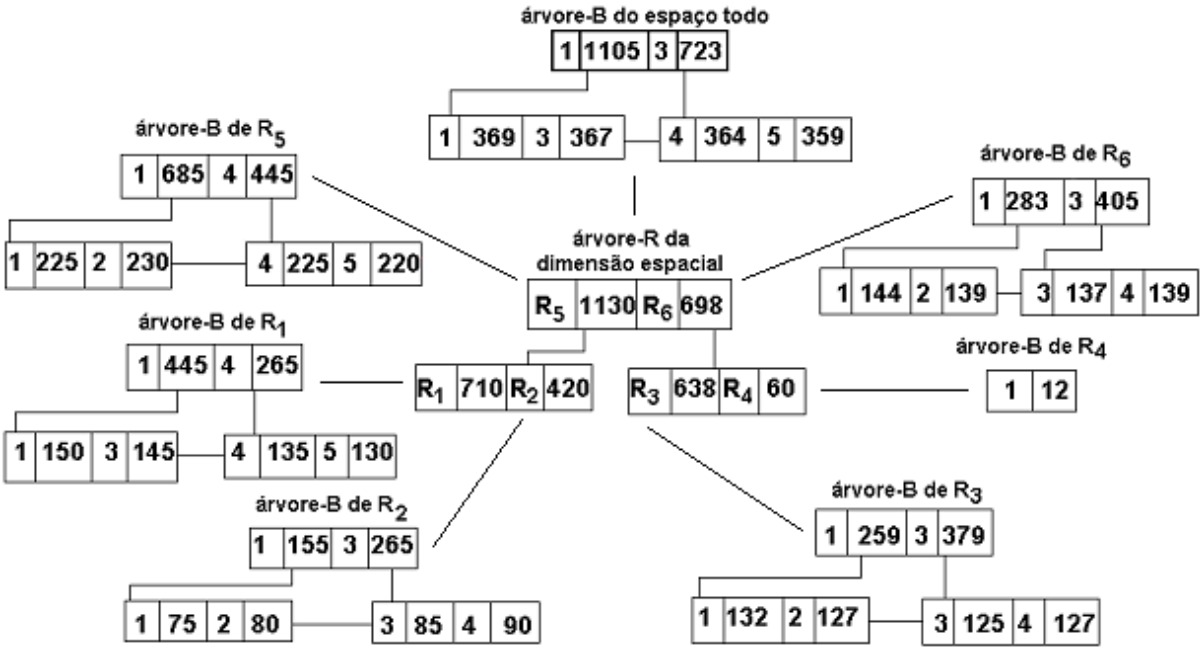


Figura 3.7. Um exemplo de aRB-tree para as regiões da Figura 3.6.a. Adaptada de Papadias et al. (2002).

A aRB-tree facilita o processo de agregação dos resultados de uma consulta, pois elimina a necessidade de visita a nós que estão totalmente incluídos na janela de consulta. Como exemplo, considere que um usuário esteja procurando todos os objetos sobrepostos pela janela de consulta da Figura 3.8 durante o intervalo [1,3]. A busca começa na raiz da R-tree. A entrada R_5 está totalmente contida na janela de consulta, então a B-tree de R_5 é acessada. O nó raiz dessa B-tree tem as entradas (1, 685) e (4, 445), que denotam os dados resumidos para os intervalos [1, 3] e [4, 5], respectivamente. Então o próximo nó da B-tree não precisa ser acessado e a contribuição de R_5 para o resultado da consulta é 685.

A segunda entrada do nó raiz da árvore, R_6, sobrepõe parcialmente a JC, fazendo com que o nó seja visitado. Dentro dele, apenas R_3 intersecta a JC, e sua B-tree é acessada. A primeira entrada da raiz dessa B-tree determina que a colaboração de R_3, para o intervalo [1, 2], é 259. Para completar o resultado, é necessário percorrer o nível inferior e recuperar o valor agregado de R_3 para o tempo 3, que é 125. O resultado final da consulta é a soma 685+259+125, que corresponde aos dados agregados das células em cinza da Figura 3.8.

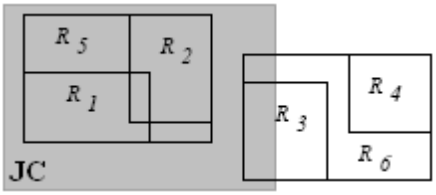


Figura 3.8. Exemplo de consulta para a aRB-tree. Adaptada de Papadias et al. (2002).

Em resumo, a aRB-tree é mais do que apenas uma estrutura de indexação, ela consiste no próprio cubo de dados do DW espaço-temporal. Se os dados agregados não são muito dinâmicos, espera-se que o tamanho da estrutura seja menor do que o cubo, pois não replica informações que permanecem constantes em intervalos de tempo adjacentes. Mesmo no pior caso, em que os dados agregados de todas as regiões se alteram a cada rótulo de tempo, o tamanho da aRB-tree é aproximadamente o dobro do tamanho do cubo. Isto porque os nós folha consomem pelo menos metade do espaço.

Os testes de desempenho da a3DR-tree e da aRB-tree foram realizados por Papadias et al. (2002) implementando a B+-tree (FOLK; ZOELLICK, 1991) e a R*-tree (BECKMANN et al., 1990). Dois parâmetros têm influência direta sobre os resultados: o tamanho da JC e o tamanho do intervalo de tempo. O primeiro determina que, à medida que a janela cresce, aumenta o número de B-trees que precisam ser visitadas. O segundo indica que o número de B-trees acessadas pela aRB-tree é constante, porque longos intervalos têm chance de ser respondidos pelo nó raiz ou nós próximos dele.

Por outro lado, conforme crescem os intervalos, a chance da informação agregada também se alterar igualmente aumenta. Logo, há o crescimento linear do custo da a3DR-tree, porque diversas versões do mesmo nó devem ser recuperadas (i.e., cada uma tem valores diferentes da medida agregada). Portanto, para intervalos curtos, a a3DR-tree é menos custosa que a aRB-tree, já que não tem necessidade de acessar uma B-tree para recuperar dados resumidos. Quando os intervalos crescem, a aRB-tree supera a a3DR-tree.

3.3 Índice de Projeção e Índice Bitmap de Junção

Índice de projeção (O'NEIL, P.; QUASS, 1997) é uma estrutura simples que consiste em, dada uma coluna `X` de uma tabela `T`, armazenar os valores que a coluna `X` assume, ordenados segundo a numeração das tuplas em `T`. A Figura 3.9.b exemplifica um índice de projeção sobre a coluna `s_nation` considerando a tabela de dimensão apresentada na Figura 3.9.a.

O índice bitmap (STOCKINGER; WU, 2007) convencional associa um vetor de bits a cada valor distinto de uma coluna `Y` indexada de uma tabela `S`, sendo que cada vetor contém uma quantidade de bits igual ao número de tuplas em `S`. Se o `i`-ésimo registro da tabela `S` contém o valor `v` na coluna `Y`, então o `i`-ésimo bit do vetor associado a `v` terá valor 1 (um). Caso contrário, terá valor 0 (zero). A cardinalidade da coluna indexada define a quantidade de vetores de bits que deverão ser armazenados.

Índices bitmap são usados para indexar DW convencionais, evitando a computação de junções estrela (SIQUEIRA et al., 2012). Para que isso seja possível, um Índice Bitmap de Junção (IBJ) deve ser construído sobre uma coluna `Z` de uma tabela de dimensão possibilitando a junção das tuplas da tabela de fatos com determinado valor em `Z`. A Figura 3.9.d exemplifica um índice bitmap de junção construído sobre a coluna `s_address` da tabela de dimensão `Supplier`. Apesar do atributo `s_address` não estar diretamente relacionado à junção estrela, há um relacionamento de 1:1 entre `s_address` e `s_suppkey`, e `s_suppkey` é referenciado por `lo_suppkey`. Se uma consulta tem como predicado `s_address='D'`, ao invés de realizar a junção da tabela de fatos `Lineorder` com a tabela de dimensão `Supplier`, é necessário apenas verificar as tuplas que contêm valor 1 (um) no vetor de bits do endereço `'D'`.

IBJ permitem ainda construir índices sobre uma coluna a partir de outro IBJ, explorando-se o rápido processamento de operações bit-a-bit. Isso é possível quando as tabelas contêm hierarquias de atributos, como ocorre com a tabela de dimensão `Supplier` da Figura 3.9.a, em que `s_region` $\preceq$ `s_nation` $\preceq$ `s_city` $\preceq$ `s_address`. Dessa forma, é possível construir o vetor de bits de

'Algeria' realizando uma operação de OR entre os vetores dos endereços 'D' e 'E' (Figura 3.9.e).

s_suppkey	s_address	s_city	s_nation	s_region
1	A	Vietnam 2	Vietnam	Asia
2	B	France 5	France	Europe
3	C	Romania 2	Romania	Europe
4	D	Algeria 6	Algeria	Africa
5	E	Algeria 0	Algeria	Africa

a) Tabela de dimensão: Supplier

s_nation
Vietnam
France
Romania
Algeria
Algeria

b) Índice de projeção

lo_suppkey	lo_custkey	lo_revenue
1	235	20785
1	512	166751
2	512	252145
3	235	197451
3	512	152389
3	106	217410
4	235	200489
5	106	182877

c) Tabela de fatos: Lineorder

A	B	C	D	E
1	0	0	0	0
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	1	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

d) Índice bitmap de junção construido sobre a coluna s_address

Vietnam	France	Roman	Algeria
1	0	0	0
1	0	0	0
0	1	0	0
0	0	1	0
0	0	1	0
0	0	1	0
0	0	0	1
0	0	0	1

e) Índice bitmap de junção construído sobre a coluna s_nation

Figura 3.9. Fragmento de dados, índice de projeção e índice bitmap de junção. Adaptada de Siqueira et al. (2012).

3.4 SB-index

O *Spatial Bitmap Index* (SB-index) proposto por Siqueira (2009) é um índice para DWE com enfoque em hierarquias predefinidas. Esse índice foi projetado para computar o predicado espacial de uma consulta SOLAP (*Spatial OLAP*) e transformá-lo em predicado convencional. Desse modo, tendo apenas predicados convencionais, a consulta pode ser respondida utilizando-se o IBJ, evitando assim operações de junção estrela.

3.4.1 Estrutura de Dados

A estrutura de dados do SB-index possui entradas compostas por um valor chave e um MBR. O valor chave corresponde à chave primária da tabela de dimensão espacial e identifica o objeto espacial representado pelo seu respectivo MBR.

Considerando o esquema híbrido da Figura 2.12.b, um conjunto de dados é apresentado na Figura 3.10. Nesse exemplo, `city_pk=1` está associado à `s_suppkey=3` e `lo_suppkey=3`. Assim, o `SB-index[0]` mantém o valor de chave igual a 1 e o MBR do objeto espacial que possui `city_pk=1`, como mostrado na Figura 3.11. Ainda na Figura 3.11, `B` corresponde ao IBJ definido sobre a coluna `city_pk` da tabela de dimensão `City`. O vetor de bits apontado por `B[0]` especifica as tuplas da tabela de fatos em que `city_pk=1`. Dessa forma, acessar o vetor de bits relacionado ao `SB-index[0]` requer apenas usar o endereço 0 (zero) para acessar `B`, ou seja, `B[0]`.

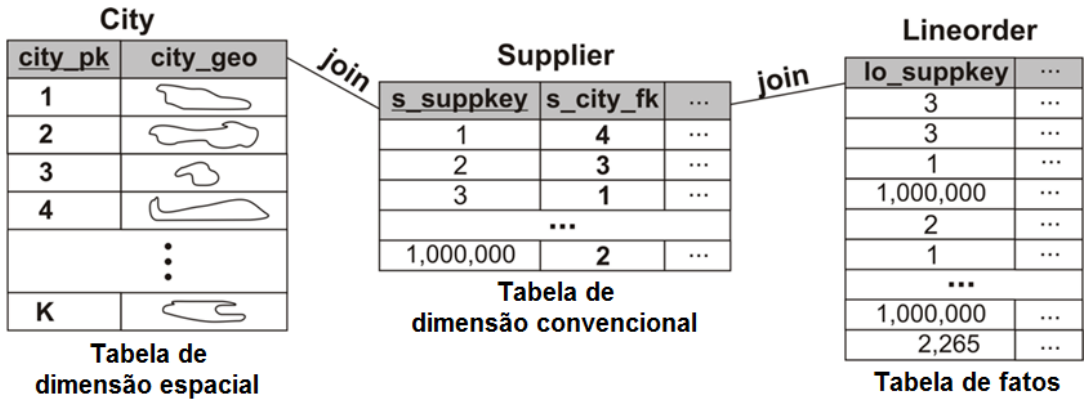


Figura 3.10. Tabela de dimensão espacial *City*, tabela de dimensão *Supplier* e tabela de fatos *Lineorder*. Adaptada de Siqueira et al. (2012).

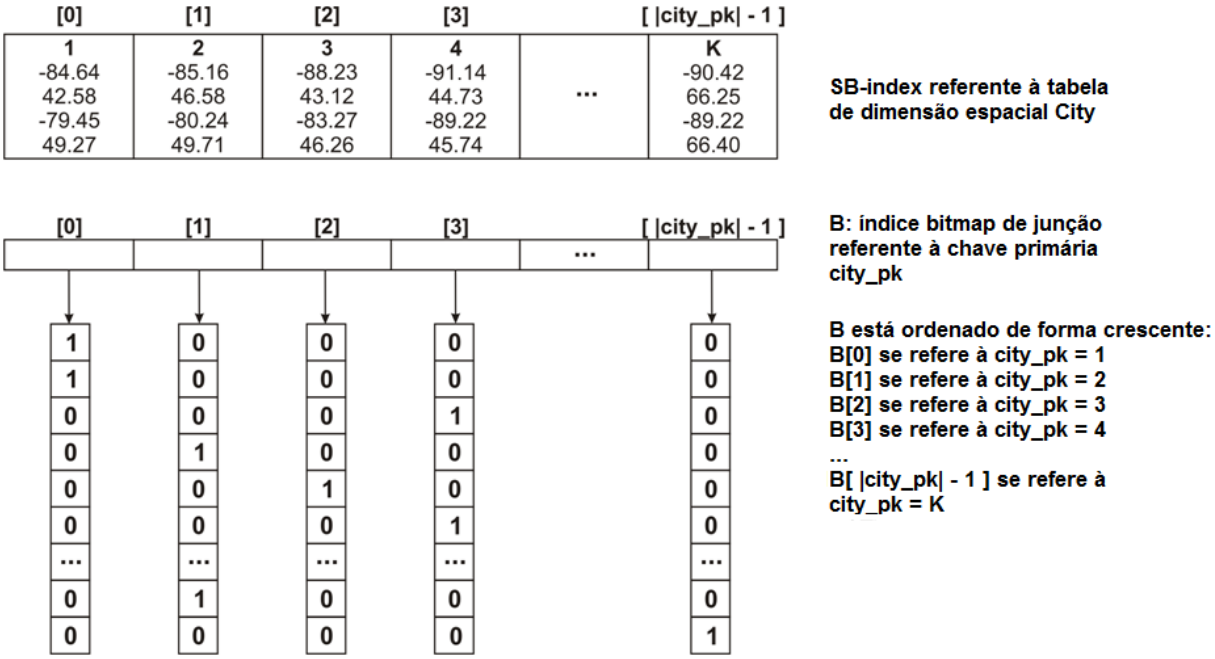


Figura 3.11. Estrutura de dados do SB-index. Adaptada de Siqueira et al. (2012).

3.4.2 Processamento de Consultas

O SB-index foi projetado para avaliar o predicado espacial de uma consulta em duas etapas. Cada uma delas é descrita como segue.

A primeira etapa (passos 1 a 3 da Figura 3.12) consiste em ler o arquivo do SB-index que está armazenado em disco (uma página de disco é lida por vez), copiar o conteúdo para a memória primária e verificar, para cada entrada da página de disco, se o MBR satisfaz o predicado espacial. Caso isso ocorra, significa que o objeto espacial é um possível candidato à resposta, e sua respectiva chave primária é adicionada a um conjunto de candidatos.

Após verificar todo o arquivo, prossegue-se para a segunda etapa (passos 4 e 5 da Figura 3.12). As chaves primárias adicionadas ao conjunto de candidatos na primeira etapa são utilizadas para recuperar a geometria exata do objeto espacial por meio do acesso à tabela de dimensão espacial e um refinamento é feito verificando-se se as geometrias satisfazem o predicado espacial. Caso o teste de satisfação do predicado espacial se confirme, a chave primária correspondente é selecionada e utilizada para compor um predicado convencional que é concatenado aos demais predicados convencionais da consulta original. O resultado final é obtido utilizando-se um IBJ.

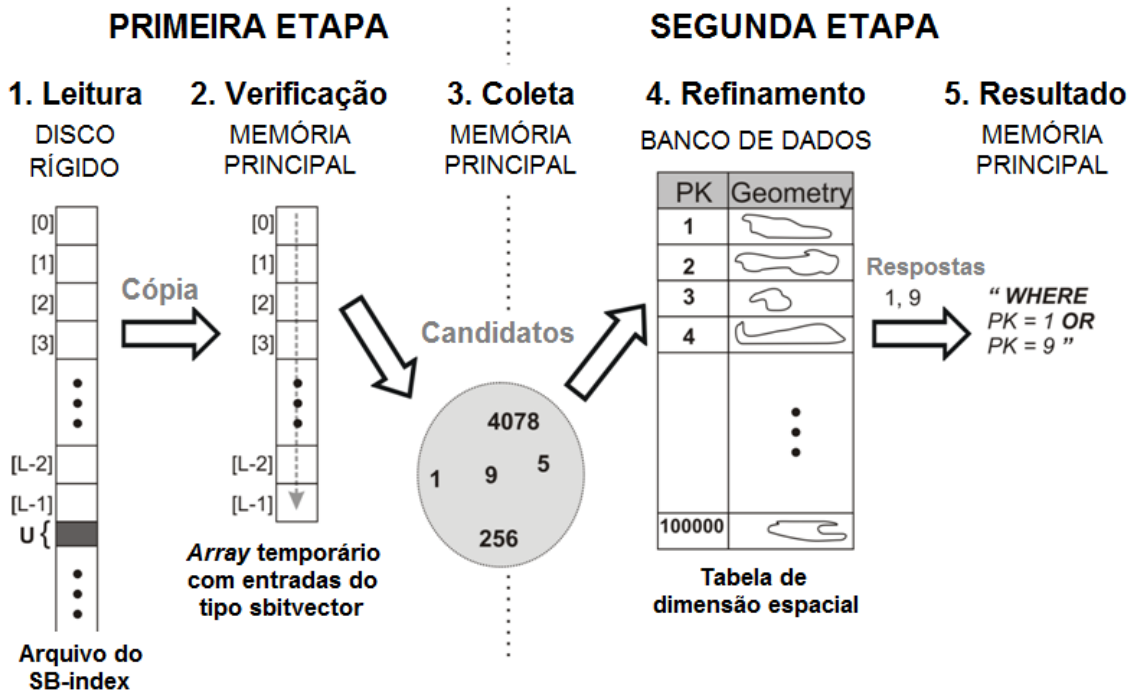


Figura 3.12. Processamento de consultas no SB-index. Adaptada de Siqueira et al. (2012).

3.5 HSB-index

Uma vez que o SB-index é conceitualmente um vetor armazenado em disco, ele requer uma verificação sequencial para testar se todas as entradas satisfazem o predicado espacial de uma consulta.

Motivados por esse fato, Siqueira et al. (2012) propuseram uma estrutura de indexação denominada *Hierarchical Spatial Bitmap Index* (HSB-index). O HSB-index possui as mesmas funcionalidades do SB-index, com a vantagem de reduzir o número de acessos a disco por meio do uso de técnicas de poda que exploram a natureza hierárquica dos dados. Tanto a estrutura de dados do HSB-index quanto o seu processamento de consultas são descritos a seguir.

3.5.1 Estrutura de Dados

O HSB-index possui uma estrutura de dados baseada em árvore, tal que cada entrada de um nó folha armazena um valor de chave referente à chave primária da tabela de dimensão espacial, o MBR correspondente e um ponteiro para um vetor de bits de um IBJ construído sobre a chave primária.

Uma vez que a estrutura dos nós internos não necessita de qualquer adaptação, qualquer índice espacial baseado em árvore com alguma técnica de clusterização pode ser usado.

A Figura 3.13 ilustra a estrutura do HSB-index para o conjunto de dados da Figura 3.10. Na Figura 3.13.a são representados os objetos espaciais da tabela de dimensão espacial `City`, enquanto que na Figura 3.13.b e na Figura 3.13.c são mostrados seus respectivos MBRs e a estrutura do índice espacial já com os clusters criados, respectivamente. Na Figura 3.13, podemos observar que o polígono identificado pelo valor de chave igual a 1 (Figura 3.13.a) é aproximado para `R1` (Figura 3.13.b) e clusterizado na região `M1` do índice espacial (Figura 3.13.c). Além disso, `R1` é armazenado em um nó folha juntamente com o valor chave 1 e associado a um vetor de bits em que os dois primeiros bits têm valor 1 (Figura 3.13.d), indicando que as duas primeiras tuplas da tabela de fatos referenciam o polígono de valor chave igual a 1.

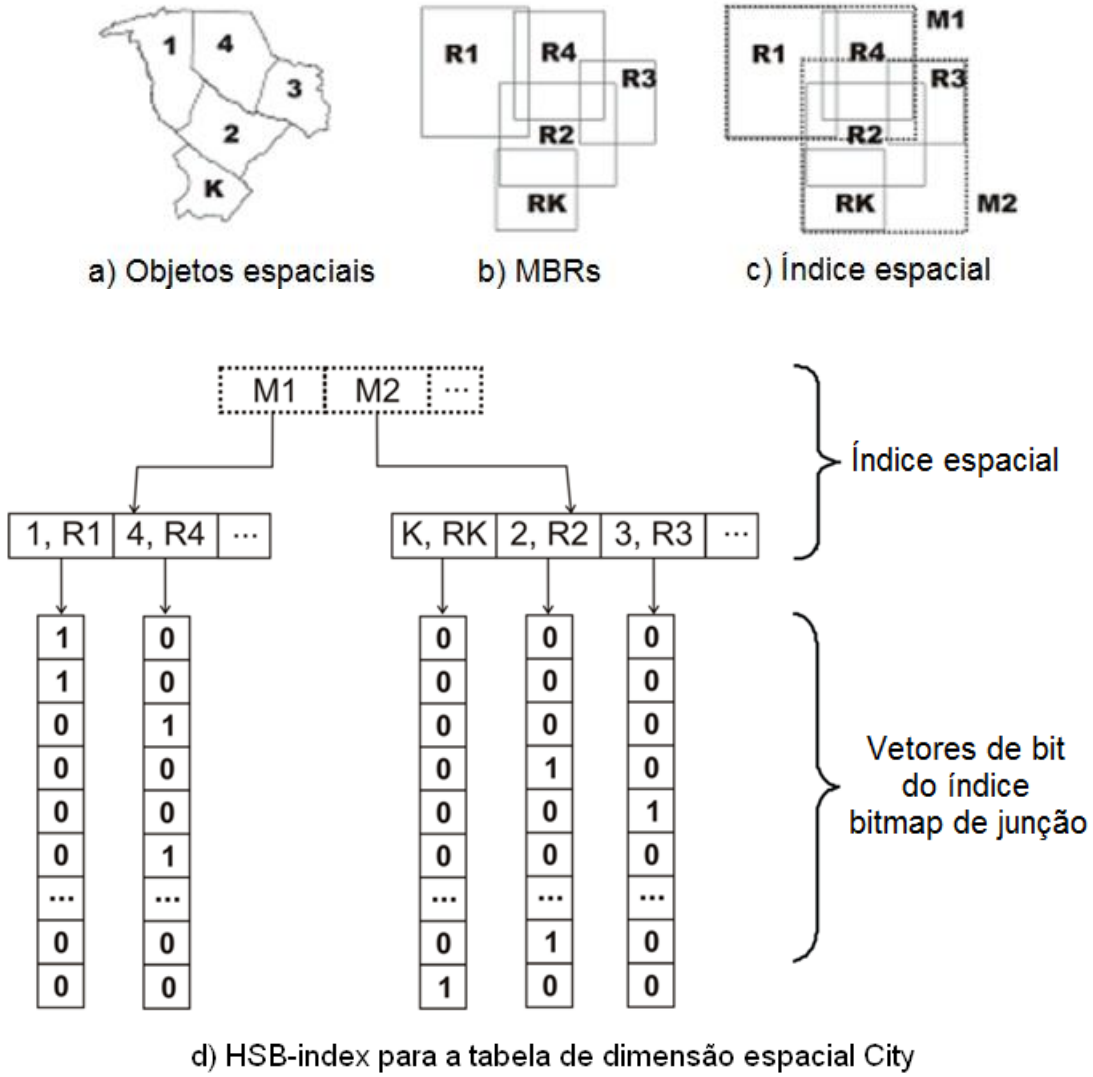


Figura 3.13. Estrutura de dados do HSB-index. Adaptada de Siqueira et al. (2012).

3.5.2 Processamento de Consultas

O processamento de consultas utilizando o HSB-index é similar ao do SB-index, com a diferença de que o SB-index avalia sequencialmente todas as entradas, enquanto o HSB-index é capaz de podar entradas por meio da clusterização dos objetos espaciais usando um índice espacial hierárquico. Em adição, o HSB-index utiliza um *buffer-pool* especializado que reduz ainda mais o custo do processamento de consultas. O *buffer-pool* refere-se a uma cópia parcial das entradas do HSB-index em memória primária.

3.6 Considerações Finais

Considerando o enfoque deste trabalho no estudo de estruturas de indexação para DW, a Tabela 3.1 foi elaborada para comparação das características de cada estrutura estudada.

Apesar da a3DR-tree e aRB-tree possuírem enfoque na indexação de DWET, essas estruturas foram projetadas considerando a abordagem relacionada a objetos móveis, como explicado na Seção 2.4. Já o STB-index, estrutura proposta neste trabalho, tem o enfoque na indexação de DWET contendo objetos espaciais que mudam aleatoriamente ao longo do tempo e está relacionado ao conceito de dimensões que mudam lentamente introduzido por Kimball e Ross (2002).

Por ser uma extensão do SB-index, o STB-index foi projetado para DWETs contendo hierarquias de atributos predefinidas e sua estrutura é sequencial.

Tabela 3.1. Características das Estruturas de Indexação Estudadas

Estrutura de Indexação	Enfoque	Hierarquia de Atributos	Estrutura
Bitmap (STOCKINGER; WU, 2007)	DW	Predefinida	-
SB-index (SIQUEIRA, 2009)	DWE	Predefinida	Sequencial
HSB-index (SIQUEIRA et al., 2012)	DWE	Predefinida	Hierárquica
aR-tree (PAPADIAS et al., 2001)	DWE	*Ad-hoc*	Hierárquica
a3DR-tree (PAPADIAS et al., 2002)	DWET	*Ad-hoc*	Hierárquica
aRB-tree (PAPADIAS et al., 2002)	DWET	*Ad-hoc*	Hierárquica
STB-index	DWET	Predefinida	Sequencial

Capítulo 4

SPATIO-TEMPORAL BITMAP INDEX

Neste capítulo é apresentado o STB-index, que consiste na estrutura de indexação proposta para DWET. Além disso, são descritos a sua estrutura de dados, o processo de construção do índice e também as etapas do processamento de consultas OLAP.

4.1 STB-index

O *Spatio-Temporal Bitmap Index*, ou apenas STB-index, é uma estrutura de indexação para DWET que provê suporte a consultas OLAP com predicados espaciais envolvendo geometrias que se modificam ao longo do tempo. As geometrias que se modificam ao longo do tempo estão armazenadas em tabelas de dimensão espacial, ou seja, referem-se a atributos espaciais das tabelas de dimensão cujos valores podem se modificar ao longo do tempo.

Assim como no SB-index, cada entrada do STB-index armazena um valor de chave e um MBR referentes a um determinado objeto espacial. A diferença está no armazenamento do tempo válido do objeto em cada entrada do STB-index, o que possibilita filtrar os objetos espaciais que irão para a fase de refinamento da consulta, que, como se sabe, é uma etapa muito custosa.

Por ser baseado no SB-index, o STB-index também foi projetado para DWETs cujos atributos espaciais formam uma hierarquia de atributos predefinida. Desse modo, o STB-index viabiliza o processamento de consultas contendo operações OLAP (tais como *drill-down* e *roll-up*) aliados a predicados espaciais (como exemplo, intersecta e está contido) sobre objetos espaciais cujas geometrias se modificam ao longo do tempo.

STB-index é uma estrutura de indexação para DWET cuja proposta: (i) introduz o Índice Bitmap em DWET; (ii) herda o legado do SB-index convertendo predicados espaço-temporais em predicados convencionais e usando técnicas criadas para o Índice Bitmap, tais como *binning*, compressão e codificação; (iii) adiciona as vantagens tradicionais já proporcionadas pelo Índice Bitmap ao DWET, tal como o suporte eficiente à multidimensionalidade; e (iv) otimiza o acesso a DWET com tabelas de dimensão espacial possuindo atributos espaciais organizados em uma hierarquia predefinida e cujas geometrias mudam ao longo do tempo. Nas seções seguintes, são descritos a estrutura de dados, o procedimento de construção do índice proposto e o processamento de consultas do STB-index.

4.1.1 Estrutura de Dados

Cada entrada `e` do STB-index possui a forma <`e.chave`, `e.MBR`, `e.tval`>, em que `e.chave` corresponde à chave primária da tabela de dimensão espaço-temporal e identifica um objeto `x`, `e.MBR` armazena o MBR de `x` e `e.tval` contém o tempo inicial e o tempo final referente ao período em que o objeto `x` é válido no mundo real. O tempo inicial e o tempo final estão convertidos para o *Unix Epoch*, ou seja, número de segundos a partir de 1 de Janeiro de 1970.

Considerando o conjunto de dados apresentado na Figura 4.1, em que o atributo `lo_suppkey` da tabela de fatos `Lineorder` referencia `s_suppkey` e o atributo `s_city_fk` da tabela de dimensão `Supplier` referencia `city_pk`, temos que `city_pk` = 1 está associado à `s_suppkey` = 2 e `lo_suppkey` = 2. Portanto, na entrada `[0]` do STB-index e_0`.chave` = 1, e_0`.MBR` = {15.35; 251.80; 35.62; 259.11} corresponde ao MBR do objeto que possui `city_pk` = 1 e e_0`.tval` = {694234800; 838609199} corresponde ao tempo de validade inicial e final do mesmo objeto, como mostrado na Figura 4.2. Ainda na mesma figura, `B` corresponde ao IBJ definido sobre a coluna `city_pk` da tabela de dimensão `City`. O vetor de bits apontado por `B[0]` especifica as tuplas da tabela de fatos em que `city_pk` = 1. Dessa forma, acessar o vetor de bits relacionado à entrada `[0]` do STB-index requer apenas o uso do endereço 0 (zero) para acessar `B`, ou seja, `B[0]`.

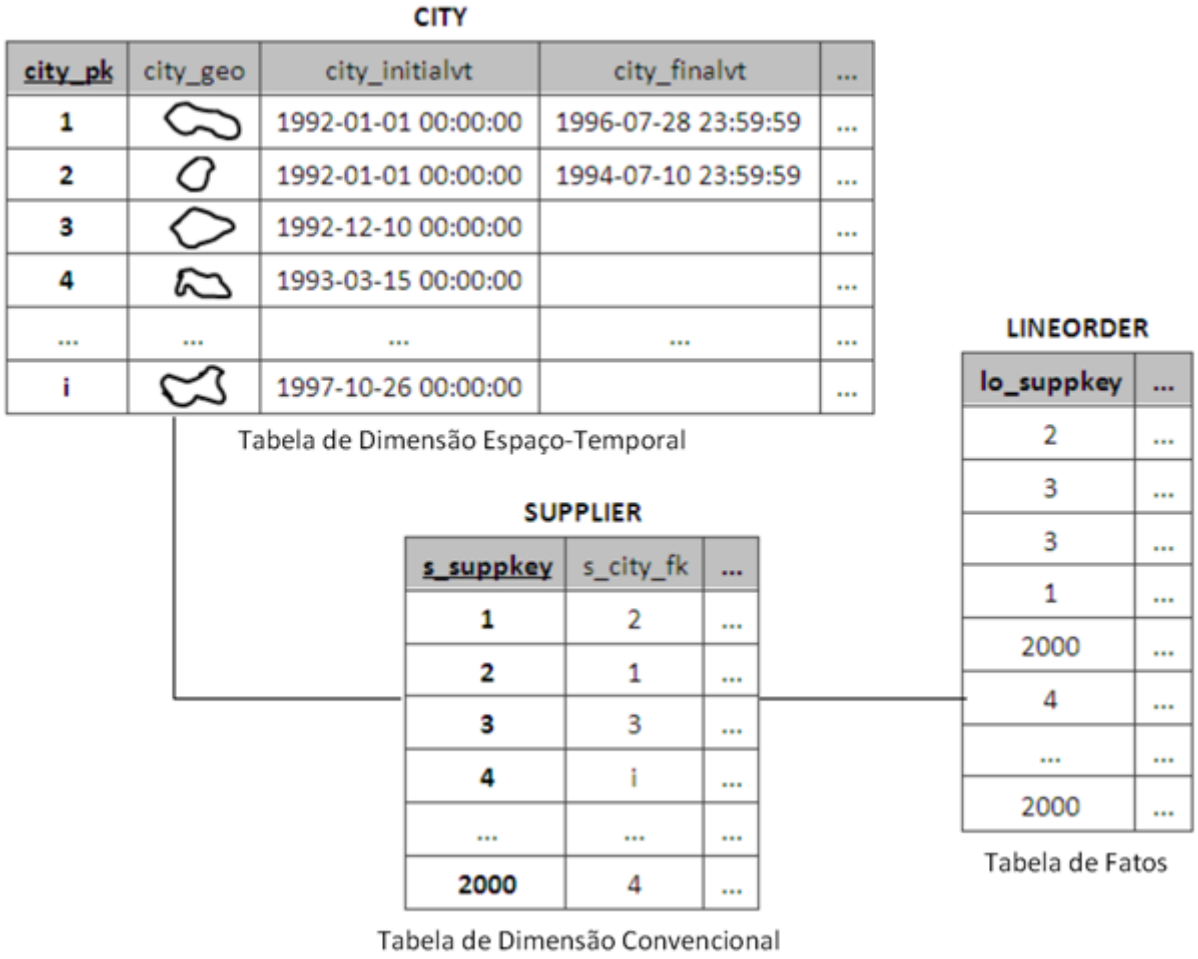


Figura 4.1. Exemplo de dados para um DWET.

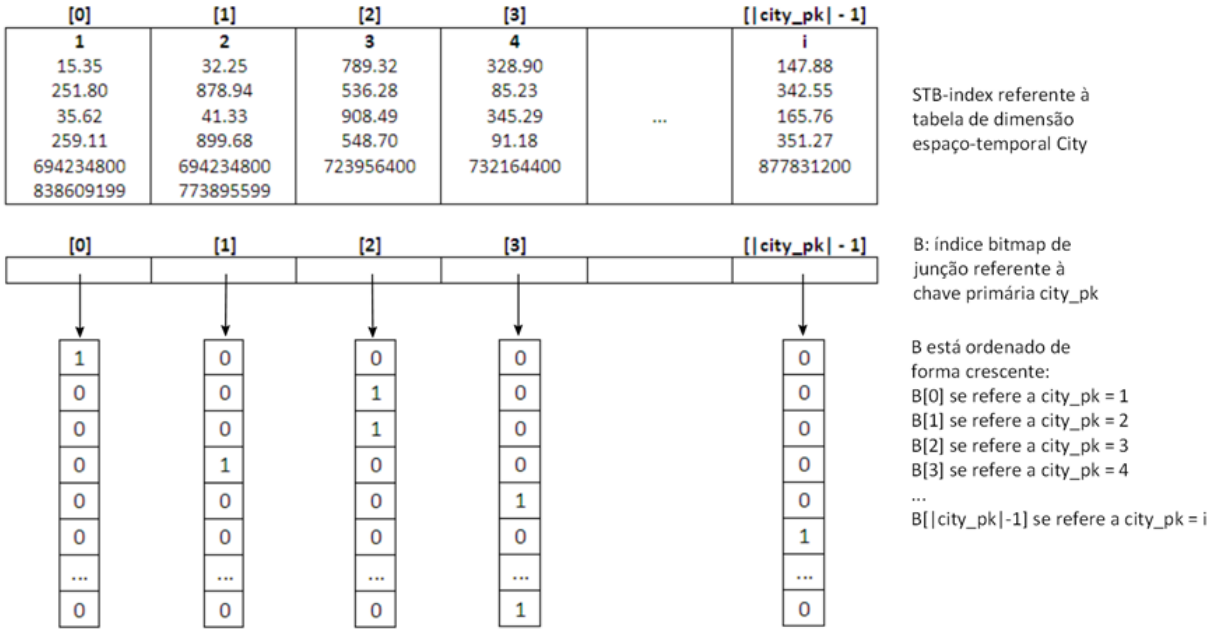


Figura 4.2. Estrutura de dados do STB-index.

4.1.2 Construção da Estrutura de Indexação

O algoritmo de construção do STB-index é apresentado na Tabela 4.1. Sejam:

- `T`: tabela de dimensão espaço-temporal;
- `pk`: chave primária da tabela `T`;
- `geo`: geometria do objeto espaço-temporal;
- `tv_inicial`: tempo válido inicial do objeto espaço-temporal `geo`;
- `tv_final`: tempo válido final do objeto espaço-temporal `geo`;
- `arquivo_idx`: arquivo do STB-index;
- `tam_cabecalho`: tamanho do cabeçalho do arquivo do STB-index;
- `tam_pagina`: tamanho da página de disco;
- `quant_max`: quantidade máxima de entradas por página de disco.

Tabela 4.1. Algoritmo de construção do STB-index.

```
Construir (T, pk, geo, tv_inicial, tv_final, arquivo_idx,
tam_cabecalho, tam_pagina, quant_max)
Saída: arquivo do STB-index construído e armazenado em disco.
Variáveis: resultado, buffer, cardinalidade, i
Início
01   resultado ← executar_comando_sgbd (SELECT pk, x_min(geo),
02   y_min(geo), x_max(geo), y_max(geo), tv_inicial, tv_final FROM T
03   ORDER BY pk)
04
05   abrir (arquivo_idx)
06   posicionar (arquivo_idx, tam_cabecalho)
07
08   i ← 0
09   cardinalidade ← 0
10   enquanto (existem_tuplas_em (resultado)) faça
11       buffer[i].pk ← resultado.pk
12       buffer[i].x_min ← resultado.x_min
13       buffer[i].y_min ← resultado.y_min
14       buffer[i].x_max ← resultado.x_max
15       buffer[i].y_max ← resultado.y_max
16       buffer[i].tv_inicial ← resultado.tv_inicial
17       buffer[i].tv_final ← resultado.tv_final
18       i ← i + 1
19       proxima_tupla (resultado)
20
21       se (i ≥ quant_max) então
22           escrever (buffer, arquivo_idx, tam_pagina)
23           cardinalidade ← cardinalidade + i
24           i ← 0
25       fim-se
26   fim-enquanto
27
28   escrever (buffer, arquivo_idx, tam_pagina)
29   cardinalidade ← cardinalidade + i
30
```

```
31    posicionar (arquivo_idx, 0)
32    escrever (cardinalidade, arquivo_idx, tam_int)
33    fechar (arquivo_idx)
Fim
```

Na Tabela 4.1, a linha 01 consiste em executar uma consulta no SGBD (`executar_comando_sgbd`) para obter a chave primária da tabela de dimensão espaço-temporal, os pares de coordenadas (`x_min, y_min`) e (`x_max, y_max`) do objeto espaço-temporal `geo` e os tempos de validade inicial e final do objeto espaço-temporal. O resultado da consulta é um conjunto de registros que podem ser acessados pelo ponteiro `resultado`.

Em seguida, após aberto o arquivo do STB-index e posicionado para que a escrita ocorra após o cabeçalho (linhas 05 e 06), inicia-se uma repetição preenchendo um *buffer* com dados resultantes da consulta submetida (linhas 10 a 26). Cada vez que o *buffer* atinge o tamanho de uma página de disco, seu conteúdo é gravado no arquivo do STB-index (linhas 21 a 25).

Após todos os registros resultantes da consulta serem lidos, a repetição termina e os dados que ficaram no *buffer* são gravados no arquivo do STB-index (linha 28).

Nas linhas 31 a 33, a cardinalidade, que armazena a quantidade de registros retornados pela consulta e gravados no arquivo do STB-index, é gravada no cabeçalho do arquivo e este é fechado.

4.1.3 Processamento de Consultas

O processamento de consultas OLAP no STB-index ocorre de forma similar ao realizado pelo SB-index, uma vez que o objetivo é transformar o predicado espaço-temporal em um predicado convencional e depois utilizar o IBJ para obter a resposta final da consulta.

O algoritmo do processamento de consultas usando o STB-index é apresentado na Tabela 4.2. Sejam:

- `arquivo_idx`: arquivo do STB-index;
- `tam_cabecalho`: tamanho do cabeçalho do arquivo do STB-index;
- `tam_pagina`: tamanho da página de disco;
- `quant_max`: quantidade máxima de entradas por página de disco;

- `cardinalidade`: quantidade de entradas do STB-index;
- `temp_val`: tempos de validade inicial e final fornecidos na consulta;
- `jc`: janela de consulta espacial;
- `T`: tabela de dimensão espaço-temporal;
- `pk`: chave primária da tabela `T`;
- `geo`: geometria do objeto espaço-temporal;
- `predic_conv`: *string* que conterá o predicado convencional resultante do processamento de consultas do STB-index;
- `consulta`: *string* da consulta sem o predicado espaço-temporal;
- `resposta`: resposta da consulta.

Tabela 4.2. Algoritmo do processamento de consultas usando o STB-index.

```
Processar_Consulta (arquivo_idx, tam_cabecalho, tam_pagina,
quant_max, cardinalidade, temp_val, jc, T, pk, geo, predic_conv,
consulta, resposta)
Saída: resposta da consulta.
Variáveis: pagina, buffer, num_candidatos, posicao, i, j
Início
01    abrir (arquivo_idx)
02    num_candidatos ← 0
03    posicao ← tam_cabecalho
04    enquanto (j < cardinalidade) faça
05        posicionar (arquivo_idx, posicao)
06        ler (pagina, arquivo_idx, tam_pagina)
07        copiar (buffer, pagina, tam_pagina)
08
09        para (i ← 0; i ≤ quant_max && j < cardinalidade; i ← i + 1,
10    j ← j + 1) faça
11            se (RT (buffer[i], temp_val)) então
12                se (RE (buffer[i], jc) então
13                    candidatos[num_candidatos] ← buffer[i].pk
14                    num_candidatos ← num_candidatos + 1
15                fim-se
16            fim-se
17        fim-para
18
19        posicao ← posicao + tam_pagina
20    fim-enquanto
21    fechar (arquivo_idx)
22
23    para (i ← 0; i ≤ num_candidatos; i ← i + 1) faça
24        se (executar_comando_sgbd (SELECT R(geo, jc) FROM T WHERE
25    pk = candidatos[i])) então
26            concatenar (candidatos[i], predic_conv)
27        fim-se
28    fim-para
29
30    concatenar (predic_conv, consulta)
31    resposta ← executar_fastbit (consulta)
Fim
```

Na primeira etapa, faz-se a busca no arquivo do STB-index. Cada página de disco é lida e copiada para a memória primária (linhas 05 a 07), sendo feita a verificação do predicado espaço-temporal, ou seja, para cada entrada, verifica-se primeiramente o tempo de validade do objeto (linha 11) e, caso ele seja válido no tempo fornecido pela consulta, é depois verificado se o MBR satisfaz às condições espaciais (linha 12). Se as condições temporais e espaciais da consulta forem satisfeitas, significa que o objeto espaço-temporal é um candidato ao conjunto resposta da consulta e sua respectiva chave primária é coletada ao conjunto de candidatos (linha 13).

Após verificar sequencialmente todas as entradas em cada página de disco do STB-index (linhas 04 a 20), é realizada a etapa de refinamento. Com as chaves primárias coletadas na primeira etapa, a tabela de dimensão espaço-temporal é acessada e então a geometria exata de cada objeto espaço-temporal é verificada de acordo com as condições espaciais (linha 24) e, caso satisfaça essas condições, sua chave primária é utilizada para compor um predicado convencional (linha 26) que será concatenado aos demais predicados convencionais da consulta original (linha 30).

Uma vez que a consulta fica somente com predicados convencionais após a realização das etapas anteriores do processamento de consultas, utiliza-se o IBJ para obter o resultado final da consulta (linha 31).

Capítulo 5

EXPERIMENTOS

O objetivo deste capítulo é validar a proposta do STB-index com relação ao estado da arte na indexação espaço-temporal em data warehouse, comparando o desempenho do STB-index com o desempenho de recursos de sistemas gerenciadores de banco de nados, tais como visões materializadas e junção estrela. Portanto, neste capítulo são descritos os experimentos realizados e discutidos os resultados de desempenho coletados. Especificamente, são detalhados a plataforma computacional utilizada nos testes de desempenho, os dados que foram utilizados nos testes de desempenho e como foram obtidos estes dados e também as consultas OLAP que foram investigadas. Em seguida, são descritos cada teste realizado e os seus resultados.

5.1 Plataforma Computacional

Os testes de desempenho foram executados em um servidor de banco de dados com processador Intel Core i7 2,67 GHz, 12 GB de RAM, dois discos rígidos de 1 TB, dois discos rígidos de 2 TB e com o sistema operacional Ubuntu 11.10.

Os programas utilizados foram:

- PostgreSQL 9.1.1 – SGBD objeto-relacional de código aberto;
- PostGIS 1.5.3 – extensão espacial para o PostgreSQL de código aberto. Permite o armazenamento e o gerenciamento de dados espaciais e inclui suporte aos índices espaciais GiST e R-tree;
- FastBit 1.2.4 – software de código aberto que implementa as funcionalidades do índice bitmap.

5.2 Dados

O esquema de dados lógico (i.e. relacional) utilizado para os testes de desempenho do STB-index (Figura 5.1) foi obtido por meio da adaptação do esquema do Spatial SSB. Ele contém uma tabela de fatos central (`Lineorder`), duas tabelas de dimensão convencional (`Part` e `Date`) e tabelas de dimensão espaço-temporal (`Customer`, `Supplier`, `Street`, `City`, `Nation` e `Region`).

A tabela de fatos e as tabelas de dimensão convencional permanecem as mesmas do esquema do Spatial SSB. Nas demais tabelas foram acrescentados o tempo de validade inicial e o tempo de validade final, para controlar a alteração tanto das geometrias quanto dos atributos convencionais. Além disso, nas tabelas de `Street`, `City`, `Nation` e `Region` foram adicionados os campos "`_operation`" e "`_pk_oldnew`" para armazenar o histórico das alterações nas geometrias. O campo "`_operation`" armazena a operação de modificação realizada, que pode ser de:

- Alteração: mudança de endereço no caso de pontos, redução do tamanho de ruas no caso de linhas, redução da área de cidades, nações e regiões no caso de polígonos.
- Divisão: criação de duas ruas a partir de uma rua ou criação de duas cidades a partir de uma cidade.
- União: junção de duas ruas para formar uma nova rua ou junção de duas cidades para formar uma nova cidade.

O campo "`_pk_oldnew`" relacionado ao objeto `x` armazena a chave primária do objeto que deu origem à `x` ou o objeto ao qual `x` dará origem, dependendo do que está armazenado no campo "`_operation`". No caso de uma operação de alteração ou divisão, o campo "`_pk_oldnew`" do novo objeto armazena a chave primária do objeto que deu origem a ele. No caso de uma operação de união, o campo "`_pk_oldnew`" dos objetos antigos, que se uniram para formar o novo objeto, armazena a chave primária do novo objeto.

Os dados para a realização dos testes de desempenho são formados por dados sintéticos e foram gerados seguindo o esquema apresentado na Figura 5.1. Para as tabelas de dimensão convencional e a tabela de fatos, foi utilizado o gerador

de dados do SSB. Para as tabelas de dimensão espaço-temporal, utilizou-se o gerador de dados do Spatial SSB denominado *Spatial Geometry Generator* para a geração de dados espaciais, que foram posteriormente modificados para dados espaço-temporais.

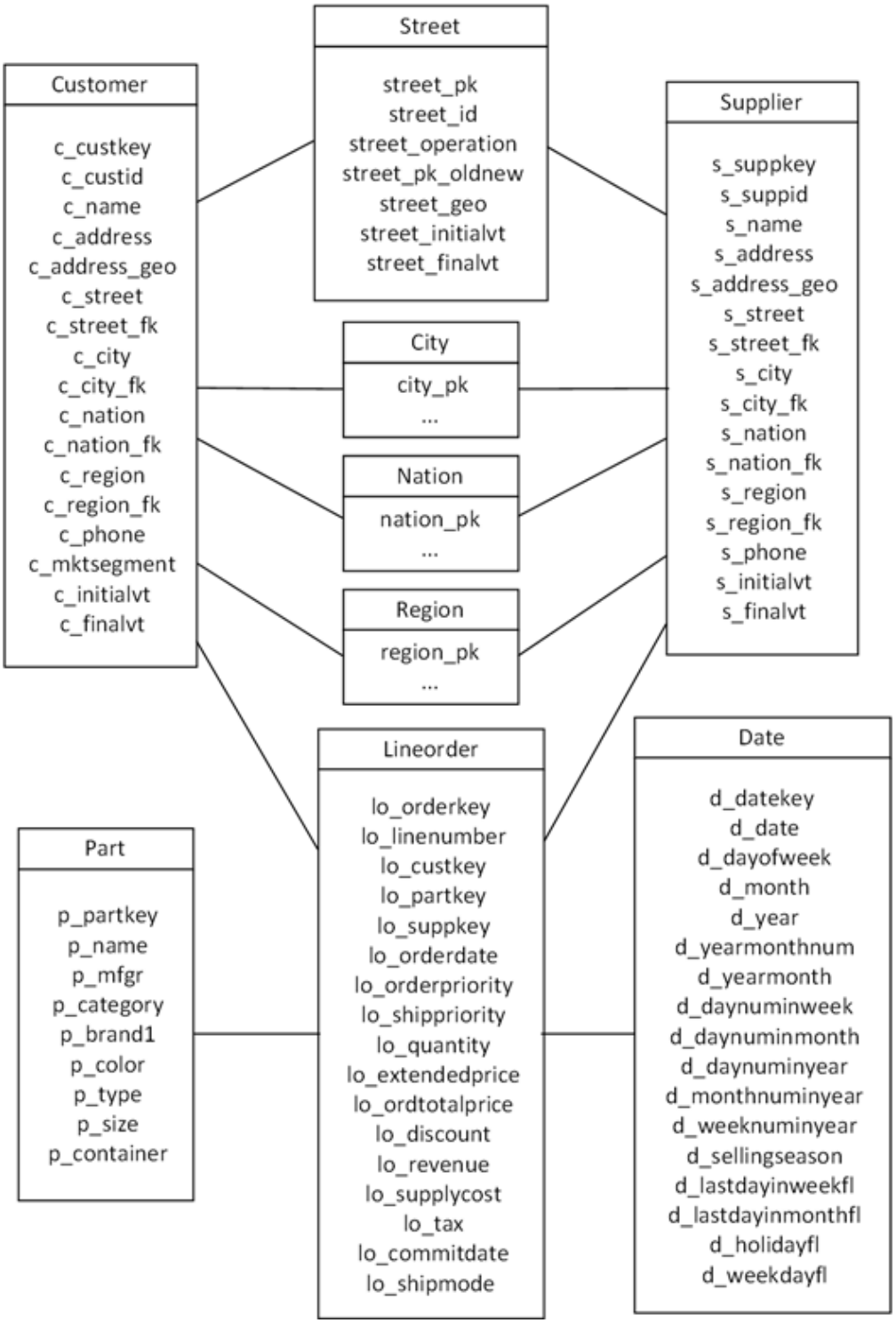


Figura 5.1. Esquema do DWET.

Após a geração dos dados, as tabelas foram carregadas no banco de dados e foram feitas as modificações nas geometrias dos endereços, ruas e cidades, utilizando as operações citadas anteriormente: alteração, divisão e união. As modificações foram feitas da seguinte forma:

1. Primeiramente, 5% dos endereços de fornecedores (`Supplier`) foram selecionados aleatoriamente e aplicou-se a operação de alteração, sendo atribuído um novo endereço a cada um;
2. Em seguida, 5% dos endereços de clientes (`Customer`) foram selecionados aleatoriamente e aplicou-se a operação de alteração, sendo atribuído um novo endereço a cada um;
3. Após a modificação dos endereços, 5% das ruas foram selecionadas aleatoriamente e foram aplicadas as operações de alteração, divisão e união, sendo que, das ruas selecionadas (5%), 1/3 sofreu a operação de alteração, 1/3 sofreu a operação de divisão e 1/3 sofreu a operação de união. Como a modificação das ruas pode ocasionar a modificação dos endereços, após cada operação de modificação das ruas foi feita a modificação dos endereços de clientes e fornecedores, conforme o necessário. Ex: se uma rua `z` é dividida e duas novas ruas `x` e `y` são criadas, é necessário atualizar a rua dos clientes e fornecedores que moravam na rua `z` para `x` ou `y`. Além disso, a redução no tamanho de uma rua pode fazer com que endereços deixem de existir.
4. Por último, foi feita a modificação das cidades. 5% das cidades foram selecionadas aleatoriamente e foram aplicadas as operações de alteração, divisão e união, sendo que 1/3 das cidades selecionadas sofreram alteração, 1/3 delas sofreram divisão e 1/3 sofreram união. Como a modificação das cidades pode ocasionar a modificação de endereços, ruas, nações e regiões, após cada operação de modificação das cidades, foi feita a modificação dos demais objetos espaciais afetados. Ex: a redução na área de uma cidade pode levar à redução de ruas que a compõem e à redução da nação e da região correspondente, além de fazer com que endereços e ruas deixem de existir.
5. As modificações de 1 a 4 foram realizadas novamente, alterando-se apenas a data da modificação. Ex: a primeira alteração dos endereços de fornecedores

ocorreu em janeiro de 1993 e a segunda alteração ocorreu em janeiro de 1996.

A Figura 5.2 apresenta a quantidade de tuplas resultante em cada tabela do DWET após a modificação das geometrias.

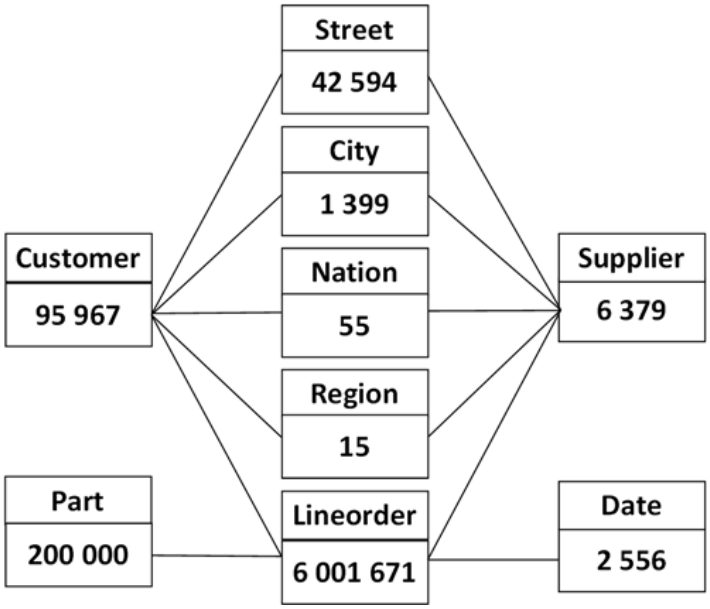


Figura 5.2. Quantidade de tuplas em casa tabela.

Para mais detalhes, o código para a modificação dos objetos espaciais foi colocado no Apêndice A.

5.3 Consultas OLAP

Tendo em vista que este trabalho está concentrado em prover melhor desempenho no processamento de consultas OLAP sobre um DWET com objetos que se modificam ao longo do tempo, os critérios para a escolha da consulta foram: (i) ter pelo menos um predicado espaço-temporal e (ii) envolver as tabelas de `Customer` e `Supplier`, pois são elas que possuem referência para as demais tabelas de dimensão espaço-temporal (`Street`, `City`, `Nation` e `Region`). Considerando esses critérios, elaborou-se uma consulta baseada na consulta Q3 do SSB (O'NEIL, P. E. et al., 2009), que é discutida a seguir.

A Figura 5.3 mostra a adaptação da consulta Q3, em que os predicados convencionais `c_region='ASIA'` e `s_region='ASIA'` foram substituídos por

predicados espaço-temporais. Os níveis explorados na consulta foram os níveis de endereços, ruas e cidades, representados por pontos, linhas e polígonos, respectivamente. Eles foram considerados porque nesses níveis foram introduzidas as modificações na geometria dos objetos. Além disso, para cada tipo de objeto espacial foi avaliado um predicado espacial: para os endereços, o predicado espacial testado foi contém (*contains*); para as ruas e cidades, utilizou-se o predicado intersecta (*intersects*). Além disso, para cada objeto foi necessário verificar se ele era válido no tempo especificado.

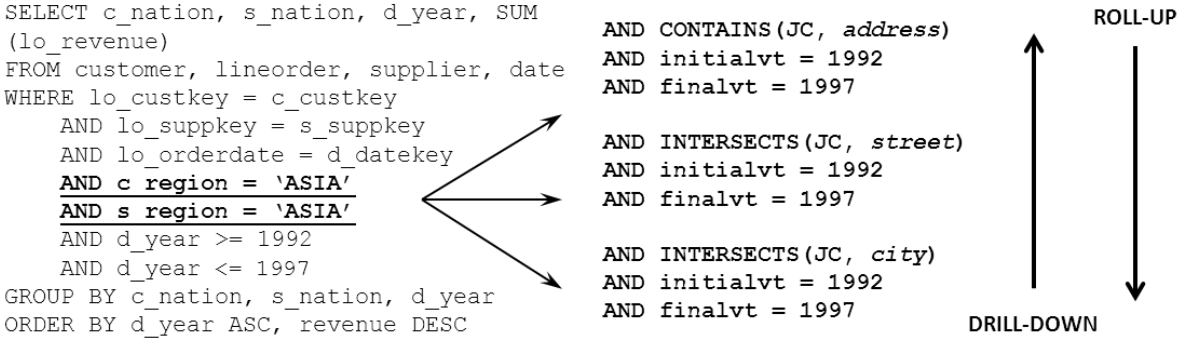


Figura 5.3. Adaptação da consulta Q3 do SSB.

As janelas de consulta (JC) foram selecionadas considerando a distribuição dos objetos no espaço, dando preferência pela sobreposição de objetos que se modificaram ao longo do tempo. A Tabela 5.1 mostra a fração do *extent* sobreposta por cada JC. Para os endereços de clientes e fornecedores foi selecionada apenas uma JC, pois a distribuição dos pontos no espaço não permitiu a seleção de JCs menores. Para as ruas e cidades foram selecionadas quatro JCs de tamanhos crescentes.

Tabela 5.1. Fração do *extent* sobreposta por cada JC.

	Endereços	**Ruas**	**Cidades**
Tamanho da JC / Tamanho do *extent*	0,050%	0,005%	0,010%
		0,010%	0,050%
		0,050%	0,100%
		0,100%	1,000%

5.4 Configurações dos Testes de Desempenho e Resultados

Em DW, o processamento de consultas OLAP é comumente efetuado usando as técnicas de junção estrela e visões materializadas (SIQUEIRA et al., 2012). Assim, essas duas técnicas foram utilizadas como base de comparação de desempenho na realização de consultas OLAP resultando em três configurações de testes:

1. Junção estrela auxiliada pelo índice espacial GiST sobre os atributos espaciais;
2. Visão materializada auxiliada pelos índices B-tree sobre o atributo `year` e GiST sobre os atributos espaciais;
3. STB-index.

A Tabela 5.2 apresenta as medidas coletadas para a construção do STB-index. O índice bitmap de junção (IBJ) construído utilizando o FastBit ocupou 901 MB considerando a indexação de todos os atributos convencionais necessários para o processamento das consultas definidas na Seção 5.3. É possível perceber que o aumento nos requisitos de armazenamento não chega a 0,5% se comparado com o IBJ.

Tabela 5.2. Medidas obtidas para construção do STB-index.

	Tempo decorrido	Número de acessos a disco	Tamanho do STB-index	Acréscimo em relação ao IBJ
Customer	40 s	517	4132 kB	0,444 %
Supplier	3 s	36	284 kB	0,031 %
Street	19 s	231	1844 kB	0,198 %
City	1 s	9	68 kB	0,007 %

A Tabela 5.3 e a Figura 5.4 mostram os resultados obtidos na execução das consultas apresentadas na Seção 5.3. Todas as consultas foram executadas 10 vezes e o resultado apresentado refere-se à média de tempo de resposta das consultas. Pode-se observar que, para todos os níveis de granularidade, o tempo de resposta do STB-index é indiscutivelmente melhor do que o tempo das visões materializadas, que, por sua vez, são melhores do que efetuar operações de junção estrela.

Tabela 5.3. Medidas coletadas no processamento das consultas para os três níveis de granularidade.

	Junção Estrela (ms)	VM (ms)	STB-index (ms)
Address	9402,537	5934,147	46,160
Street	12703,433	9635,285	151,219
City	9350,373	6429,551	121,380

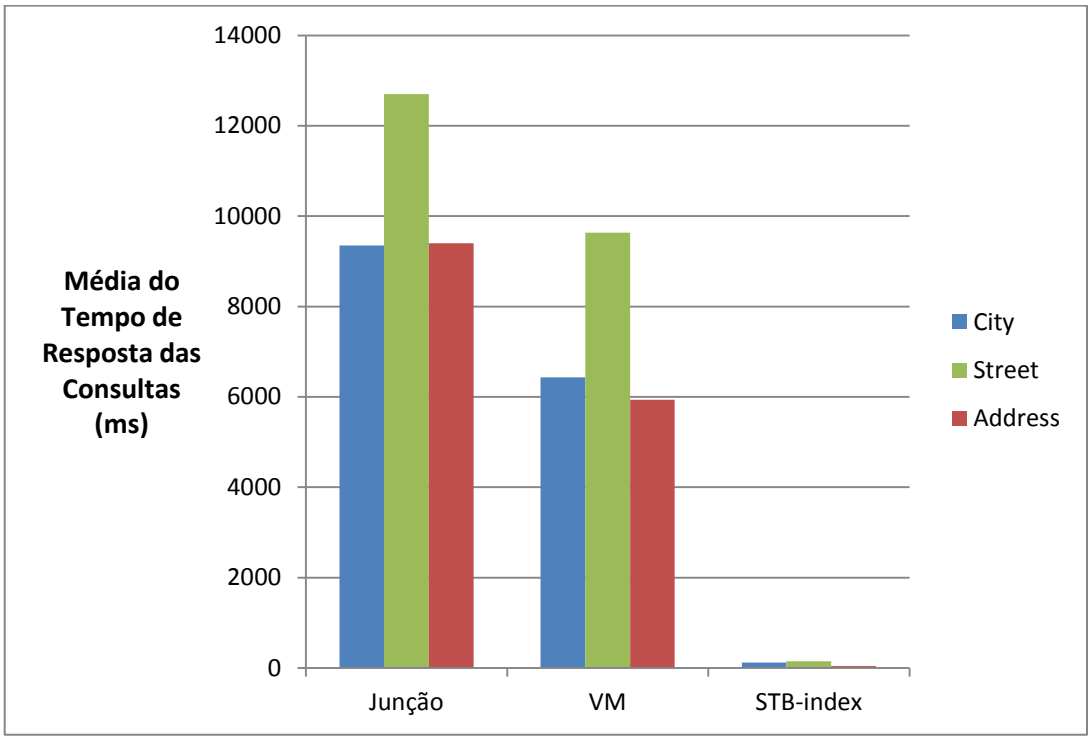


Figura 5.4. Comparação do tempo de resposta para os três níveis de granularidade utilizando junção estrela, visão materializada e STB-index.

A Tabela 5.4 apresenta a redução de tempo do STB-index em relação à junção estrela e à visão materializada e desta em relação à junção estrela. A redução de tempo do STB-index chega a 99,22% em relação à visão materializada no nível de endereço.

Tabela 5.4. Redução de tempo.

	STB-index x Junção estrela	STB-index x VM	VM x Junção estrela
Address	99,51%	99,22%	36,89%
Street	98,81%	98,43%	24,15%
City	98,70%	98,12%	30,97%

Capítulo 6

CONCLUSÕES E TRABALHOS FUTUROS

Este capítulo apresenta as considerações finais sobre o trabalho de pesquisa realizado, exibindo as suas principais contribuições e listando as indicações de trabalhos futuros.

6.1 Conclusões

Neste trabalho de pesquisa foi proposta a estrutura de indexação STB-index para DWET considerando dados espaciais que se modificam ao longo do tempo. Os dados espaciais podem sofrer as seguintes operações que modificam a sua geometria: alteração, união e divisão.

O STB-index é uma extensão do SB-index e foi projetado com o objetivo de reduzir o tempo de resposta de consultas em DWET com dados espaço-temporais de três tipos: pontos, linhas e polígonos, sendo que os dados formam hierarquias de atributo espaço-temporais, como exemplo, $\texttt{region} \preceq \texttt{nation} \preceq \texttt{city} \preceq \texttt{street} \preceq \texttt{address}$.

Os experimentos realizados para validar a proposta do STB-index e compará-lo com os recursos de junção estrela e visão materializada fornecidos pelos SGBD foram concentrados em consultas OLAP do tipo *drill-down* e *roll-up* com predicados espaciais do tipo intersecta, para linhas e polígonos, e do tipo contém, para pontos.

De acordo com os testes realizados, o STB-index apresentou resultados bastante relevantes. Além de não requerer muito espaço de armazenamento, adicionando no máximo 0,5% aos custos de armazenamento do IBJ, o STB-index

também reduziu o tempo de resposta das consultas OLAP entre 98,12% e 99,22% se comparado com as visões materializadas, o que comprova a eficiência do índice proposto.

6.2 Trabalhos Futuros

Nas subseções seguintes são sugeridos trabalhos futuros para a continuação deste trabalho de pesquisa. Além deles, há também:

- Realizar testes de desempenho utilizando outros sistemas gerenciadores de banco de dados.
- Realizar testes de desempenho para investigar o impacto do tamanho da janela de consulta, do intervalo de tempo e granularidade utilizado no predicado temporal da consulta OLAP, dos tipos de operações (*drill-down* e *roll-up*) e do volume de operações no desempenho do STB-index.

6.2.1 Alteração da Estrutura Sequencial do STB-index

Atualmente, o STB-index está implementado como um vetor em disco magnético (i.e. arquivo sequencial). Por isso, todas as entradas do índice sempre serão verificadas no processamento de qualquer consulta OLAP. Além disso, não há uma forma eficiente para reaproveitar as entradas do STB-index acessadas em consultas anteriores, a fim de evitar novos acessos a disco. Dessa forma, é possível estender o STB-index para uma estrutura hierárquica de árvore com o auxílio de um *buffer-pool*, de forma similar ao que foi feito para o HSB-index, para analisar se há melhora no desempenho de consultas OLAP.

6.2.2 Variantes do STB-index

Durante a fase de concepção do STB-index, algumas variantes foram preliminarmente propostas conceitualmente. Um trabalho futuro consiste em

implementar cada uma dessas variantes e realizar testes de desempenho com as mesmas. Abaixo estão descritas cada uma delas:

- A primeira variante consiste em armazenar o tempo válido inicial e final do objeto espacial em uma B+-tree de forma a processar o predicado temporal da consulta OLAP antes do predicado espacial. Os candidatos que satisfizerem o predicado temporal são passados como entrada para o SB-index, que irá filtrar os candidatos pelo predicado espacial e transformá-lo em predicado convencional.
- A segunda variante consiste em criar índices bitmap sobre o tempo válido inicial e final dos objetos espaciais utilizando a técnica de *binning* para filtrar o predicado temporal da consulta OLAP antes do predicado espacial. Assim como na proposta anterior, os candidatos que satisfizerem o predicado temporal são passados como entrada para o SB-index, que irá filtrar os candidatos pelo predicado espacial e transformá-lo em predicado convencional.
- A última proposta preliminar consiste em utilizar a primeira variante, mas filtrando o predicado espacial da consulta OLAP antes do predicado temporal. Ou seja, o SB-index é utilizado para filtrar pelo predicado espacial e os candidatos que satisfizerem esse predicado são passados para a B+-tree, que irá filtrar pelo predicado temporal e transformar em predicado convencional.

6.2.3 Benchmark para Data Warehouse Espaço-Temporal

Não foi encontrado na literatura um benchmark para DWET para a realização de testes de desempenho com o STB-index a fim de validá-lo. Isso motivou a criação de um esquema baseado no Spatial SSB para prover suporte a geometrias que evoluem ao longo do tempo. A partir do esquema criado, dados foram gerados com o apoio de ferramentas próprias para DW (gerador de dados do SSB) e DWE (*Spatial Geometry Generator*). A formalização e especificação de consultas para o DWET e a extensão da forma de geração dos dados para o mesmo podem ser feitas a fim de estabelecer um novo benchmark para a análise e validação de trabalhos que envolvam DWET. Em adição, pode-se criar uma ferramenta específica para geração de dados para DWET.

Referências

BARBOSA, D.; MANOLESCU, I.; YU, J. X. Application Benchmark. In: LIU, L.; ÖZSU, M. T. (Eds.). **Encyclopedia of Database Systems**. Springer, 2009. v. 1, p. 99-100.

BECKMANN, N.; KRIEGEL, H.-P.; SCHNEIDER, R.; SEEGER, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. **SIGMOD Record**, New York, NY, v. 19, n. 2, p. 322-331, jun. 1990.

BIMONTE, S.; TCHOUNIKINE, A.; MIQUEL, M. Towards a spatial multidimensional model. In: INTERNATIONAL WORKSHOP ON DATA WAREHOUSING AND OLAP (DOLAP), 8th, 2005, Bremen, Germany. **Proceedings...** New York, NY: ACM, 2005. p. 39-46.

BRINKHOFF, T.; KRIEGEL, H.-P.; SCHNEIDER, R.; SEEGER, B. Multi-Step Processing of Spatial Joins. **SIGMOD Record**, New York, NY, v. 23, n. 2, p. 197-208, jun. 1994.

CASTRO, C. V. R. **CSTM**: A Conceptual Spatiotemporal Model for Data Warehouses. 2010. 117 p. Master's Thesis (Masters in Computer Science) - Informatics Center, Federal University of Pernambuco, Recife. 2010.

CIFERRI, C. D. A. **Distribuição dos dados em ambientes de data warehousing**: o sistema WebD2W e algoritmos voltados à fragmentação horizontal dos dados. 2002. 263 f. Tese (Doutorado em Ciência da Computação) - Centro de Informática, Universidade Federal de Pernambuco, Recife. 2002.

CIFERRI, R. R. **Análise da influência do fator distribuição espacial dos dados no desempenho de métodos de acesso multidimensionais**. 2002. 246 f. Tese (Doutorado em Ciência da Computação) - Centro de Informática, Universidade Federal de Pernambuco, Recife. 2002.

CÂMARA, G.; CASANOVA, M. A.; HEMERLY, A. S.; MAGALHÃES, G. C.; MEDEIROS, C. M. B. **Anatomia de Sistemas de Informação Geográfica**. São José dos Campos, SP: INPE, 1996. 205 p.

DYRESON, C.; GRANDI, F.; KÄFER, W. et al. A consensus glossary of temporal database concepts. **SIGMOD Record**, New York, NY, v. 23, n. 1, p. 52-64, mar. 1994.

EDER, J.; KONCILIA, C.; MORZY, T. A Model for a Temporal Data Warehouse. In: OPEN ENTERPRISE SOLUTIONS: SYSTEMS, EXPERIENCES, AND ORGANIZATIONS (OES-SEO), 2001, Rome, Italy. **Proceedings...** 2001. p. 86-97.

EDER, J.; KONCILIA, C.; MORZY, T. The COMET Metamodel for Temporal Data Warehouses. In: INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION SYSTEMS ENGINEERING (CAiSE), 14th, 2002, Toronto, Canada. **Proceedings...** Berlin / Heidelberg: Springer, 2006. p.83-99. (Lecture Notes in Computer Science, v. 2348).

FOLK, M. J.; ZOELLICK, B. **File Structures**: A Conceptual Toolkit. 2nd ed. Boston, MA: Addison-Wesley, 1991. 590 p.

GAEDE, V.; GÜNTHER, O. Multidimensional access methods. **ACM Computing Surveys (CSUR)**, New York, NY, v. 30, n. 2, p. 170-231, jun. 1998.

GARCIA-MOLINA, H.; WIDOM, J.; ULLMAN, J. D. **Database Systems**: The Complete Book. 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2008. 1248 p.

GOLFARELLI, M.; RIZZI, S. A Survey on Temporal Data Warehousing. **International Journal of Data Warehousing and Mining (IJDWM)**, v. 5, n. 1, p. 1-17, 2009.

GUTTMAN, A. R-trees: a dynamic index structure for spatial searching. **SIGMOD Record**, New York, NY, v. 14, n. 2, p. 47-57, jun. 1984.

GÓMEZ, L.; KUIJPERS, B.; MOELANS, B.; VAISMAN, A. A Survey of Spatio-Temporal Data Warehousing. **International Journal of Data Warehousing and Mining (IJDWM)**, v. 5, n. 3, p. 28-55, 2009.

GÜTING, R. H. An introduction to spatial database systems. **The VLDB Journal**, Secaucus, NJ, v. 3, n. 4, p. 357-399, oct. 1994.

HARINARAYAN, V.; RAJARAMAN, A.; ULLMAN, J. D. Implementing data cubes efficiently. **SIGMOD Record**, New York, NY, v. 25, n. 2, p. 205-216, jun. 1996.

INMON, W. H. **Building the Data Warehouse**. 4th ed. Indianapolis, IN: Wiley, 2005. 576 p.

KIMBALL, R.; ROSS, M. **The Data Warehouse Toolkit**: The Complete Guide to Dimensional Modeling. 2nd ed. New York, NY: Wiley, 2002. 464 p.

MALINOWSKI, E.; ZIMÁNYI, E. Spatial Hierarchies and Topological Relationships in the Spatial MultiDimER Model. In: BRITISH NATIONAL CONFERENCE ON DATABASES: ENTERPRISE, SKILLS AND INNOVATION (BNCOD), 22nd, 2005, Sunderland, UK. **Proceedings...** Berlin / Heidelberg: Springer, 2005. p. 181-194. (Lecture Notes in Computer Science, v. 3567).

MALINOWSKI, E.; ZIMÁNYI, E. Requirements Specification and Conceptual Modeling for Spatial Data Warehouses. In: INTERNATIONAL CONFERENCE ON ON THE MOVE TO MEANINGFUL INTERNET SYSTEMS (OTM), 2006, Montpellier,

France. **Proceedings...** Berlin / Heidelberg: Springer, 2006a. p. 1616-1625. (Lecture Notes in Computer Science, v. 4278).

MALINOWSKI, E.; ZIMÁNYI, E. A conceptual solution for representing time in data warehouse dimensions. In: ASIA-PACIFIC CONFERENCE ON CONCEPTUAL MODELLING (APCCM), 3rd, 2006, Hobart, Australia. **Proceedings...** Darlinghurst, Australia: ACS, 2006b. p. 45-54. (CRPIT, v. 53).

MALINOWSKI, E.; ZIMÁNYI, E. **Advanced Data Warehouse Design**: From Conventional to Spatial and Temporal Applications. Berlin / Heidelberg: Springer, 2008a. 444 p. (Data-Centric Systems and Applications).

MALINOWSKI, E.; ZIMÁNYI, E. A conceptual model for temporal data warehouses and its transformation to the ER and the object-relational models. **Data & Knowledge Engineering**, Amsterdam, The Netherlands, v. 64, n. 1, p. 101-133, jan. 2008b.

MANOLOPOULOS, Y.; THEODORIDIS, Y.; TSOTRAS, V. J. Primary Index. In: LIU, L.; ÖZSU, M. T. (Eds.). **Encyclopedia of Database Systems**. Springer, 2009. v. 1, p. 2135-2136.

NASCIMENTO, S. M.; TSURUDA, R. M.; SIQUEIRA, T. L. L.; TIMES, V. C. T.; CIFERRI, R. R.; CIFERRI, C. D. A. The Spatial Star Schema Benchmark. In: BRAZILIAN SYMPOSIUM ON GEOINFORMATICS (GEOINFO), 12nd, 2011, Campos do Jordão, SP. **Proceedings...** São José dos Campos, SP: MCT/INPE, 2011. p.73-84.

O'NEIL, P. E.; O'NEIL, E. J.; CHEN, X. **Star Schema Benchmark**. Disponível em: <http://www.cs.umb.edu/~poneil/StarSchemaB.PDF>. Acesso em: out. 2012.

O'NEIL, P.; QUASS, D. Improved query performance with variant indexes. **SIGMOD Record**, New York, NY, v. 26, n. 2, p. 38-49, jun. 1997.

PAPADIAS, D.; KALNIS, P.; ZHANG, J.; TAO, Y. Efficient OLAP Operations in Spatial Data Warehouses. In: INTERNATIONAL SYMPOSIUM ON ADVANCES IN SPATIAL AND TEMPORAL DATABASES (SSTD), 7th, 2001, Redondo Beach, CA. **Proceedings...** Berlin / Heidelberg: Springer, 2001. p. 443-459. (Lecture Notes in Computer Science, v. 2121).

PAPADIAS, D.; TAO, Y.; KALNIS, P.; ZHANG, J. Indexing Spatio-Temporal Data Warehouses. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING (ICDE), 18th, 2002, San Jose, CA. **Proceedings...** Washington, DC: IEEE, 2002. p. 166-175.

PESTANA, G.; SILVA, M. M. DA. Multidimensional modeling based on spatial, temporal and spatio-temporal stereotypes. In: ESRI INTERNATIONAL USER CONFERENCE, 2005, San Diego, CA. **Proceedings...** 2005.

POURABBAS, E.; RAFANELLI, M. Characterization of hierarchies and some operators in OLAP environment. In: ACM INTERNATIONAL WORKSHOP ON DATA

WAREHOUSING AND OLAP (DOLAP), 2nd, 1999, Kansas City, Missouri. **Proceedings...** New York, NY: ACM, 1999. p. 54-59.

RIGAUX, P.; SCHOLL, M.; VOISARD, A. **Spatial Databases**: With Application to GIS. San Francisco, CA: Morgan Kaufmann, 2001. 410 p.

RIZZI, S. Conceptual Modeling Solutions for the Data Warehouse. In: WREMBEL, R.; KONCILIA, C. (Eds.). **Data Warehouses and OLAP**: Concepts, Architectures and Solutions. Hershey, PA: IRM Press, 2007. p. 1-26.

SAMPAIO, M. C.; SOUSA, A. G. de; BAPTISTA, C. de S. Towards a logical multidimensional model for spatial data warehousing and OLAP. In: INTERNATIONAL WORKSHOP ON DATA WAREHOUSING AND OLAP (DOLAP), 9th, 2006, Arlington, Virginia. **Proceedings...** New York, NY: ACM, 2006. p. 83-90.

SIQUEIRA, T. L. L. **SB-INDEX**: um índice espacial baseado em bitmap para data warehouse geográfico. 2009. 118 f. Dissertação (Mestrado em Ciência da Computação) - Departamento de Computação, Universidade Federal de São Carlos, São Carlos. 2009.

SIQUEIRA, T. L. L.; CIFERRI, C. D. A.; TIMES, V. C.; CIFERRI, R. R. The SB-index and the HSB-index: efficient indices for spatial data warehouses. **Geoinformatica**, Hingham, MA, v. 16, n. 1, p. 165-205, jan. 2012.

SIQUEIRA, T. L. L.; CIFERRI, C. D. A.; TIMES, V. C.; OLIVEIRA, A. G. de; CIFERRI, R. R. The impact of spatial data redundancy on SOLAP query performance. **Journal of the Brazilian Computer Society**, v. 15, n. 2, p. 19-34, jun. 2009.

SIQUEIRA, T. L. L.; CIFERRI, R. R.; TIMES, V. C.; CIFERRI, C. D. A. Investigating the Effects of Spatial Data Redundancy in Query Performance over Geographical Data Warehouses. In: BRAZILIAN SYMPOSIUM ON GEOINFORMATICS (GeoInfo), 10th, 2008, Rio de Janeiro, Brazil. **Proceedings...** INPE, 2008. p. 1-12.

SIQUEIRA, T. L. L.; CIFERRI, R. R.; TIMES, V. C.; CIFERRI, C. D. A. A spatial bitmap-based index for geographical data warehouses. In: ACM SYMPOSIUM ON APPLIED COMPUTING (SAC), 2009, Honolulu, Hawaii. **Proceedings...** New York, NY: ACM, 2009. p. 1336-1342.

SIQUEIRA, T. L. L.; CIFERRI, R. R.; TIMES, V. C.; CIFERRI, C. D. A. Benchmarking spatial data warehouses. In: INTERNATIONAL CONFERENCE ON DATA WAREHOUSING AND KNOWLEDGE DISCOVERY (DaWaK), 12th, 2010, Bilbao, Spain. **Proceedings...** Berlin / Heidelberg: Springer, 2010. p. 40-51.

STEFANOVIC, N. **Design and Implementation of On-Line Analytical Processing (OLAP) of Spatial Data**. 1997. 108 p. Master's Thesis (Master of Science) - Department of Computing Science, Simon Fraser University, Canada. 1997.

STOCKINGER, K.; WU, K. Bitmap Indices for Data Warehouses. In: WREMBEL, R.; KONCILIA, C. (Eds.). **Data Warehouses and OLAP**: Concepts, Architectures and Solutions. Hershey, PA: IRM Press, 2007. p. 157-178.

TAO, Y.; PAPADIAS, D. Spatio-Temporal Data Warehouses. In: LIU, L.; ÖZSU, M. T. (Eds.). **Encyclopedia of Database Systems**. Springer, 2009. v. 1, p. 2731-2735.

TRAINA JR., C.; TRAINA, A. J. M.; SEEGER, B.; FALOUTSOS, C. Slim-Trees: High Performance Metric Trees Minimizing Overlap Between Nodes. In: INTERNATIONAL CONFERENCE ON EXTENDING DATABASE TECHNOLOGY: ADVANCES IN DATABASE TECHNOLOGY (EDBT), 7th, 2000, Konstanz, Germany. **Proceedings...** Berlin / Heidelberg: Springer, 2000. p. 51-65.

U.S. CENSUS BUREAU. **TIGER**: Topologically Integrated Geographic Encoding and Referencing system. Disponível em: <http://www.census.gov/geo/www/tiger>. Acesso em: out. 2012.

VAISMAN, A.; ZIMÁNYI, E. What is Spatio-Temporal Data Warehousing? In: INTERNATIONAL CONFERENCE ON DATA WAREHOUSING AND KNOWLEDGE DISCOVERY (DaWaK), 11th, 2009, Linz, Austria. **Proceedings...** Berlin / Heidelberg: Springer, 2009. p. 9-23. (Lecture Notes in Computer Science, v. 5691).

Apêndice A

CÓDIGO PARA MODIFICAÇÃO DOS OBJETOS ESPACIAIS

```
-- *** MODIFICAÇÃO DOS ENDEREÇOS DOS FORNECEDORES *** --

-- Operações: alter

/*
Escolha dos endereços a serem alterados.
*/

CREATE OR REPLACE FUNCTION f_adds_select (percentage REAL) RETURNS VOID AS
$$
DECLARE
  quant INTEGER;
  aux INTEGER;
BEGIN
  RAISE NOTICE '*** FUNCTION f_adds_select ***';

  SELECT INTO aux COUNT(*) FROM supplier WHERE s_finalvt IS NULL;

  quant = CAST( (aux * percentage) AS INTEGER );

  CREATE TABLE as_alter AS
    SELECT s_suppkey FROM supplier WHERE s_finalvt IS NULL
    ORDER BY RANDOM()
    LIMIT quant;
END;
$$ LANGUAGE plpgsql;

/*
Criação de uma tabela temporária com todos os atributos dos fornecedores
cujo endereço irá se alterar.
*/

CREATE OR REPLACE FUNCTION f_adds_create () RETURNS VOID AS $$
BEGIN
  RAISE NOTICE '*** FUNCTION f_adds_create ***';

  CREATE TABLE as_alter1 AS
    SELECT * FROM supplier WHERE s_suppkey IN (
      SELECT * FROM as_alter
    );
```

```
END;
$$ LANGUAGE plpgsql;

/*
Atualização dos fornecedores com os novos endereços.
*/

CREATE OR REPLACE FUNCTION f_adds_update1 (initialvt TIMESTAMP WITH TIME
ZONE) RETURNS VOID AS $$
DECLARE
  quant INTEGER;
  cur_suppliers CURSOR FOR
    SELECT s_suppkey FROM as_alter1 ORDER BY s_suppkey;
  var_supplier supplier.s_suppkey%TYPE;
  cur_addresses REFCURSOR;
  var_address address%ROWTYPE;
  id INTEGER;
BEGIN
  RAISE NOTICE '*** FUNCTION f_adds_update1 ***';

  -- Seleção de novos endereços. Utilização da função "generate_series"
  --  para agilizar a escolha, pois a tabela de endereços é muito grande.
  SELECT INTO quant COUNT(*) FROM as_alter;

  OPEN cur_addresses FOR
    SELECT * FROM address WHERE
      address_pk IN (
        SELECT FLOOR( RANDOM() * (max_id - min_id + 1) )::INTEGER + min_id
FROM
          GENERATE_SERIES(1, quant * 5),
          ( SELECT MAX(address_pk) AS max_id, MIN(address_pk) AS min_id
FROM address ) t
        LIMIT quant * 5
      )
      AND address_pk NOT IN (
        SELECT s_address_fk FROM supplier WHERE s_finalvt IS NULL
        UNION
        SELECT c_address_fk FROM customer WHERE c_finalvt IS NULL
      )
    ORDER BY RANDOM()
    LIMIT quant;

  RAISE NOTICE 'ADDS --> % new addresses selected!', quant;

  -- Atualização dos atributos dos fornecedores escolhidos.
  SELECT INTO id MAX(s_suppkey) FROM supplier;
  id := id + 1;

  OPEN cur_suppliers;
  LOOP
    FETCH cur_suppliers INTO var_supplier;
    FETCH cur_addresses INTO var_address;
    EXIT WHEN NOT FOUND;

    UPDATE as_alter1 SET
      s_suppkey = id,
      s_address_geo = var_address.address_geo,
      s_address_fk = var_address.address_pk,
      s_address = var_address.street_fk || ' ST, ' ||
var_address.address_pk,
      s_street_fk = var_address.street_fk,
```

```
        s_street = var_address.street_fk || ' ST',
        s_initialvt = initialvt,
        s_finalvt = NULL
      WHERE s_suppkey = var_supplier;

      id := id + 1;
    END LOOP;
    CLOSE cur_suppliers;

    UPDATE as_alter1 SET
      s_city_fk = ct.city_pk,
      s_city = SUBSTRING(SUBSTRING(nt.nation_name FROM 1 FOR 8) FROM '[A-
Z]*.[A-Z]*') || ' CT ' || ct.city_pk,
      s_nation_fk = nt.nation_pk,
      s_nation = nt.nation_name,
      s_region_fk = rg.region_pk,
      s_region = rg.region_name
    FROM street st, city ct, nation nt, region rg
    WHERE s_street_fk = st.street_pk
      AND st.city_fk = ct.city_pk
      AND ct.nation_fk = nt.nation_pk
      AND nt.region_fk = rg.region_pk;

    RAISE NOTICE 'ADDS --> Suppliers updated!';
END;
$$ LANGUAGE plpgsql;

/*
Atualização da tabela de fornecedores (Supplier) e da tabela de fatos
(Lineorder).
*/

CREATE OR REPLACE FUNCTION f_adds_update2 (finalvt TIMESTAMP WITH TIME
ZONE, initialvt TIMESTAMP WITH TIME ZONE) RETURNS VOID AS $$
BEGIN
    RAISE NOTICE '*** FUNCTION f_adds_update2 ***';

    -- Atualização do tempo válido final dos fornecedores cujo endereço se
    --  alterou.
    UPDATE supplier  SET s_finalvt = finalvt WHERE s_suppkey IN (
      SELECT * FROM as_alter
    );

    -- Inserção dos fornecedores com os novos endereços.
    INSERT INTO supplier SELECT * FROM as_alter1;

    -- Atualização da chave estrangeira dos fornecedores na tabela de fatos.
    UPDATE lineorder SET lo_suppkey = s_suppkey
    FROM as_alter1
    WHERE lo_suppid = s_suppid
      AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');
END;
$$ LANGUAGE plpgsql;

/*
Remoção das tabelas auxiliares.
*/

CREATE OR REPLACE FUNCTION f_adds_drop () RETURNS VOID AS $$
BEGIN
    RAISE NOTICE '*** FUNCTION f_adds_drop ***';
```

```
  DROP TABLE as_alter1;
  DROP TABLE as_alter;
END;
$$ LANGUAGE plpgsql;



-- *** MODIFICAÇÃO DOS ENDEREÇOS DOS CLIENTES *** --

-- Operações: alter

/*
Escolha dos endereços a serem alterados.
*/

CREATE OR REPLACE FUNCTION f_addc_select (percentage REAL) RETURNS VOID AS
$$
DECLARE
  quant INTEGER;
  aux INTEGER;
BEGIN
  RAISE NOTICE '*** FUNCTION f_addc_select ***';

  SELECT INTO aux COUNT(*) FROM customer WHERE c_finalvt IS NULL;

  quant = CAST( (aux * percentage) AS INTEGER );

  CREATE TABLE ac_alter AS
    SELECT c_custkey FROM customer WHERE c_finalvt IS NULL
    ORDER BY RANDOM()
    LIMIT quant;
END;
$$ LANGUAGE plpgsql;

/*
Criação de uma tabela temporária com todos os atributos dos clientes cujo
endereço irá se alterar.
*/

CREATE OR REPLACE FUNCTION f_addc_create () RETURNS VOID AS $$
BEGIN
  RAISE NOTICE '*** FUNCTION f_addc_create ***';

  CREATE TABLE ac_alter1 AS
    SELECT * FROM customer WHERE c_custkey IN (
      SELECT * FROM ac_alter
    );
END;
$$ LANGUAGE plpgsql;

/*
Atualização dos clientes com os novos endereços.
*/

CREATE OR REPLACE FUNCTION f_addc_update1 (initialvt TIMESTAMP WITH TIME
ZONE) RETURNS VOID AS $$
DECLARE
  quant INTEGER;
  cur_customers CURSOR FOR
    SELECT c_custkey FROM ac_alter1 ORDER BY c_custkey;
```

```
  var_customer customer.c_custkey%TYPE;
  cur_addresses REFCURSOR;
  var_address address%ROWTYPE;
  id INTEGER;
BEGIN
  RAISE NOTICE '*** FUNCTION f_addc_update1 ***';

  -- Seleção de novos endereços. Utilização da função "generate_series"
  --  para agilizar a escolha, pois a tabela de endereços é muito grande.
  SELECT INTO quant COUNT(*) FROM ac_alter;

  OPEN cur_addresses FOR
    SELECT * FROM address WHERE
      address_pk IN (
        SELECT FLOOR( RANDOM() * (max_id - min_id + 1) )::INTEGER + min_id
FROM
          GENERATE_SERIES(1, quant * 5),
          ( SELECT MAX(address_pk) AS max_id, MIN(address_pk) AS min_id
FROM address ) t
        LIMIT quant * 5
      )
      AND address_pk NOT IN (
        SELECT s_address_fk FROM supplier WHERE s_finalvt IS NULL
        UNION
        SELECT c_address_fk FROM customer WHERE c_finalvt IS NULL
      )
    ORDER BY RANDOM()
    LIMIT quant;

  RAISE NOTICE 'ADDC --> % new addresses selected!', quant;

  -- Alteração dos atributos dos clientes escolhidos.
  SELECT INTO id MAX(c_custkey) FROM customer;
  id := id + 1;

  OPEN cur_customers;
  LOOP
    FETCH cur_customers INTO var_customer;
    FETCH cur_addresses INTO var_address;
    EXIT WHEN NOT FOUND;

    UPDATE ac_alter1 SET
      c_custkey = id,
      c_address_geo = var_address.address_geo,
      c_address_fk = var_address.address_pk,
      c_address = var_address.street_fk || ' ST, ' ||
var_address.address_pk,
      c_street_fk = var_address.street_fk,
      c_street = var_address.street_fk || ' ST',
      c_initialvt = initialvt,
      c_finalvt = NULL
    WHERE c_custkey = var_customer;

    id := id + 1;
  END LOOP;
  CLOSE cur_customers;

  UPDATE ac_alter1 SET
    c_city_fk = ct.city_pk,
    c_city = SUBSTRING(SUBSTRING(nt.nation_name FROM 1 FOR 8) FROM '[A-
Z]*.[A-Z]*') || ' CT ' || ct.city_pk,
```

```
    c_nation_fk = nt.nation_pk,
    c_nation = nt.nation_name,
    c_region_fk = rg.region_pk,
    c_region = rg.region_name
  FROM street st, city ct, nation nt, region rg
  WHERE c_street_fk = st.street_pk
    AND st.city_fk = ct.city_pk
    AND ct.nation_fk = nt.nation_pk
    AND nt.region_fk = rg.region_pk;

  RAISE NOTICE 'ADDC --> Customers updated!';
END;
$$ LANGUAGE plpgsql;

/*
Atualização da tabela de clientes (customer) e da tabela de fatos
(lineorder).
*/

CREATE OR REPLACE FUNCTION f_addc_update2 (finalvt TIMESTAMP WITH TIME
ZONE, initialvt TIMESTAMP WITH TIME ZONE) RETURNS VOID AS $$
BEGIN
  RAISE NOTICE '*** FUNCTION f_addc_update2 ***';

  -- Alteração do tempo válido final dos clientes cujo endereço se alterou.

  UPDATE customer  SET c_finalvt = finalvt WHERE c_custkey IN (
    SELECT * FROM ac_alter
  );

  -- Inserção dos clientes com os novos endereços.
  INSERT INTO customer SELECT * FROM ac_alter1;

  -- Atualização da chave estrangeira dos clientes na tabela de fatos.
  UPDATE lineorder SET lo_custkey = c_custkey
  FROM ac_alter1
  WHERE lo_custid = c_custid
    AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');
END;
$$ LANGUAGE plpgsql;

/*
Remoção das tabelas auxiliares.
*/

CREATE OR REPLACE FUNCTION f_addc_drop () RETURNS VOID AS $$
BEGIN
  RAISE NOTICE '*** FUNCTION f_addc_drop ***';

  DROP TABLE ac_alter1;
  DROP TABLE ac_alter;
END;
$$ LANGUAGE plpgsql;



-- *** MODIFICAÇÃO DAS RUAS *** --

-- Operações: 1/3 alter, 1/3 split, 1/6 + 1/6 split & merge

/*
```

```
Escolha das ruas a serem alteradas e divididas.
*/

CREATE OR REPLACE FUNCTION f_str_select1 (percentage REAL) RETURNS VOID AS
$$
DECLARE
  quant INTEGER;
  aux INTEGER;
BEGIN
  RAISE NOTICE '*** FUNCTION f_str_select1 ***';

  SELECT INTO aux COUNT(*) FROM street WHERE street_finalvt IS NULL;
  quant := CAST(((aux * percentage) / 3) AS INTEGER);

  CREATE TABLE s_alter AS
    SELECT street_pk AS pk
    FROM (
      SELECT street_pk FROM street
      WHERE street_finalvt IS NULL
      ORDER BY RANDOM()
      LIMIT quant
    ) t
    ORDER BY pk;

  CREATE TABLE s_split AS
    SELECT street_pk AS pk
    FROM (
      SELECT street_pk FROM street
      WHERE street_finalvt IS NULL
        AND street_pk NOT IN (
          SELECT * FROM s_alter
        )
      ORDER BY RANDOM()
      LIMIT quant
    ) t
    ORDER BY pk;

  CREATE TABLE s_merge (
    pk1 INTEGER,
    pk2 INTEGER,
    loc1 REAL,
    loc2 REAL
  );

  PERFORM AddGeometryColumn('s_merge', 'geo1', -1, 'LINESTRING', 2);
  PERFORM AddGeometryColumn('s_merge', 'geo2', -1, 'LINESTRING', 2);
  PERFORM AddGeometryColumn('s_merge', 'geo1_1', -1, 'LINESTRING', 2);
  PERFORM AddGeometryColumn('s_merge', 'geo1_2', -1, 'LINESTRING', 2);
  PERFORM AddGeometryColumn('s_merge', 'geo2_1', -1, 'LINESTRING', 2);
  PERFORM AddGeometryColumn('s_merge', 'geo2_2', -1, 'LINESTRING', 2);

  CREATE TABLE s_temp_merge (
    pk1 INTEGER,
    pk2 INTEGER,
    loc1 REAL,
    loc2 REAL
  );

  PERFORM AddGeometryColumn('s_temp_merge', 'geo1', -1, 'LINESTRING', 2);
  PERFORM AddGeometryColumn('s_temp_merge', 'geo2', -1, 'LINESTRING', 2);
END;
```

```
$$ LANGUAGE plpgsql;

/*
Escolha das ruas a serem unidas.
*/

CREATE OR REPLACE FUNCTION f_str_select2 (percentage REAL) RETURNS VOID AS
$$
DECLARE
  aux INTEGER;
  quant INTEGER;
  cur_merge REFCURSOR;
  var_merge1 RECORD;
  var_merge2 RECORD;
  num INTEGER;
  x1 NUMERIC(7,3);
  y1 NUMERIC(7,3);
  x2 NUMERIC(7,3);
  y2 NUMERIC(7,3);
BEGIN
  RAISE NOTICE '*** FUNCTION f_str_select2 ***';

  SELECT INTO aux COUNT(*) FROM street WHERE street_finalvt IS NULL;
  quant := CAST(((aux * percentage) / 6) AS INTEGER);

  WHILE quant > 0 LOOP
    INSERT INTO s_temp_merge (pk1) (
      SELECT street_pk
      FROM street
      WHERE street_finalvt IS NULL
        AND street_pk NOT IN (
          SELECT pk FROM s_alter
          UNION
          SELECT pk FROM s_split
          UNION
          SELECT pk1 FROM s_merge
          UNION
          SELECT pk2 FROM s_merge
        )
      ORDER BY RANDOM()
      LIMIT quant
    );

    UPDATE s_temp_merge
    SET geo1 = street_geo
    FROM street
    WHERE street_pk = pk1;

    OPEN cur_merge FOR
      SELECT pk1, geo1 FROM s_temp_merge ORDER BY pk1;
    LOOP
      FETCH cur_merge INTO var_merge1;
      EXIT WHEN NOT FOUND;

      SELECT INTO var_merge2 *
      FROM (
        SELECT t1.street_pk, t1.street_geo, ST_Intersects(var_merge1.geo1,
t1.street_geo) AS flag
        FROM (
          SELECT street_pk, street_geo
          FROM street
```

```
            WHERE street_finalvt IS NULL
              AND street_pk NOT IN (
                SELECT pk FROM s_alter
                UNION
                SELECT pk FROM s_split
                UNION
                SELECT pk1 FROM s_merge
                UNION
                SELECT pk2 FROM s_merge
                UNION
                SELECT pk1 FROM s_temp_merge
                UNION
                SELECT pk2 FROM s_temp_merge WHERE pk2 IS NOT NULL
              )
          ) AS t1
          WHERE var_merge1.pk1 <> t1.street_pk
        ) AS t2
        WHERE t2.flag = true
        ORDER BY RANDOM()
        LIMIT 1;

        UPDATE s_temp_merge
        SET pk2 = var_merge2.street_pk,
          geo2 = var_merge2.street_geo
        WHERE pk1 = var_merge1.pk1;
      END LOOP;

      CLOSE cur_merge;

      DELETE FROM s_temp_merge WHERE pk2 IS NULL;

      UPDATE s_temp_merge
      SET loc1 = ST_Line_Locate_Point(geo1, ST_Intersection(geo1, geo2)),
        loc2 = ST_Line_Locate_Point(geo2, ST_Intersection(geo1, geo2));

      DELETE FROM s_temp_merge
      WHERE loc1 = 0 OR loc1 = 1 OR loc2 = 0 OR loc2 = 1;

      INSERT INTO s_merge
      SELECT * FROM s_temp_merge;

      SELECT INTO num COUNT(*)
      FROM s_temp_merge;

      quant := quant - num;

      RAISE NOTICE 'quant: %', quant;

      DELETE FROM s_temp_merge;

    END LOOP;
    DROP TABLE s_temp_merge;

    -- Armazenamento das ruas divididas.
    UPDATE s_merge
    SET geo1_1 = ST_Line_Substring(geo1, 0, loc1),
      geo1_2 = ST_Line_Substring(geo1, loc1, 1),
      geo2_1 = ST_Line_Substring(geo2, 0, loc2),
      geo2_2 = ST_Line_Substring(geo2, loc2, 1);

    OPEN cur_merge FOR
```

```
    SELECT pk1 FROM s_merge;
  LOOP
    FETCH cur_merge INTO var_merge1;
    EXIT WHEN NOT FOUND;

    SELECT INTO x1 ST_X(ST_EndPoint(geo1_1)) FROM s_merge WHERE pk1 =
var_merge1.pk1;
    SELECT INTO y1 ST_Y(ST_EndPoint(geo1_1)) FROM s_merge WHERE pk1 =
var_merge1.pk1;
    SELECT INTO x2 ST_X(ST_StartPoint(geo1_2)) FROM s_merge WHERE pk1 =
var_merge1.pk1;
    SELECT INTO y2 ST_Y(ST_StartPoint(geo1_2)) FROM s_merge WHERE pk1 =
var_merge1.pk1;

    UPDATE s_merge SET geo1_1 = ST_RemovePoint(geo1_1, ST_NPoints(geo1_1)
- 1) WHERE pk1 = var_merge1.pk1;
    UPDATE s_merge SET geo1_2 = ST_RemovePoint(geo1_2, 0) WHERE pk1 =
var_merge1.pk1;
    UPDATE s_merge SET geo2_1 = ST_RemovePoint(geo2_1, ST_NPoints(geo2_1)
- 1) WHERE pk1 = var_merge1.pk1;
    UPDATE s_merge SET geo2_2 = ST_RemovePoint(geo2_2, 0) WHERE pk1 =
var_merge1.pk1;

    UPDATE s_merge SET geo1_1 = ST_AddPoint(geo1_1, ST_Point(x1, y1))
WHERE pk1 = var_merge1.pk1;
    UPDATE s_merge SET geo1_2 = ST_AddPoint(geo1_2, ST_Point(x2, y2), 0)
WHERE pk1 = var_merge1.pk1;
    UPDATE s_merge SET geo2_1 = ST_AddPoint(geo2_1, ST_Point(x1, y1))
WHERE pk1 = var_merge1.pk1;
    UPDATE s_merge SET geo2_2 = ST_AddPoint(geo2_2, ST_Point(x2, y2), 0)
WHERE pk1 = var_merge1.pk1;
  END LOOP;
  CLOSE cur_merge;
END;
$$ LANGUAGE plpgsql;

/*
Criação de tabela temporária com todos os atributos das ruas que se
modificaram.
*/

CREATE OR REPLACE FUNCTION f_str_create () RETURNS VOID AS $$
BEGIN
  RAISE NOTICE '*** FUNCTION f_str_create ***';

  -- Ruas que se alteraram.
  CREATE TABLE s_alter1 AS SELECT * FROM street WHERE street_pk IN (
    SELECT pk FROM s_alter
  );

  -- Ruas que se dividiram.
  CREATE TABLE s_split1 AS SELECT * FROM street WHERE street_pk IN (
    SELECT * FROM s_split
  );

  CREATE TABLE s_split2 AS SELECT * FROM s_split1;

  -- Ruas que se dividiram (depois irão se unir).
  CREATE TABLE s_split1merge1 AS SELECT * FROM street WHERE street_pk IN (
    SELECT pk1 FROM s_merge
  );
```

```
  CREATE TABLE s_split1merge2 AS SELECT * FROM s_split1merge1;

  CREATE TABLE s_split2merge1 AS SELECT * FROM street WHERE street_pk IN (
    SELECT pk2 FROM s_merge
  );

  CREATE TABLE s_split2merge2 AS SELECT * FROM s_split2merge1;
  CREATE TABLE s_merge1merge1 AS SELECT * FROM s_split1merge1 WHERE 1 = 2;
  CREATE TABLE s_merge1merge2 AS SELECT * FROM s_merge1merge1;
  CREATE TABLE s_merge2merge1 AS SELECT * FROM s_merge1merge1;
  CREATE TABLE s_merge2merge2 AS SELECT * FROM s_merge1merge1;
  CREATE TABLE s_merge1 AS SELECT * FROM s_merge1merge1;
  CREATE TABLE s_merge2 AS SELECT * FROM s_merge1merge1;

  ALTER TABLE s_merge1 ADD COLUMN street_pk_oldnew2 INTEGER;
  PERFORM AddGeometryColumn('s_merge1', 'street_geo2', -1, 'LINESTRING',
2);

  ALTER TABLE s_merge2 ADD COLUMN street_pk_oldnew2 INTEGER;
  PERFORM AddGeometryColumn('s_merge2', 'street_geo2', -1, 'LINESTRING',
2);

  -- Endereços que irão se alterar em decorrência da modificação das ruas.
  CREATE TABLE as_alter (
    s_suppkey INTEGER
  );

  CREATE TABLE ac_alter (
    c_custkey INTEGER
  );

  CREATE TABLE temp_supplier AS SELECT * FROM supplier WHERE 1 = 2;
  CREATE TABLE temp_customer AS SELECT * FROM customer WHERE 1 = 2;
END;
$$ LANGUAGE plpgsql;

/*
Atualização dos atributos das ruas que se alteraram.
*/

CREATE OR REPLACE FUNCTION f_str_update_alter1 (finalvt TIMESTAMP WITH TIME
ZONE, initialvt TIMESTAMP WITH TIME ZONE) RETURNS VOID AS $$
DECLARE
  cur_street REFCURSOR;
  var_street street.street_pk%TYPE;
  pk INTEGER;
  id INTEGER;
BEGIN
  RAISE NOTICE '*** FUNCTION f_str_update_alter1 ***';

  -- Atualização do tempo válido final das ruas que se alteraram.
  UPDATE street
  SET street_finalvt = finalvt
  WHERE street_pk IN (
    SELECT sa.pk FROM s_alter sa
  );
  RAISE NOTICE 'Tempo válido final atualizado!';

  -- Atualização dos atributos street_geo, street_operation,
  --  street_pk_oldnew e street_initialvt.
```

```
  UPDATE s_alter1
  SET street_geo = ST_Line_SubString(street_geo, 0, 0.9),
    street_operation = 'a',
    street_pk_oldnew = street_pk,
    street_initialvt = initialvt;

  RAISE NOTICE 'Ruas alteradas atualizadas!';

  -- Atualização de street_pk e street_id.
  SELECT INTO pk MAX(street_pk) FROM street;
  SELECT INTO id MAX(street_id) FROM street;

  pk := pk + 1;
  id := id + 1;

  OPEN cur_street FOR
    SELECT street_pk FROM s_alter1 ORDER BY street_pk;
  LOOP
    FETCH cur_street INTO var_street;
    EXIT WHEN NOT FOUND;
    UPDATE s_alter1 SET street_pk = pk, street_id = id WHERE street_pk =
var_street;
    pk := pk + 1;
    id := id + 1;
  END LOOP;
  CLOSE cur_street;

  RAISE NOTICE 'street_pk e street_id atualizados!';

  INSERT INTO street SELECT * FROM s_alter1;
END;
$$ LANGUAGE plpgsql;

/*
Atualização dos endereços decorrente da alteração das ruas.
*/

CREATE OR REPLACE FUNCTION f_str_update_alter2 (finalvt TIMESTAMP WITH TIME
ZONE, initialvt TIMESTAMP WITH TIME ZONE) RETURNS VOID AS $$
DECLARE
  cur_street REFCURSOR;
  var_street street%ROWTYPE;
  cur_address REFCURSOR;
  var_address RECORD;
  cur_supplier REFCURSOR;
  var_supplier supplier.s_suppkey%TYPE;
  cur_customer REFCURSOR;
  var_customer customer.c_custkey%TYPE;
  pk INTEGER;
  flag BOOLEAN;
BEGIN
  RAISE NOTICE '*** FUNCTION f_str_update_alter2 ***';

  INSERT INTO temp_supplier SELECT * FROM supplier WHERE s_finalvt IS NULL
    AND s_street_fk IN (
      SELECT sa.pk FROM s_alter sa
    );
  INSERT INTO temp_customer SELECT * FROM customer WHERE c_finalvt IS NULL
    AND c_street_fk IN (
      SELECT sa.pk FROM s_alter sa
    );
```

```
  OPEN cur_street FOR
    SELECT * FROM s_alter1 ORDER BY street_pk;
  LOOP
    FETCH cur_street INTO var_street;
    EXIT WHEN NOT FOUND;

    OPEN cur_address FOR
      SELECT address_pk, address_geo FROM address WHERE street_fk =
var_street.street_pk_oldnew;
    LOOP
      FETCH cur_address INTO var_address;
      EXIT WHEN NOT FOUND;

      SELECT INTO flag ST_Intersects(var_street.street_geo,
var_address.address_geo);

      IF NOT flag THEN
        INSERT INTO as_alter
          SELECT s_suppkey FROM temp_supplier WHERE s_address_fk =
var_address.address_pk;
        DELETE FROM temp_supplier WHERE s_address_fk =
var_address.address_pk;

        INSERT INTO ac_alter
          SELECT c_custkey FROM temp_customer WHERE c_address_fk =
var_address.address_pk;
        DELETE FROM temp_customer WHERE c_address_fk =
var_address.address_pk;

        DELETE FROM address WHERE address_pk = var_address.address_pk;
      END IF;
    END LOOP;
    CLOSE cur_address;
  END LOOP;
  CLOSE cur_street;

  UPDATE supplier SET s_finalvt = finalvt WHERE s_finalvt IS NULL
    AND s_suppkey IN (
      SELECT t.s_suppkey FROM temp_supplier t
    );
  UPDATE customer SET c_finalvt = finalvt WHERE c_finalvt IS NULL
    AND c_custkey IN (
      SELECT t.c_custkey FROM temp_customer t
    );

  UPDATE address
  SET street_fk = street_pk
  FROM s_alter1
  WHERE street_fk = street_pk_oldnew;

  -- s_city_fk e s_city devem ser atualizados quando as cidades se alteram.
  UPDATE temp_supplier
  SET s_initialvt = initialvt, s_city_fk = s.city_fk,
      s_city = SUBSTRING(SUBSTRING(s_city FROM 1 FOR 8) FROM '[A-Z]*.[A-
Z]*') || ' CT ' || s.city_fk,
      s_street_fk = s.street_pk, s_street = s.street_pk || ' ST',
      s_address = s.street_pk || ' ST, ' || s_address_fk
  FROM s_alter1 s
  WHERE s_street_fk = s.street_pk_oldnew;
```

```
  -- c_city_fk e c_city devem ser atualizados quando as cidades se alteram.
  UPDATE temp_customer
  SET c_initialvt = initialvt, c_city_fk = s.city_fk,
      c_city = SUBSTRING(SUBSTRING(c_city FROM 1 FOR 8) FROM '[A-Z]*.[A-
Z]*') || ' CT ' || s.city_fk,
      c_street_fk = s.street_pk, c_street = s.street_pk || ' ST',
      c_address = s.street_pk || ' ST, ' || c_address_fk
  FROM s_alter1 s
  WHERE c_street_fk = s.street_pk_oldnew;

  SELECT INTO pk MAX(s_suppkey) FROM supplier;
  pk := pk + 1;

  OPEN cur_supplier FOR
    SELECT s_suppkey FROM temp_supplier ORDER BY s_suppkey;
  LOOP
    FETCH cur_supplier INTO var_supplier;
    EXIT WHEN NOT FOUND;
    UPDATE temp_supplier SET s_suppkey = pk WHERE s_suppkey =
var_supplier;
    pk := pk + 1;
  END LOOP;
  CLOSE cur_supplier;

  SELECT INTO pk MAX(c_custkey) FROM customer;
  pk := pk + 1;

  OPEN cur_customer FOR
    SELECT c_custkey FROM temp_customer ORDER BY c_custkey;
  LOOP
    FETCH cur_customer INTO var_customer;
    EXIT WHEN NOT FOUND;
    UPDATE temp_customer SET c_custkey = pk WHERE c_custkey =
var_customer;
    pk := pk + 1;
  END LOOP;
  CLOSE cur_customer;

  INSERT INTO supplier SELECT * FROM temp_supplier;
  INSERT INTO customer SELECT * FROM temp_customer;

  UPDATE lineorder SET lo_suppkey = s_suppkey
  FROM temp_supplier
  WHERE lo_suppid = s_suppid
    AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

  UPDATE lineorder SET lo_custkey = c_custkey
  FROM temp_customer
  WHERE lo_custid = c_custid
    AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

  DELETE FROM temp_supplier;
  DELETE FROM temp_customer;

  PERFORM f_adds_create();
  PERFORM f_adds_update1(initialvt);
  PERFORM f_adds_update2(finalvt, initialvt);

  PERFORM f_addc_create();
  PERFORM f_addc_update1(initialvt);
  PERFORM f_addc_update2(finalvt, initialvt);
```

```
END;
$$ LANGUAGE plpgsql;

/*
Atualização dos atributos das ruas que se dividiram.
*/

CREATE OR REPLACE FUNCTION f_str_update_split1 (finalvt TIMESTAMP WITH TIME
ZONE, initialvt TIMESTAMP WITH TIME ZONE) RETURNS VOID AS $$
DECLARE
  cur_street REFCURSOR;
  var_street street.street_pk%TYPE;
  pk INTEGER;
  id INTEGER;
BEGIN
  RAISE NOTICE '*** FUNCTION f_str_update_split1 ***';

  -- Alteração do tempo válido final das ruas que se modificaram.
  UPDATE street
  SET street_finalvt = finalvt
  WHERE street_pk IN (
    SELECT s.pk FROM s_split s
  );
  RAISE NOTICE 'Tempo válido final atualizado!';

  -- Alteração dos atributos street_geo, street_operation, street_pk_oldnew
e street_initialvt.
  UPDATE s_split1
  SET street_geo = ST_Line_SubString(street_geo, 0, 0.5),
    street_operation = 's',
    street_pk_oldnew = street_pk,
    street_initialvt = initialvt,
    street_finalvt = NULL;

  UPDATE s_split2
  SET street_geo = ST_Line_SubString(street_geo, 0.5, 1),
    street_operation = 's',
    street_pk_oldnew = street_pk,
    street_initialvt = initialvt,
    street_finalvt = NULL;

  RAISE NOTICE 'Ruas divididas atualizadas!';

  -- Alteração de street_pk e street_id.
  SELECT INTO pk MAX(street_pk) FROM street;
  SELECT INTO id MAX(street_id) FROM street;

  pk := pk + 1;
  id := id + 1;

  OPEN cur_street FOR
    SELECT street_pk FROM s_split1 ORDER BY street_pk;
  LOOP
    FETCH cur_street INTO var_street;
    EXIT WHEN NOT FOUND;
    UPDATE s_split1 SET street_pk = pk, street_id = id WHERE street_pk =
var_street;
    pk := pk + 1;
    id := id + 1;
```

```
    UPDATE s_split2 SET street_pk = pk, street_id = id WHERE street_pk =
var_street;
    pk := pk + 1;
    id := id + 1;
  END LOOP;
  CLOSE cur_street;

  RAISE NOTICE 'street_pk e street_id atualizados!';

  INSERT INTO street SELECT * FROM s_split1;
  INSERT INTO street SELECT * FROM s_split2;
END;
$$ LANGUAGE plpgsql;

/*
Atualização dos endereços decorrente da divisão das ruas.
*/

CREATE OR REPLACE FUNCTION f_str_update_split2 (finalvt TIMESTAMP WITH TIME
ZONE, initialvt TIMESTAMP WITH TIME ZONE) RETURNS VOID AS $$
DECLARE
  cur_supplier REFCURSOR;
  var_supplier supplier.s_suppkey%TYPE;
  cur_customer REFCURSOR;
  var_customer customer.c_custkey%TYPE;
  pk INTEGER;
BEGIN
  RAISE NOTICE '*** FUNCTION f_str_update_split2 ***';

  INSERT INTO temp_supplier SELECT * FROM supplier WHERE s_finalvt IS NULL
    AND s_street_fk IN (
      SELECT ss.pk FROM s_split ss
    );
  INSERT INTO temp_customer SELECT * FROM customer WHERE c_finalvt IS NULL
    AND c_street_fk IN (
      SELECT ss.pk FROM s_split ss
    );

  UPDATE supplier SET s_finalvt = finalvt WHERE s_finalvt IS NULL
    AND s_street_fk IN (
      SELECT ss.pk FROM s_split ss
    );
  UPDATE customer SET c_finalvt = finalvt WHERE c_finalvt IS NULL
    AND c_street_fk IN (
      SELECT ss.pk FROM s_split ss
    );

  UPDATE address
  SET street_fk = s.street_pk
  FROM s_split1 s
  WHERE street_fk = s.street_pk_oldnew AND ST_INTERSECTS(address_geo,
s.street_geo);

  -- s_city_fk e s_city devem ser atualizados quando as cidades se alteram.
  UPDATE temp_supplier
  SET s_initialvt = initialvt, s_city_fk = s.city_fk,
      s_city = SUBSTRING(SUBSTRING(s_city FROM 1 FOR 8) FROM '[A-Z]*.[A-
Z]*') || ' CT ' || s.city_fk,
      s_street_fk = s.street_pk, s_street = s.street_pk || ' ST',
      s_address = s.street_pk || ' ST, ' || s_address_fk
  FROM s_split1 s
```

```
  WHERE s_street_fk = s.street_pk_oldnew AND ST_INTERSECTS(s_address_geo,
s.street_geo);

  -- c_city_fk e c_city devem ser atualizados quando as cidades se alteram.
  UPDATE temp_customer
  SET c_initialvt = initialvt, c_city_fk = s.city_fk,
      c_city = SUBSTRING(SUBSTRING(c_city FROM 1 FOR 8) FROM '[A-Z]*.[A-
Z]*') || ' CT ' || s.city_fk,
      c_street_fk = s.street_pk, c_street = s.street_pk || ' ST',
      c_address = s.street_pk || ' ST, ' || c_address_fk
  FROM s_split1 s
  WHERE c_street_fk = s.street_pk_oldnew AND ST_INTERSECTS(c_address_geo,
s.street_geo);

  UPDATE address
  SET street_fk = street_pk
  FROM s_split2
  WHERE street_fk = street_pk_oldnew;

  -- s_city_fk e s_city devem ser atualizados quando as cidades se alteram.
  UPDATE temp_supplier
  SET s_initialvt = initialvt, s_city_fk = s.city_fk,
      s_city = SUBSTRING(SUBSTRING(s_city FROM 1 FOR 8) FROM '[A-Z]*.[A-
Z]*') || ' CT ' || s.city_fk,
      s_street_fk = s.street_pk, s_street = s.street_pk || ' ST',
      s_address = s.street_pk || ' ST, ' || s_address_fk
  FROM s_split2 s
  WHERE s_street_fk = s.street_pk_oldnew;

  -- c_city_fk e c_city devem ser atualizados quando as cidades se alteram.
  UPDATE temp_customer
  SET c_initialvt = initialvt, c_city_fk = s.city_fk,
      c_city = SUBSTRING(SUBSTRING(c_city FROM 1 FOR 8) FROM '[A-Z]*.[A-
Z]*') || ' CT ' || s.city_fk,
      c_street_fk = s.street_pk, c_street = s.street_pk || ' ST',
      c_address = s.street_pk || ' ST, ' || c_address_fk
  FROM s_split2 s
  WHERE c_street_fk = s.street_pk_oldnew;

  SELECT INTO pk MAX(s_suppkey) FROM supplier;
  pk := pk + 1;

  OPEN cur_supplier FOR
    SELECT s_suppkey FROM temp_supplier ORDER BY s_suppkey;
  LOOP
    FETCH cur_supplier INTO var_supplier;
    EXIT WHEN NOT FOUND;
    UPDATE temp_supplier SET s_suppkey = pk WHERE s_suppkey =
var_supplier;
    pk := pk + 1;
  END LOOP;
  CLOSE cur_supplier;

  SELECT INTO pk MAX(c_custkey) FROM customer;
  pk := pk + 1;

  OPEN cur_customer FOR
    SELECT c_custkey FROM temp_customer ORDER BY c_custkey;
  LOOP
    FETCH cur_customer INTO var_customer;
    EXIT WHEN NOT FOUND;
```

```
    UPDATE temp_customer SET c_custkey = pk WHERE c_custkey =
var_customer;
    pk := pk + 1;
  END LOOP;
  CLOSE cur_customer;

  INSERT INTO supplier SELECT * FROM temp_supplier;
  INSERT INTO customer SELECT * FROM temp_customer;

  UPDATE lineorder SET lo_suppkey = s_suppkey
  FROM temp_supplier
  WHERE lo_suppid = s_suppid
    AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

  UPDATE lineorder SET lo_custkey = c_custkey
  FROM temp_customer
  WHERE lo_custid = c_custid
    AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

  DELETE FROM temp_supplier;
  DELETE FROM temp_customer;
END;
$$ LANGUAGE plpgsql;

/*
Atualização dos atributos das ruas que se uniram.
*/

CREATE OR REPLACE FUNCTION f_str_update_merge1 (finalvt TIMESTAMP WITH TIME
ZONE, initialvt TIMESTAMP WITH TIME ZONE) RETURNS VOID AS $$
DECLARE
  cur_street REFCURSOR;
  var_street street.street_pk%TYPE;
  cur_street2 REFCURSOR;
  var_street2 street.street_pk%TYPE;
  pk INTEGER;
  id INTEGER;
BEGIN
  RAISE NOTICE '*** FUNCTION f_str_update_merge1 ***';

  -- Alteração do tempo válido final das ruas que se uniram.
  UPDATE street
  SET street_finalvt = finalvt
  WHERE street_pk IN (
    SELECT sm.pk1 FROM s_merge sm
    UNION
    SELECT sm.pk2 FROM s_merge sm
  );
  RAISE NOTICE 'Tempo válido final atualizado!';

  -- Alteração dos atributos street_geo, street_operation, street_pk_oldnew
e street_initialvt.
  UPDATE s_split1merge1
  SET street_operation = 's',
    street_pk_oldnew = street_pk,
    street_initialvt = initialvt,
    street_finalvt = initialvt,
    street_geo = m.geo1_1
  FROM s_merge m
  WHERE street_pk = m.pk1;
```

```
  UPDATE s_split1merge2
  SET street_operation = 's',
    street_pk_oldnew = street_pk,
    street_initialvt = initialvt,
    street_finalvt = initialvt,
    street_geo = m.geo1_2
  FROM s_merge m
  WHERE street_pk = m.pk1;

  UPDATE s_split2merge1
  SET street_operation = 's',
    street_pk_oldnew = street_pk,
    street_initialvt = initialvt,
    street_finalvt = initialvt,
    street_geo = m.geo2_1
  FROM s_merge m
  WHERE street_pk = m.pk2;

  UPDATE s_split2merge2
  SET street_operation = 's',
    street_pk_oldnew = street_pk,
    street_initialvt = initialvt,
    street_finalvt = initialvt,
    street_geo = m.geo2_2
  FROM s_merge m
  WHERE street_pk = m.pk2;

  RAISE NOTICE 'Ruas divididas atualizadas!';

  INSERT INTO s_merge1merge1 SELECT * FROM s_split1merge1;
  INSERT INTO s_merge1merge2 SELECT * FROM s_split1merge2;
  INSERT INTO s_merge2merge1 SELECT * FROM s_split2merge1;
  INSERT INTO s_merge2merge2 SELECT * FROM s_split2merge2;

  -- Alteração de street_pk e street_id.
  SELECT INTO pk MAX(street_pk) FROM street;
  SELECT INTO id MAX(street_id) FROM street;

  pk := pk + 1;
  id := id + 1;

  OPEN cur_street FOR
    SELECT street_pk FROM s_split1merge1 ORDER BY street_pk;
  OPEN cur_street2 FOR
    SELECT street_pk FROM s_split2merge1 ORDER BY street_pk;
  LOOP
    FETCH cur_street INTO var_street;
    FETCH cur_street2 INTO var_street2;
    EXIT WHEN NOT FOUND;
    UPDATE s_split1merge1 SET street_pk = pk, street_id = id WHERE
street_pk = var_street;
    UPDATE s_merge1merge1 SET street_id = id WHERE street_pk = var_street;
    pk := pk + 1;
    id := id + 1;
    UPDATE s_split1merge2 SET street_pk = pk, street_id = id WHERE
street_pk = var_street;
    UPDATE s_merge1merge2 SET street_id = id WHERE street_pk = var_street;
    pk := pk + 1;
    id := id + 1;
    UPDATE s_split2merge1 SET street_pk = pk, street_id = id WHERE
street_pk = var_street2;
```

```
    UPDATE s_merge2merge1 SET street_id = id WHERE street_pk =
var_street2;
    pk := pk + 1;
    id := id + 1;
    UPDATE s_split2merge2 SET street_pk = pk, street_id = id WHERE
street_pk = var_street2;
    UPDATE s_merge2merge2 SET street_id = id WHERE street_pk =
var_street2;
    pk := pk + 1;
    id := id + 1;
  END LOOP;
  CLOSE cur_street;
  CLOSE cur_street2;

  OPEN cur_street FOR
    SELECT street_pk FROM s_merge1merge1 ORDER BY street_pk;
  OPEN cur_street2 FOR
    SELECT street_pk FROM s_merge2merge1 ORDER BY street_pk;
  LOOP
    FETCH cur_street INTO var_street;
    FETCH cur_street2 INTO var_street2;
    EXIT WHEN NOT FOUND;
    UPDATE s_merge1merge1 SET street_pk = pk WHERE street_pk = var_street;
    pk := pk + 1;
    UPDATE s_merge1merge2 SET street_pk = pk WHERE street_pk = var_street;
    pk := pk + 1;
    UPDATE s_merge2merge1 SET street_pk = pk WHERE street_pk =
var_street2;
    pk := pk + 1;
    UPDATE s_merge2merge2 SET street_pk = pk WHERE street_pk =
var_street2;
    pk := pk + 1;
  END LOOP;
  CLOSE cur_street;
  CLOSE cur_street2;

  RAISE NOTICE 'street_pk e street_id atualizados!';

  -- Construção das tabelas temporárias com todos os atributos das ruas que
se uniram.
  INSERT INTO s_merge1 SELECT * FROM s_merge1merge1;
  INSERT INTO s_merge2 SELECT * FROM s_merge1merge2;

  UPDATE s_merge1
  SET street_pk_oldnew2 = pk2,
    street_geo2 = geo2_1
  FROM s_merge
  WHERE street_pk_oldnew = pk1;

  UPDATE s_merge2
  SET street_pk_oldnew2 = pk2,
    street_geo2 = geo2_2
  FROM s_merge
  WHERE street_pk_oldnew = pk1;

  -- Alteração de street_pk.
  OPEN cur_street FOR
    SELECT street_pk FROM s_merge1 ORDER BY street_pk;
  OPEN cur_street2 FOR
    SELECT street_pk FROM s_merge2 ORDER BY street_pk;
  LOOP
```

```
    FETCH cur_street INTO var_street;
    FETCH cur_street2 INTO var_street2;
    EXIT WHEN NOT FOUND;
    UPDATE s_merge1 SET street_pk = pk, street_id = id WHERE street_pk =
var_street;
    pk := pk + 1;
    id := id + 1;
    UPDATE s_merge2 SET street_pk = pk, street_id = id WHERE street_pk =
var_street2;
    pk := pk + 1;
    id := id + 1;
  END LOOP;
  CLOSE cur_street;
  CLOSE cur_street2;

  -- Alteração dos atributos street_operation e street_pk_oldnew.
  UPDATE s_merge1merge1 s
  SET street_operation = 'm',
    street_pk_oldnew = m.street_pk
  FROM s_merge1 m
  WHERE m.street_pk_oldnew = s.street_pk_oldnew;

  UPDATE s_merge1merge2 s
  SET street_operation = 'm',
    street_pk_oldnew = m.street_pk
  FROM s_merge2 m
  WHERE m.street_pk_oldnew = s.street_pk_oldnew;

  UPDATE s_merge2merge1 s
  SET street_operation = 'm',
    street_pk_oldnew = m.street_pk
  FROM s_merge1 m
  WHERE m.street_pk_oldnew2 = s.street_pk_oldnew;

  UPDATE s_merge2merge2 s
  SET street_operation = 'm',
    street_pk_oldnew = m.street_pk
  FROM s_merge2 m
  WHERE m.street_pk_oldnew2 = s.street_pk_oldnew;

  -- Alteração dos atributos das ruas que se uniram.
  UPDATE s_merge1
  SET street_geo = ST_Linemerge(ST_Union(street_geo, street_geo2)),
    street_operation = 'n',
    street_pk_oldnew = NULL,
    street_finalvt = NULL;

  UPDATE s_merge2
  SET street_geo = ST_Linemerge(ST_Union(street_geo, street_geo2)),
    street_operation = 'n',
    street_pk_oldnew = NULL,
    street_finalvt = NULL;

  RAISE NOTICE 'Ruas unidas atualizadas!';

  INSERT INTO street SELECT * FROM s_split1merge1;
  INSERT INTO street SELECT * FROM s_split1merge2;
  INSERT INTO street SELECT * FROM s_split2merge1;
  INSERT INTO street SELECT * FROM s_split2merge2;
  INSERT INTO street SELECT * FROM s_merge1merge1;
  INSERT INTO street SELECT * FROM s_merge1merge2;
```

```
  INSERT INTO street SELECT * FROM s_merge2merge1;
  INSERT INTO street SELECT * FROM s_merge2merge2;

  INSERT INTO street (street_pk, street_id, street_operation,
street_pk_oldnew, street_geo,
      street_initialvt, street_finalvt, city_fk) SELECT street_pk,
street_id, street_operation,
      street_pk_oldnew, street_geo, street_initialvt, street_finalvt,
city_fk FROM s_merge1;
  INSERT INTO street (street_pk, street_id, street_operation,
street_pk_oldnew, street_geo,
      street_initialvt, street_finalvt, city_fk) SELECT street_pk,
street_id, street_operation,
      street_pk_oldnew, street_geo, street_initialvt, street_finalvt,
city_fk FROM s_merge2;
END;
$$ LANGUAGE plpgsql;

/*
Atualização dos endereços decorrente da união das ruas.
*/

CREATE OR REPLACE FUNCTION f_str_update_merge2 (finalvt TIMESTAMP WITH TIME
ZONE, initialvt TIMESTAMP WITH TIME ZONE) RETURNS VOID AS $$
DECLARE
  cur_street REFCURSOR;
  var_street RECORD;
  cur_supplier REFCURSOR;
  var_supplier supplier.s_suppkey%TYPE;
  cur_customer REFCURSOR;
  var_customer customer.c_custkey%TYPE;
  pk INTEGER;
BEGIN
  RAISE NOTICE '*** FUNCTION f_str_update_merge2 ***';

  INSERT INTO temp_supplier SELECT * FROM supplier WHERE s_finalvt IS NULL
    AND s_street_fk IN (
      SELECT pk1 FROM s_merge
      UNION
      SELECT pk2 FROM s_merge
    );
  INSERT INTO temp_customer SELECT * FROM customer WHERE c_finalvt IS NULL
    AND c_street_fk IN (
      SELECT pk1 FROM s_merge
      UNION
      SELECT pk2 FROM s_merge
    );

  UPDATE supplier SET s_finalvt = finalvt WHERE s_finalvt IS NULL
    AND s_street_fk IN (
      SELECT pk1 FROM s_merge
      UNION
      SELECT pk2 FROM s_merge
    );
  UPDATE customer SET c_finalvt = finalvt WHERE c_finalvt IS NULL
    AND c_street_fk IN (
      SELECT pk1 FROM s_merge
      UNION
      SELECT pk2 FROM s_merge
    );
```

```
  OPEN cur_street FOR
    SELECT * FROM s_merge ORDER BY pk1, pk2;
  LOOP
    FETCH cur_street INTO var_street;
    EXIT WHEN NOT FOUND;

    SELECT INTO pk street_id FROM s_split1merge1 WHERE street_pk_oldnew =
var_street.pk1;
    SELECT INTO pk street_pk_oldnew FROM s_merge1merge1 WHERE street_id =
pk;

    UPDATE address SET street_fk = street_pk FROM s_merge1
      WHERE (street_fk = var_street.pk1 OR street_fk = var_street.pk2)
        AND street_pk = pk
        AND ST_INTERSECTS(address_geo, street_geo);
    UPDATE temp_supplier SET s_initialvt = initialvt, s_street_fk =
street_pk,
        s_street = street_pk || ' ST', s_address = street_pk || ' ST, ' ||
s_address_fk
      FROM s_merge1
      WHERE (s_street_fk = var_street.pk1 OR s_street_fk = var_street.pk2)
        AND street_pk = pk
        AND ST_INTERSECTS(s_address_geo, street_geo);
    UPDATE temp_customer SET c_initialvt = initialvt, c_street_fk =
street_pk,
        c_street = street_pk || ' ST', c_address = street_pk || ' ST, ' ||
c_address_fk
      FROM s_merge1
      WHERE (c_street_fk = var_street.pk1 OR c_street_fk = var_street.pk2)
        AND street_pk = pk
        AND ST_INTERSECTS(c_address_geo, street_geo);

    SELECT INTO pk street_id FROM s_split1merge2 WHERE street_pk_oldnew =
var_street.pk1;
    SELECT INTO pk street_pk_oldnew FROM s_merge1merge2 WHERE street_id =
pk;

    UPDATE address SET street_fk = street_pk FROM s_merge2
      WHERE (street_fk = var_street.pk1 OR street_fk = var_street.pk2)
        AND street_pk = pk;
    UPDATE temp_supplier SET s_initialvt = initialvt, s_street_fk =
street_pk,
        s_street = street_pk || ' ST', s_address = street_pk || ' ST, ' ||
s_address_fk
      FROM s_merge2
      WHERE (s_street_fk = var_street.pk1 OR s_street_fk = var_street.pk2)
        AND street_pk = pk;
    UPDATE temp_customer SET c_initialvt = initialvt, c_street_fk =
street_pk,
        c_street = street_pk || ' ST', c_address = street_pk || ' ST, ' ||
c_address_fk
      FROM s_merge2
      WHERE (c_street_fk = var_street.pk1 OR c_street_fk = var_street.pk2)
        AND street_pk = pk;

  END LOOP;
  CLOSE cur_street;

  SELECT INTO pk MAX(s_suppkey) FROM supplier;
  pk := pk + 1;
```

```
  OPEN cur_supplier FOR
    SELECT s_suppkey FROM temp_supplier ORDER BY s_suppkey;
  LOOP
    FETCH cur_supplier INTO var_supplier;
    EXIT WHEN NOT FOUND;
    UPDATE temp_supplier SET s_suppkey = pk WHERE s_suppkey =
var_supplier;
    pk := pk + 1;
  END LOOP;
  CLOSE cur_supplier;

  SELECT INTO pk MAX(c_custkey) FROM customer;
  pk := pk + 1;

  OPEN cur_customer FOR
    SELECT c_custkey FROM temp_customer ORDER BY c_custkey;
  LOOP
    FETCH cur_customer INTO var_customer;
    EXIT WHEN NOT FOUND;
    UPDATE temp_customer SET c_custkey = pk WHERE c_custkey =
var_customer;
    pk := pk + 1;
  END LOOP;
  CLOSE cur_customer;

  INSERT INTO supplier SELECT * FROM temp_supplier;
  INSERT INTO customer SELECT * FROM temp_customer;

  UPDATE lineorder SET lo_suppkey = s_suppkey
  FROM temp_supplier
  WHERE lo_suppid = s_suppid
    AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

  UPDATE lineorder SET lo_custkey = c_custkey
  FROM temp_customer
  WHERE lo_custid = c_custid
    AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

  DELETE FROM temp_supplier;
  DELETE FROM temp_customer;
END;
$$ LANGUAGE plpgsql;

/*
Remoção das tabelas auxiliares.
*/

CREATE OR REPLACE FUNCTION f_str_drop () RETURNS VOID AS $$
BEGIN
  RAISE NOTICE '*** FUNCTION f_str_drop ***';

  DROP TABLE as_alter1;
  DROP TABLE as_alter;

  DROP TABLE ac_alter1;
  DROP TABLE ac_alter;

  DROP TABLE temp_supplier;
  DROP TABLE temp_customer;

  DROP TABLE s_alter1;
```

```
  DROP TABLE s_split1;
  DROP TABLE s_split2;
  DROP TABLE s_split1merge1;
  DROP TABLE s_split1merge2;
  DROP TABLE s_split2merge1;
  DROP TABLE s_split2merge2;
  DROP TABLE s_merge1merge1;
  DROP TABLE s_merge1merge2;
  DROP TABLE s_merge2merge1;
  DROP TABLE s_merge2merge2;
  DROP TABLE s_merge1;
  DROP TABLE s_merge2;

  DROP TABLE s_alter;
  DROP TABLE s_split;
  DROP TABLE s_merge;
END;
$$ LANGUAGE plpgsql;



-- *** MODIFICAÇÃO DAS CIDADES *** --

-- Operações: 1/3 alter , 1/3 split, 1/3 merge

/*
Escolha das cidades a serem alteradas e divididas.
*/

CREATE OR REPLACE FUNCTION f_city_select1(percentage REAL) RETURNS VOID AS
$$
DECLARE
  quant INTEGER;
  aux INTEGER;
BEGIN
  RAISE NOTICE '*** FUNCTION f_city_select1 ***';

  SELECT INTO aux COUNT(*) FROM city WHERE city_finalvt IS NULL;
  quant := CAST(((aux * percentage) / 3) AS INTEGER);

  CREATE TABLE c_alter AS
    SELECT city_pk AS pk
    FROM (
      SELECT city_pk FROM city
      WHERE city_finalvt IS NULL
      ORDER BY RANDOM()
      LIMIT quant
    ) t
    ORDER BY pk;

  PERFORM AddGeometryColumn('c_alter', 'geo', -1, 'GEOMETRY', 2 );
  PERFORM AddGeometryColumn('c_alter', 'geo1', -1, 'GEOMETRYCOLLECTION', 2
);
  PERFORM AddGeometryColumn('c_alter', 'geo2', -1, 'GEOMETRY', 2 );

  CREATE TABLE c_split AS
    SELECT city_pk AS pk
    FROM (
      SELECT city_pk FROM city
      WHERE city_finalvt IS NULL
        AND city_pk NOT IN (SELECT pk FROM c_alter)
```

```
      ORDER BY RANDOM()
      LIMIT quant
    ) t
    ORDER BY pk;

  PERFORM AddGeometryColumn('c_split', 'geo', -1, 'GEOMETRY', 2 );
  PERFORM AddGeometryColumn('c_split', 'geo1', -1, 'GEOMETRY', 2 );
  PERFORM AddGeometryColumn('c_split', 'geo2', -1, 'GEOMETRY', 2 );

  CREATE TABLE c_merge (
    pk1 INTEGER,
    pk2 INTEGER,
    new_pk INTEGER
  );

  PERFORM AddGeometryColumn('c_merge', 'geo1', -1, 'GEOMETRY', 2 );
  PERFORM AddGeometryColumn('c_merge', 'geo2', -1, 'GEOMETRY', 2 );

  CREATE TABLE c_temp_merge (
    pk1 INTEGER,
    pk2 INTEGER,
    nation_fk1 INTEGER
  );

  PERFORM AddGeometryColumn('c_temp_merge', 'geo1', -1, 'GEOMETRY', 2 );
  PERFORM AddGeometryColumn('c_temp_merge', 'geo2', -1, 'GEOMETRY', 2 );

END;
$$ LANGUAGE plpgsql;

/*
Escolha das cidades a serem unidas.
*/

CREATE OR REPLACE FUNCTION f_city_select2(percentage REAL) RETURNS VOID AS
$$
DECLARE
  quant INTEGER;
  aux INTEGER;
  cur_merge REFCURSOR;
  var_merge1 RECORD;
  var_merge2 RECORD;
  num INTEGER;
BEGIN
  RAISE NOTICE '*** FUNCTION f_city_select2 ***';

  SELECT INTO aux COUNT(*) FROM city WHERE city_finalvt IS NULL;
  quant := CAST(((aux * percentage) / 6) AS INTEGER);

  WHILE quant > 0 LOOP
    INSERT INTO c_temp_merge (pk1) (
      SELECT city_pk
      FROM city
      WHERE city_finalvt IS NULL
        AND city_pk NOT IN (
          SELECT pk FROM c_alter
          UNION
          SELECT pk FROM c_split
          UNION
          SELECT pk1 FROM c_merge
          UNION
```

```
            SELECT pk2 FROM c_merge
          )
        ORDER BY RANDOM()
        LIMIT quant
      );

      UPDATE c_temp_merge
      SET geo1 = city_geo, nation_fk1 = nation_fk
      FROM city
      WHERE city_pk = pk1;

      OPEN cur_merge FOR
        SELECT pk1, geo1, nation_fk1 FROM c_temp_merge ORDER BY pk1;
      LOOP
        FETCH cur_merge INTO var_merge1;
        EXIT WHEN NOT FOUND;

        SELECT INTO var_merge2 *
        FROM (
          SELECT t1.city_pk, t1.city_geo
          FROM (
            SELECT city_pk, city_geo, nation_fk
            FROM city
            WHERE city_finalvt IS NULL
              AND city_pk NOT IN (
                SELECT pk FROM c_alter
                UNION
                SELECT pk FROM c_split
                UNION
                SELECT pk1 FROM c_merge
                UNION
                SELECT pk2 FROM c_merge
                UNION
                SELECT pk1 FROM c_temp_merge
                UNION
                SELECT pk2 FROM c_temp_merge WHERE pk2 IS NOT NULL
              )
          ) AS t1
          WHERE t1.nation_fk = var_merge1.nation_fk1
            AND GeometryType(ST_Intersection(var_merge1.geo1, t1.city_geo))
like 'MULTILINESTRING'
        ) AS t2
        ORDER BY RANDOM()
        LIMIT 1;

        UPDATE c_temp_merge
        SET pk2 = var_merge2.city_pk,
          geo2 = var_merge2.city_geo
        WHERE pk1 = var_merge1.pk1;
      END LOOP;

      CLOSE cur_merge;

      DELETE FROM c_temp_merge WHERE pk2 IS NULL;

      INSERT INTO c_merge      (pk1, pk2, geo1, geo2) SELECT pk1, pk2, geo1,
geo2 FROM c_temp_merge;

      SELECT INTO num COUNT(*) FROM c_temp_merge;
      quant := quant - num;
      RAISE NOTICE 'quant: %', quant;
```

```
    DELETE FROM c_temp_merge;

  END LOOP;
  DROP TABLE c_temp_merge;
END;
$$ LANGUAGE plpgsql;

/*
Criação de tabela temporária com todos os atributos das cidades que se
modificaram.
*/

CREATE OR REPLACE FUNCTION f_city_create() RETURNS VOID AS $$
BEGIN
  RAISE NOTICE '*** FUNCTION f_city_create ***';

  -- Cidades que se alteraram.
  CREATE TABLE c_alter1 AS SELECT * FROM city
    WHERE city_pk IN (SELECT pk FROM c_alter);

  -- Cidades que se dividiram.
  CREATE TABLE c_split1 AS SELECT * FROM city
    WHERE city_pk IN (SELECT pk FROM c_split);

  CREATE TABLE c_split2 AS SELECT * FROM c_split1;

  -- Cidades que se uniram.
  CREATE TABLE c_merge1 AS SELECT * FROM city WHERE 1 = 2;

  CREATE TABLE temp_region AS SELECT * FROM region WHERE 1 = 2;
  CREATE TABLE temp_nation AS SELECT * FROM nation WHERE 1 = 2;
  CREATE TABLE temp_city AS SELECT * FROM city WHERE 1 = 2;
  CREATE TABLE temp_street AS SELECT * FROM street WHERE 1 = 2;

  CREATE TABLE temp_street1 (
    pk INTEGER
  );

  CREATE TABLE temp_street2 (
    pk INTEGER
  );

  CREATE TABLE s_alter(
    pk INTEGER
  );

  CREATE TABLE s_alter1 AS SELECT * FROM street WHERE 1 = 2;

  CREATE TABLE s_split(
    pk INTEGER
  );

  CREATE TABLE s_split1 AS SELECT * FROM street WHERE 1 = 2;
  CREATE TABLE s_split2 AS SELECT * FROM street WHERE 1 = 2;

  CREATE TABLE as_alter (
    s_suppkey INTEGER
  );

  CREATE TABLE ac_alter (
```

```
    s_suppkey INTEGER
  );

  CREATE TABLE temp_customer AS SELECT * FROM customer WHERE 1 = 2;
  CREATE TABLE temp_supplier AS SELECT * FROM supplier WHERE 1 = 2;
END;
$$ LANGUAGE plpgsql;

/*
Atualização dos atributos das cidades que se alteraram.
*/

CREATE OR REPLACE FUNCTION f_city_update_alter1 (finalvt TIMESTAMP WITH
TIME ZONE, initialvt TIMESTAMP WITH TIME ZONE) RETURNS VOID AS $$
DECLARE
  cur_city REFCURSOR;
  var_city RECORD;
  id INTEGER;
  pk INTEGER;
  num_geo INTEGER;
  area FLOAT;
  max_area FLOAT;
  pos INTEGER;
  pos_f INTEGER;
BEGIN
  RAISE NOTICE '*** FUNCTION f_city_update_alter1 ***';

  UPDATE city SET city_finalvt = finalvt WHERE city_pk IN (SELECT c.pk FROM
c_alter c);

  UPDATE c_alter c SET geo = city_geo FROM city WHERE c.pk = city_pk;

  OPEN cur_city FOR
    SELECT c.pk FROM c_alter c ORDER BY c.pk;
  LOOP
    FETCH cur_city INTO var_city;
    EXIT WHEN NOT FOUND;

    UPDATE c_alter ca SET geo1 = t4.geom3 FROM (
      SELECT ST_Polygonize(t3.geom2) AS geom3 FROM (
        SELECT ST_Union(ST_Boundary(ST_GeometryN(t2.geom1, 1)), t2.line1)
AS geom2 FROM (
          SELECT t1.geom1, ST_MakeLine(ST_MakePoint(t1.xmin, (t1.ymin +
(t1.ymax - t1.ymin) / 10)), ST_MakePoint(t1.xmax, (t1.ymin + (t1.ymax -
t1.ymin) / 10))) AS line1 FROM (
            SELECT geo AS geom1, ST_XMin(geo) AS xmin, ST_XMax(geo) AS
xmax, ST_YMin(geo) AS ymin, ST_YMax(geo) AS ymax
            FROM c_alter c
            WHERE c.pk = var_city.pk
          ) t1
        ) t2
      ) t3
    ) t4
    WHERE ca.pk = var_city.pk;

  END LOOP;
  CLOSE cur_city;

  OPEN cur_city FOR
    SELECT c.pk FROM c_alter c ORDER BY c.pk;
  LOOP
```

```
    FETCH cur_city INTO var_city;
    EXIT WHEN NOT FOUND;

    SELECT INTO num_geo ST_NumGeometries(c.geo1)
    FROM c_alter c WHERE c.pk = var_city.pk;

    max_area := 0;
    pos := 1;
    IF num_geo <> 0 THEN
      FOR i IN 1 .. num_geo LOOP
        SELECT INTO area ST_Area(ST_GeometryN(c.geo1, pos))
        FROM c_alter c WHERE c.pk = var_city.pk;

        IF area > max_area THEN
          max_area := area;
          pos_f := pos;
        END IF;
        pos := pos + 1;
      END LOOP;

      UPDATE c_alter c SET geo2 = ST_Multi(ST_GeometryN(c.geo1, pos_f))
      WHERE c.pk = var_city.pk;
    END IF;

  END LOOP;
  CLOSE cur_city;

  UPDATE c_alter1
  SET city_operation = 'a', city_pk_oldnew = city_pk,
    city_initialvt = initialvt, city_geo = c.geo2
  FROM c_alter c WHERE city_pk = c.pk;

  -- Alteração de street_pk e street_id.
  SELECT INTO pk MAX(city_pk) FROM city;
  SELECT INTO id MAX(city_id) FROM city;

  pk := pk + 1;
  id := id + 1;

  OPEN cur_city FOR
    SELECT c.pk FROM c_alter c ORDER BY c.pk;
  LOOP
    FETCH cur_city INTO var_city;
    EXIT WHEN NOT FOUND;

    UPDATE c_alter1 SET city_pk = pk, city_id = id WHERE city_pk =
var_city.pk;
    pk := pk + 1;
    id := id + 1;

  END LOOP;
  CLOSE cur_city;

  RAISE NOTICE 'Cidades alteradas atualizadas!';

  INSERT INTO city SELECT * FROM c_alter1;
END;
$$ LANGUAGE plpgsql;


/*
```

```
Atualização das ruas e endereços decorrente da alteração das cidades.
*/

CREATE OR REPLACE FUNCTION f_city_update_alter2 (finalvt TIMESTAMP WITH
TIME ZONE, initialvt TIMESTAMP WITH TIME ZONE) RETURNS VOID AS $$
DECLARE
  cur_city REFCURSOR;
  var_city RECORD;
  cur_supplier REFCURSOR;
  var_supplier supplier.s_suppkey%TYPE;
  cur_customer REFCURSOR;
  var_customer customer.c_custkey%TYPE;
  cur_street REFCURSOR;
  var_street street.street_pk%TYPE;
  pk INTEGER;
  id INTEGER;

BEGIN
  RAISE NOTICE '*** FUNCTION f_city_update_alter2 ***';

  OPEN cur_city FOR
    SELECT c.pk, geo2 FROM c_alter c ORDER BY c.pk;
  LOOP
    FETCH cur_city INTO var_city;
    EXIT WHEN NOT FOUND;

    INSERT INTO temp_street
    SELECT * FROM street
    WHERE city_fk = var_city.pk AND street_finalvt IS NULL;

    INSERT INTO temp_street1
    SELECT street_pk FROM temp_street
    WHERE ST_Contains(var_city.geo2, street_geo);

    INSERT INTO temp_street2
    SELECT street_pk FROM temp_street
    WHERE ST_Disjoint(var_city.geo2, street_geo);

    DELETE FROM temp_street WHERE street_pk IN (
      SELECT t1.pk FROM temp_street1 t1
      UNION
      SELECT t2.pk FROM temp_street2 t2
    );


    UPDATE street
    SET city_fk = c.city_pk
    FROM c_alter1 c
    WHERE c.city_pk_oldnew = var_city.pk
      AND street_pk IN (SELECT t.pk FROM temp_street1 t);


    INSERT INTO temp_supplier
    SELECT * FROM supplier
    WHERE s_finalvt IS NULL
      AND s_street_fk IN (SELECT t.pk FROM temp_street1 t);

    UPDATE supplier
    SET s_finalvt = finalvt
    WHERE s_suppkey IN (SELECT s_suppkey FROM temp_supplier);
```

```
     UPDATE temp_supplier
     SET s_initialvt = initialvt, s_city_fk = c.city_pk,
       s_city = SUBSTRING(SUBSTRING(s_city FROM 1 FOR 8) FROM '[A-Z]*.[A-
Z]*') || ' CT ' || c.city_pk
     FROM c_alter1 c
     WHERE c.city_pk_oldnew = var_city.pk;

     SELECT INTO pk MAX(s_suppkey) FROM supplier;
     pk := pk + 1;

     OPEN cur_supplier FOR
       SELECT s_suppkey FROM temp_supplier ORDER BY s_suppkey;
     LOOP
       FETCH cur_supplier INTO var_supplier;
       EXIT WHEN NOT FOUND;
       UPDATE temp_supplier SET s_suppkey = pk WHERE s_suppkey =
var_supplier;
       pk := pk + 1;
     END LOOP;
     CLOSE cur_supplier;

     INSERT INTO supplier SELECT * FROM temp_supplier;

     UPDATE lineorder SET lo_suppkey = s_suppkey
     FROM temp_supplier
     WHERE lo_suppid = s_suppid
       AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

     DELETE FROM temp_supplier;


     INSERT INTO temp_customer
     SELECT * FROM customer
     WHERE c_finalvt IS NULL
       AND c_street_fk IN (SELECT t.pk FROM temp_street1 t);

     UPDATE customer
     SET c_finalvt = finalvt
     WHERE c_custkey IN (SELECT c_custkey FROM temp_customer);

     UPDATE temp_customer
     SET c_initialvt = initialvt, c_city_fk = c.city_pk,
       c_city = SUBSTRING(SUBSTRING(c_city FROM 1 FOR 8) FROM '[A-Z]*.[A-
Z]*') || ' CT ' || c.city_pk
     FROM c_alter1 c
     WHERE c.city_pk_oldnew = var_city.pk;

     SELECT INTO pk MAX(c_custkey) FROM customer;
     pk := pk + 1;

     OPEN cur_customer FOR
       SELECT c_custkey FROM temp_customer ORDER BY c_custkey;
     LOOP
       FETCH cur_customer INTO var_customer;
       EXIT WHEN NOT FOUND;
       UPDATE temp_customer SET c_custkey = pk WHERE c_custkey =
var_customer;
       pk := pk + 1;
     END LOOP;
     CLOSE cur_customer;
```

```
INSERT INTO customer SELECT * FROM temp_customer;

UPDATE lineorder SET lo_custkey = c_custkey
FROM temp_customer
WHERE lo_custid = c_custid
  AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

DELETE FROM temp_customer;


INSERT INTO as_alter
SELECT s_suppkey FROM supplier
WHERE s_finalvt IS NULL
  AND s_street_fk IN (SELECT t.pk FROM temp_street2 t);

INSERT INTO ac_alter
SELECT c_custkey FROM customer
WHERE c_finalvt IS NULL
  AND c_street_fk IN (SELECT t.pk FROM temp_street2 t);

UPDATE street SET street_finalvt = finalvt
WHERE street_pk IN (SELECT t.pk FROM temp_street2 t);


INSERT INTO s_alter SELECT street_pk FROM temp_street;
INSERT INTO s_alter1 SELECT * FROM temp_street;

DELETE FROM temp_street1;
DELETE FROM temp_street2;
DELETE FROM temp_street;

-- Atualização do tempo válido final das ruas que se alteraram.
UPDATE street
SET street_finalvt = finalvt
WHERE street_pk IN (
  SELECT sa.pk FROM s_alter sa
);
RAISE NOTICE 'Tempo válido final atualizado!';

-- Atualização dos atributos street_geo, street_operation,
--  street_pk_oldnew e street_initialvt.
UPDATE s_alter1
SET street_geo = ST_Intersection(street_geo, var_city.geo2),
  street_operation = 'a',
  street_pk_oldnew = street_pk,
  street_initialvt = initialvt,
  city_fk = c.city_pk
FROM c_alter1 c
WHERE c.city_pk_oldnew = var_city.pk;

RAISE NOTICE 'Ruas alteradas atualizadas!';

-- Atualização de street_pk e street_id.
SELECT INTO pk MAX(street_pk) FROM street;
SELECT INTO id MAX(street_id) FROM street;

pk := pk + 1;
id := id + 1;

OPEN cur_street FOR
  SELECT street_pk FROM s_alter1 ORDER BY street_pk;
```

```
    LOOP
      FETCH cur_street INTO var_street;
      EXIT WHEN NOT FOUND;
      UPDATE s_alter1 SET street_pk = pk, street_id = id WHERE street_pk =
var_street;
      pk := pk + 1;
      id := id + 1;
    END LOOP;
    CLOSE cur_street;

    RAISE NOTICE 'street_pk e street_id atualizados!';

    INSERT INTO street SELECT * FROM s_alter1;

    PERFORM f_str_update_alter2(finalvt, initialvt);

    DROP TABLE as_alter1;
    DELETE FROM as_alter;

    DROP TABLE ac_alter1;
    DELETE FROM ac_alter;

    DELETE FROM s_alter;
    DELETE FROM s_alter1;
  END LOOP;
  CLOSE cur_city;
END;
$$ LANGUAGE plpgsql;

/*
Atualização dos atributos das cidades que se dividiram.
*/

CREATE OR REPLACE FUNCTION f_city_update_split1 (finalvt TIMESTAMP WITH
TIME ZONE, initialvt TIMESTAMP WITH TIME ZONE) RETURNS VOID AS $$
DECLARE
  cur_city REFCURSOR;
  var_city RECORD;
  id INTEGER;
  pk INTEGER;
BEGIN
  RAISE NOTICE '*** FUNCTION f_city_update_split1 ***';

  UPDATE city SET city_finalvt = finalvt WHERE city_pk IN (SELECT c.pk FROM
c_split c);

  UPDATE c_split c SET geo = city_geo FROM city WHERE c.pk = city_pk;

  OPEN cur_city FOR
    SELECT c.pk FROM c_split c ORDER BY c.pk;
  LOOP
    FETCH cur_city INTO var_city;
    EXIT WHEN NOT FOUND;

    UPDATE c_split cs SET geo1 = ST_Multi(t4.geom3) FROM (
      SELECT ST_GeometryN(ST_Polygonize(t3.geom2), 1) AS geom3 FROM (
        SELECT ST_Union(ST_Boundary(ST_GeometryN(t2.geom1, 1)), t2.line1)
AS geom2 FROM (
          SELECT t1.geom1, ST_MakeLine(ST_MakePoint(t1.xmin, (t1.ymin +
(t1.ymax - t1.ymin) / 2)),
```

```
                ST_MakePoint(t1.xmax, (t1.ymin + (t1.ymax - t1.ymin) / 2)))
AS line1 FROM (
              SELECT geo AS geom1, ST_XMin(geo) AS xmin, ST_XMax(geo) AS
xmax, ST_YMin(geo) AS ymin, ST_YMax(geo) AS ymax
              FROM c_split c
              WHERE c.pk = var_city.pk
            ) t1
          ) t2
        ) t3
    ) t4
    WHERE cs.pk = var_city.pk;

  END LOOP;
  CLOSE cur_city;

  UPDATE c_split SET geo2 = ST_Multi(ST_Difference(geo, geo1));

  UPDATE c_split1
  SET city_operation = 's', city_pk_oldnew = city_pk,
    city_initialvt = initialvt, city_geo = c.geo1
  FROM c_split c WHERE city_pk = c.pk;

  UPDATE c_split2
  SET city_operation = 's', city_pk_oldnew = city_pk,
    city_initialvt = initialvt, city_geo = c.geo2
  FROM c_split c WHERE city_pk = c.pk;

  -- Alteração de street_pk e street_id.
  SELECT INTO pk MAX(city_pk) FROM city;
  SELECT INTO id MAX(city_id) FROM city;

  pk := pk + 1;
  id := id + 1;

  OPEN cur_city FOR
    SELECT c.pk FROM c_split c ORDER BY c.pk;
  LOOP
    FETCH cur_city INTO var_city;
    EXIT WHEN NOT FOUND;

    UPDATE c_split1 SET city_pk = pk, city_id = id WHERE city_pk =
var_city.pk;
    pk := pk + 1;
    id := id + 1;

    UPDATE c_split2 SET city_pk = pk, city_id = id WHERE city_pk =
var_city.pk;
    pk := pk + 1;
    id := id + 1;

  END LOOP;
  CLOSE cur_city;

  RAISE NOTICE 'Cidades divididas atualizadas!';

  INSERT INTO city SELECT * FROM c_split1;
  INSERT INTO city SELECT * FROM c_split2;

END;
$$ LANGUAGE plpgsql;
```

```
/*
Atualização das ruas e endereços decorrente da divisão das cidades.
*/

CREATE OR REPLACE FUNCTION f_city_update_split2 (finalvt TIMESTAMP WITH
TIME ZONE, initialvt TIMESTAMP WITH TIME ZONE) RETURNS VOID AS $$
DECLARE
  cur_city REFCURSOR;
  var_city RECORD;
  cur_supplier REFCURSOR;
  var_supplier supplier.s_suppkey%TYPE;
  cur_customer REFCURSOR;
  var_customer customer.c_custkey%TYPE;
  cur_street REFCURSOR;
  var_street street.street_pk%TYPE;
  pk INTEGER;
  id INTEGER;
BEGIN
  RAISE NOTICE '*** FUNCTION f_city_update_split2 ***';

  OPEN cur_city FOR
    SELECT c.pk, c.geo1, c.geo2 FROM c_split c ORDER BY c.pk;
  LOOP
    FETCH cur_city INTO var_city;
    EXIT WHEN NOT FOUND;

    INSERT INTO temp_street
    SELECT * FROM street
    WHERE city_fk = var_city.pk AND street_finalvt IS NULL;

    INSERT INTO temp_street1
    SELECT street_pk FROM temp_street
    WHERE ST_Contains(var_city.geo1, street_geo);

    INSERT INTO temp_street2
    SELECT street_pk FROM temp_street
    WHERE ST_Contains(var_city.geo2, street_geo);

    DELETE FROM temp_street WHERE street_pk IN (
      SELECT t1.pk FROM temp_street1 t1
      UNION
      SELECT t2.pk FROM temp_street2 t2
    );

    UPDATE street
    SET city_fk = c.city_pk
    FROM c_split1 c
    WHERE c.city_pk_oldnew = var_city.pk
      AND street_pk IN (SELECT t.pk FROM temp_street1 t);

    UPDATE street
    SET city_fk = c.city_pk
    FROM c_split2 c
    WHERE c.city_pk_oldnew = var_city.pk
      AND street_pk IN (SELECT t.pk FROM temp_street2 t);


    INSERT INTO temp_supplier
    SELECT * FROM supplier
    WHERE s_finalvt IS NULL
      AND s_street_fk IN (SELECT t.pk FROM temp_street1 t);
```

```
     UPDATE supplier
     SET s_finalvt = finalvt
     WHERE s_suppkey IN (SELECT s_suppkey FROM temp_supplier);

     UPDATE temp_supplier
     SET s_initialvt = initialvt, s_city_fk = c.city_pk,
       s_city = SUBSTRING(SUBSTRING(s_city FROM 1 FOR 8) FROM '[A-Z]*.[A-
Z]*') || ' CT ' || c.city_pk
     FROM c_split1 c
     WHERE c.city_pk_oldnew = var_city.pk;

     SELECT INTO pk MAX(s_suppkey) FROM supplier;
     pk := pk + 1;

     OPEN cur_supplier FOR
       SELECT s_suppkey FROM temp_supplier ORDER BY s_suppkey;
     LOOP
       FETCH cur_supplier INTO var_supplier;
       EXIT WHEN NOT FOUND;
       UPDATE temp_supplier SET s_suppkey = pk WHERE s_suppkey =
var_supplier;
       pk := pk + 1;
     END LOOP;
     CLOSE cur_supplier;

     INSERT INTO supplier SELECT * FROM temp_supplier;

     UPDATE lineorder SET lo_suppkey = s_suppkey
     FROM temp_supplier
     WHERE lo_suppid = s_suppid
       AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

     DELETE FROM temp_supplier;

     INSERT INTO temp_supplier
     SELECT * FROM supplier
     WHERE s_finalvt IS NULL
       AND s_street_fk IN (SELECT t.pk FROM temp_street2 t);

     UPDATE supplier
     SET s_finalvt = finalvt
     WHERE s_suppkey IN (SELECT s_suppkey FROM temp_supplier);

     UPDATE temp_supplier
     SET s_initialvt = initialvt, s_city_fk = c.city_pk,
       s_city = SUBSTRING(SUBSTRING(s_city FROM 1 FOR 8) FROM '[A-Z]*.[A-
Z]*') || ' CT ' || c.city_pk
     FROM c_split2 c
     WHERE c.city_pk_oldnew = var_city.pk;

     SELECT INTO pk MAX(s_suppkey) FROM supplier;
     pk := pk + 1;

     OPEN cur_supplier FOR
       SELECT s_suppkey FROM temp_supplier ORDER BY s_suppkey;
     LOOP
       FETCH cur_supplier INTO var_supplier;
       EXIT WHEN NOT FOUND;
       UPDATE temp_supplier SET s_suppkey = pk WHERE s_suppkey =
var_supplier;
```

```
      pk := pk + 1;
    END LOOP;
    CLOSE cur_supplier;

    INSERT INTO supplier SELECT * FROM temp_supplier;

    UPDATE lineorder SET lo_suppkey = s_suppkey
    FROM temp_supplier
    WHERE lo_suppid = s_suppid
      AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

    DELETE FROM temp_supplier;

    INSERT INTO temp_customer
    SELECT * FROM customer
    WHERE c_finalvt IS NULL
      AND c_street_fk IN (SELECT t.pk FROM temp_street1 t);

    UPDATE customer
    SET c_finalvt = finalvt
    WHERE c_custkey IN (SELECT c_custkey FROM temp_customer);

    UPDATE temp_customer
    SET c_initialvt = initialvt, c_city_fk = c.city_pk,
      c_city = SUBSTRING(SUBSTRING(c_city FROM 1 FOR 8) FROM '[A-Z]*.[A-
Z]*') || ' CT ' || c.city_pk
    FROM c_split1 c
    WHERE c.city_pk_oldnew = var_city.pk;

    SELECT INTO pk MAX(c_custkey) FROM customer;
    pk := pk + 1;

    OPEN cur_customer FOR
      SELECT c_custkey FROM temp_customer ORDER BY c_custkey;
    LOOP
      FETCH cur_customer INTO var_customer;
      EXIT WHEN NOT FOUND;
      UPDATE temp_customer SET c_custkey = pk WHERE c_custkey =
var_customer;
      pk := pk + 1;
    END LOOP;
    CLOSE cur_customer;

    INSERT INTO customer SELECT * FROM temp_customer;

    UPDATE lineorder SET lo_custkey = c_custkey
    FROM temp_customer
    WHERE lo_custid = c_custid
      AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

    DELETE FROM temp_customer;

    INSERT INTO temp_customer
    SELECT * FROM customer
    WHERE c_finalvt IS NULL
      AND c_street_fk IN (SELECT t.pk FROM temp_street2 t);

    UPDATE customer
    SET c_finalvt = finalvt
    WHERE c_custkey IN (SELECT c_custkey FROM temp_customer);
```

```
    UPDATE temp_customer
    SET c_initialvt = initialvt, c_city_fk = c.city_pk,
      c_city = SUBSTRING(SUBSTRING(c_city FROM 1 FOR 8) FROM '[A-Z]*.[A-
Z]*') || ' CT ' || c.city_pk
    FROM c_split2 c
    WHERE c.city_pk_oldnew = var_city.pk;

    SELECT INTO pk MAX(c_custkey) FROM customer;
    pk := pk + 1;

    OPEN cur_customer FOR
      SELECT c_custkey FROM temp_customer ORDER BY c_custkey;
    LOOP
      FETCH cur_customer INTO var_customer;
      EXIT WHEN NOT FOUND;
      UPDATE temp_customer SET c_custkey = pk WHERE c_custkey =
var_customer;
      pk := pk + 1;
    END LOOP;
    CLOSE cur_customer;

    INSERT INTO customer SELECT * FROM temp_customer;

    UPDATE lineorder SET lo_custkey = c_custkey
    FROM temp_customer
    WHERE lo_custid = c_custid
      AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

    DELETE FROM temp_customer;

    INSERT INTO s_split SELECT street_pk FROM temp_street;
    INSERT INTO s_split1 SELECT * FROM temp_street;
    INSERT INTO s_split2 SELECT * FROM temp_street;

    DELETE FROM temp_street1;
    DELETE FROM temp_street2;
    DELETE FROM temp_street;

    -- Alteração do tempo válido final das ruas que se modificaram.
    UPDATE street
    SET street_finalvt = finalvt
    WHERE street_pk IN (SELECT s.pk FROM s_split s);

    RAISE NOTICE 'Tempo válido final atualizado!';

    -- Alteração dos atributos street_geo, street_operation,
    --  street_pk_oldnew e street_initialvt.
    UPDATE s_split1
    SET street_geo = ST_Intersection(street_geo, var_city.geo1),
      street_operation = 's',
      street_pk_oldnew = street_pk,
      street_initialvt = initialvt,
      street_finalvt = NULL,
      city_fk = c.city_pk
    FROM c_split1 c
    WHERE c.city_pk_oldnew = var_city.pk;

    UPDATE s_split2
    SET street_geo = ST_Intersection(street_geo, var_city.geo2),
      street_operation = 's',
      street_pk_oldnew = street_pk,
```

```
       street_initialvt = initialvt,
       street_finalvt = NULL,
       city_fk = c.city_pk
     FROM c_split2 c
     WHERE c.city_pk_oldnew = var_city.pk;

     RAISE NOTICE 'Ruas divididas atualizadas!';

     -- Alteração de street_pk e street_id.
     SELECT INTO pk MAX(street_pk) FROM street;
     SELECT INTO id MAX(street_id) FROM street;

     pk := pk + 1;
     id := id + 1;

     OPEN cur_street FOR
       SELECT street_pk FROM s_split1 ORDER BY street_pk;
     LOOP
       FETCH cur_street INTO var_street;
       EXIT WHEN NOT FOUND;
       UPDATE s_split1 SET street_pk = pk, street_id = id WHERE street_pk =
var_street;
       pk := pk + 1;
       id := id + 1;
       UPDATE s_split2 SET street_pk = pk, street_id = id WHERE street_pk =
var_street;
       pk := pk + 1;
       id := id + 1;
     END LOOP;
     CLOSE cur_street;

     RAISE NOTICE 'street_pk e street_id atualizados!';

     INSERT INTO street SELECT * FROM s_split1;
     INSERT INTO street SELECT * FROM s_split2;

     PERFORM f_str_update_split2(finalvt, initialvt);

     DELETE FROM s_split;
     DELETE FROM s_split1;
     DELETE FROM s_split2;

  END LOOP;
  CLOSE cur_city;
END;
$$ LANGUAGE plpgsql;

/*
Atualização dos atributos das cidades que se uniram.
*/

CREATE OR REPLACE FUNCTION f_city_update_merge1 (finalvt TIMESTAMP WITH
TIME ZONE, initialvt TIMESTAMP WITH TIME ZONE) RETURNS VOID AS $$
DECLARE
  cur_city REFCURSOR;
  var_city RECORD;
  id INTEGER;
  pk INTEGER;
BEGIN
  RAISE NOTICE '*** FUNCTION f_city_update_merge1 ***';
```

```
-- Alteração dos atributos city_operation e city_finalvt das cidades que
--  se uniram.
INSERT INTO temp_city
  SELECT * FROM city c
  WHERE c.city_operation <> 'n'
    AND c.city_pk IN (
      SELECT pk1 FROM c_merge
      UNION
      SELECT pk2 FROM c_merge
    );

UPDATE city
SET city_finalvt = finalvt
WHERE city_pk IN (SELECT city_pk FROM temp_city);

SELECT INTO pk MAX(city_pk) FROM city;

pk := pk + 1;

OPEN cur_city FOR
  SELECT city_pk FROM temp_city ORDER BY city_pk;
LOOP
  FETCH cur_city INTO var_city;
  EXIT WHEN NOT FOUND;
  UPDATE c_merge SET pk1 = pk WHERE pk1 = var_city.city_pk;
  UPDATE c_merge SET pk2 = pk WHERE pk2 = var_city.city_pk;
  UPDATE temp_city SET city_pk = pk WHERE city_pk = var_city.city_pk;

  pk := pk + 1;
END LOOP;
CLOSE cur_city;

INSERT INTO city SELECT * FROM temp_city;
DELETE FROM temp_city;

INSERT INTO c_merge1
  SELECT * FROM city
  WHERE city_pk IN (SELECT pk1 FROM c_merge);

UPDATE city
SET city_operation = 'm',
  city_finalvt = finalvt
WHERE
  city_pk IN (
    SELECT pk1 FROM c_merge
    UNION
    SELECT pk2 FROM c_merge
  );
RAISE NOTICE 'Operação e tempo válido final atualizados!';

-- Alteração dos atributos city_operation, city_pk_oldnew, city_geo e
--  city_initialvt.
UPDATE c_merge1
SET city_operation = 'n',
  city_pk_oldnew = NULL,
  city_geo = ST_Multi(ST_Union(city_geo, m.geo2)),
  city_initialvt = initialvt
FROM c_merge m
WHERE city_pk = m.pk1;

-- Alteração de street_pk e street_id.
```

```
  SELECT INTO id MAX(city_id) FROM city;

  id := id + 1;

  OPEN cur_city FOR
    SELECT pk1, pk2 FROM c_merge ORDER BY pk1;
  LOOP
    FETCH cur_city INTO var_city;
    EXIT WHEN NOT FOUND;
    UPDATE city SET city_pk_oldnew = pk WHERE city_pk = var_city.pk1 OR
city_pk = var_city.pk2;
    UPDATE c_merge SET new_pk = pk WHERE pk1 = var_city.pk1;
    UPDATE c_merge1 SET city_pk = pk, city_id = id WHERE city_pk =
var_city.pk1;

    pk := pk + 1;
    id := id + 1;
  END LOOP;
  CLOSE cur_city;

  RAISE NOTICE 'Cidades unidas atualizadas!';

  INSERT INTO city SELECT * FROM c_merge1;

END;
$$ LANGUAGE plpgsql;

/*
Atualização das ruas e endereços decorrente da união das cidades.
*/

CREATE OR REPLACE FUNCTION f_city_update_merge2 (finalvt TIMESTAMP WITH
TIME ZONE, initialvt TIMESTAMP WITH TIME ZONE) RETURNS VOID AS $$
DECLARE
  cur_supplier REFCURSOR;
  var_supplier supplier.s_suppkey%TYPE;
  cur_customer REFCURSOR;
  var_customer customer.c_custkey%TYPE;
  pk INTEGER;

BEGIN
  RAISE NOTICE '*** FUNCTION f_city_update_merge2 ***';

  UPDATE street
  SET city_fk = c.new_pk
  FROM c_merge c
  WHERE (city_fk = c.pk1 OR city_fk = c.pk2)
    AND street_finalvt IS NULL;


  INSERT INTO temp_supplier
  SELECT * FROM supplier
  WHERE s_finalvt IS NULL
    AND s_city_fk IN (
      SELECT pk1 FROM c_merge
      UNION
      SELECT pk2 FROM c_merge
    );

  UPDATE supplier SET s_finalvt = finalvt WHERE s_suppkey IN (SELECT
s_suppkey FROM temp_supplier);
```

```
  UPDATE temp_supplier
  SET s_initialvt = initialvt, s_city_fk = c.new_pk,
       s_city = SUBSTRING(SUBSTRING(s_city FROM 1 FOR 8) FROM '[A-Z]*.[A-
Z]*') || ' CT ' || c.new_pk
  FROM c_merge c
  WHERE (s_city_fk = c.pk1 OR s_city_fk = c.pk2)
    AND s_finalvt IS NULL;

  SELECT INTO pk MAX(s_suppkey) FROM supplier;
  pk := pk + 1;

  OPEN cur_supplier FOR
    SELECT s_suppkey FROM temp_supplier ORDER BY s_suppkey;
  LOOP
    FETCH cur_supplier INTO var_supplier;
    EXIT WHEN NOT FOUND;
    UPDATE temp_supplier SET s_suppkey = pk WHERE s_suppkey =
var_supplier;
    pk := pk + 1;
  END LOOP;
  CLOSE cur_supplier;

  INSERT INTO supplier SELECT * FROM temp_supplier;

  RAISE NOTICE 'SUPPLIERS UPDATED!';


  INSERT INTO temp_customer
  SELECT * FROM customer
  WHERE c_finalvt IS NULL
    AND c_city_fk IN (
      SELECT pk1 FROM c_merge
      UNION
      SELECT pk2 FROM c_merge
    );

  UPDATE customer SET c_finalvt = finalvt
  WHERE c_custkey IN (SELECT c_custkey FROM temp_customer);

  UPDATE temp_customer
  SET c_initialvt = initialvt, c_city_fk = c.new_pk,
       c_city = SUBSTRING(SUBSTRING(c_city FROM 1 FOR 8) FROM '[A-Z]*.[A-
Z]*') || ' CT ' || c.new_pk
  FROM c_merge c
  WHERE (c_city_fk = c.pk1 OR c_city_fk = c.pk2)
    AND c_finalvt IS NULL;

  SELECT INTO pk MAX(c_custkey) FROM customer;
  pk := pk + 1;

  OPEN cur_customer FOR
    SELECT c_custkey FROM temp_customer ORDER BY c_custkey;
  LOOP
    FETCH cur_customer INTO var_customer;
    EXIT WHEN NOT FOUND;
    UPDATE temp_customer SET c_custkey = pk WHERE c_custkey =
var_customer;
    pk := pk + 1;
  END LOOP;
  CLOSE cur_customer;
```

```
  INSERT INTO customer SELECT * FROM temp_customer;

  RAISE NOTICE 'CUSTOMERS UPDATED!';


  UPDATE lineorder SET lo_suppkey = s_suppkey
  FROM temp_supplier
  WHERE lo_suppid = s_suppid
    AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

  UPDATE lineorder SET lo_custkey = c_custkey
  FROM temp_customer
  WHERE lo_custid = c_custid
    AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

  RAISE NOTICE 'TABLE LINEORDER UPDATED!';

  DELETE FROM temp_supplier;
  DELETE FROM temp_customer;
END;
$$ LANGUAGE plpgsql;

/*
Atualização das nações e regiões decorrente da alteração das cidades.
*/
CREATE OR REPLACE FUNCTION f_city_update_final (finalvt TIMESTAMP WITH TIME
ZONE, initialvt TIMESTAMP WITH TIME ZONE) RETURNS VOID AS $$
DECLARE
  cur_region REFCURSOR;
  var_region region.region_pk%TYPE;
  cur_nation REFCURSOR;
  var_nation nation.nation_pk%TYPE;
  cur_supplier REFCURSOR;
  var_supplier supplier.s_suppkey%TYPE;
  cur_customer REFCURSOR;
  var_customer customer.c_custkey%TYPE;
  pk INTEGER;
  id INTEGER;

BEGIN
  RAISE NOTICE '*** FUNCTION f_city_update_final ***';

  INSERT INTO temp_nation SELECT * FROM nation WHERE nation_pk IN (
    SELECT DISTINCT nation_fk FROM c_alter1);

  UPDATE nation SET nation_finalvt = finalvt WHERE nation_pk IN (
    SELECT nation_pk FROM temp_nation);

  UPDATE temp_nation SET
    nation_initialvt = initialvt,
    nation_operation = 'a',
    nation_pk_oldnew = nation_pk,
    nation_geo = ST_Multi(t.geom)
  FROM (
    SELECT nation_fk, ST_Union(city_geo) AS geom
    FROM city
    WHERE city_finalvt IS NULL
    GROUP BY nation_fk
  ) t
  WHERE nation_pk = t.nation_fk;
```

```
  SELECT INTO pk MAX(nation_pk) FROM nation;
  SELECT INTO id MAX(nation_id) FROM nation;

  pk := pk + 1;
  id := id + 1;

  OPEN cur_nation FOR
    SELECT nation_pk FROM temp_nation ORDER BY nation_pk;
  LOOP
    FETCH cur_nation INTO var_nation;
    EXIT WHEN NOT FOUND;
    UPDATE temp_nation SET nation_pk = pk, nation_id = id,
      nation_name = SUBSTRING(nation_name FROM '[A-Z]*.[A-Z]*') || ' ' ||
pk
    WHERE nation_pk = var_nation;
    pk := pk + 1;
    id := id + 1;
  END LOOP;
  CLOSE cur_nation;

  INSERT INTO nation SELECT * FROM temp_nation;

  RAISE NOTICE 'NATIONS UPDATED!';

  UPDATE city SET nation_fk = nation_pk
  FROM temp_nation
  WHERE city_finalvt IS NULL
    AND nation_finalvt IS NULL
    AND nation_fk = nation_pk_oldnew;

  UPDATE supplier SET s_nation_fk = nation_pk,
      s_nation = SUBSTRING(s_nation FROM '[A-Z]*.[A-Z]*') || ' ' ||
nation_pk
  FROM temp_nation
  WHERE s_initialvt = initialvt
    AND s_nation_fk = nation_pk_oldnew;

  UPDATE customer SET c_nation_fk = nation_pk,
      c_nation = SUBSTRING(c_nation FROM '[A-Z]*.[A-Z]*') || ' ' ||
nation_pk
  FROM temp_nation
  WHERE c_initialvt = initialvt
    AND c_nation_fk = nation_pk_oldnew;

  INSERT INTO temp_supplier
  SELECT * FROM supplier
  WHERE s_initialvt <> initialvt
    AND s_finalvt IS NULL
    AND s_nation_fk IN (SELECT nation_pk_oldnew FROM temp_nation);

  UPDATE supplier SET s_finalvt = finalvt WHERE s_suppkey IN (SELECT
s_suppkey FROM temp_supplier);

  UPDATE temp_supplier SET s_initialvt = initialvt, s_nation_fk =
nation_pk,
      s_nation = SUBSTRING(s_nation FROM '[A-Z]*.[A-Z]*') || ' ' ||
nation_pk
  FROM temp_nation
  WHERE s_nation_fk = nation_pk_oldnew;
```

```
  SELECT INTO pk MAX(s_suppkey) FROM supplier;
  pk := pk + 1;

  OPEN cur_supplier FOR
    SELECT s_suppkey FROM temp_supplier ORDER BY s_suppkey;
  LOOP
    FETCH cur_supplier INTO var_supplier;
    EXIT WHEN NOT FOUND;
    UPDATE temp_supplier SET s_suppkey = pk WHERE s_suppkey =
var_supplier;
    pk := pk + 1;
  END LOOP;
  CLOSE cur_supplier;

  INSERT INTO supplier SELECT * FROM temp_supplier;

  RAISE NOTICE 'SUPPLIERS UPDATED!';

  INSERT INTO temp_customer
  SELECT * FROM customer
  WHERE c_initialvt <> initialvt
    AND c_finalvt IS NULL
    AND c_nation_fk IN (SELECT nation_pk_oldnew FROM temp_nation);

  UPDATE customer SET c_finalvt = finalvt WHERE c_custkey IN (SELECT
c_custkey FROM temp_customer);

  UPDATE temp_customer SET c_initialvt = initialvt, c_nation_fk =
nation_pk,
       c_nation = SUBSTRING(c_nation FROM '[A-Z]*.[A-Z]*') || ' ' ||
nation_pk
  FROM temp_nation
  WHERE c_nation_fk = nation_pk_oldnew;

  SELECT INTO pk MAX(c_custkey) FROM customer;
  pk := pk + 1;

  OPEN cur_customer FOR
    SELECT c_custkey FROM temp_customer ORDER BY c_custkey;
  LOOP
    FETCH cur_customer INTO var_customer;
    EXIT WHEN NOT FOUND;
    UPDATE temp_customer SET c_custkey = pk WHERE c_custkey =
var_customer;
    pk := pk + 1;
  END LOOP;
  CLOSE cur_customer;

  INSERT INTO customer SELECT * FROM temp_customer;

  RAISE NOTICE 'CUSTOMERS UPDATED!';

  UPDATE lineorder SET lo_suppkey = s_suppkey
  FROM temp_supplier
  WHERE lo_suppid = s_suppid
    AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

  RAISE NOTICE 'TABLE LINEORDER UPDATED 1!';

  UPDATE lineorder SET lo_custkey = c_custkey
  FROM temp_customer
```

```
  WHERE lo_custid = c_custid
    AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

  RAISE NOTICE 'TABLE LINEORDER UPDATED 2!';

  DELETE FROM temp_supplier;
  DELETE FROM temp_customer;

  INSERT INTO temp_region SELECT * FROM region WHERE region_pk IN (
    SELECT DISTINCT region_fk FROM temp_nation);

  UPDATE region SET region_finalvt = finalvt WHERE region_pk IN (
    SELECT region_pk FROM temp_region);

  UPDATE temp_region SET
    region_initialvt = initialvt,
    region_operation = 'a',
    region_pk_oldnew = region_pk,
    region_geo = ST_Multi(t.geom)
  FROM (
    SELECT region_fk, ST_Union(nation_geo) AS geom
    FROM nation
    WHERE nation_finalvt IS NULL
    GROUP BY region_fk
  ) t
  WHERE region_pk = t.region_fk;

  SELECT INTO pk MAX(region_pk) FROM region;
  SELECT INTO id MAX(region_id) FROM region;

  pk := pk + 1;
  id := id + 1;

  OPEN cur_region FOR
    SELECT region_pk FROM temp_region ORDER BY region_pk;
  LOOP
    FETCH cur_region INTO var_region;
    EXIT WHEN NOT FOUND;
    UPDATE temp_region SET region_pk = pk, region_id = id,
      region_name = SUBSTRING(region_name FROM '[A-Z]*.[A-Z]*') || ' ' ||
pk
    WHERE region_pk = var_region;
    pk := pk + 1;
    id := id + 1;
  END LOOP;
  CLOSE cur_region;

  INSERT INTO region SELECT * FROM temp_region;

  RAISE NOTICE 'REGIONS UPDATED!';

  UPDATE nation SET region_fk = region_pk
  FROM temp_region
  WHERE nation_finalvt IS NULL
    AND region_finalvt IS NULL
    AND region_fk = region_pk_oldnew;

  UPDATE supplier SET s_region_fk = region_pk,
      s_region = SUBSTRING(s_region FROM '[A-Z]*.[A-Z]*') || ' ' ||
region_pk
  FROM temp_region
```

```
  WHERE s_initialvt = initialvt
    AND s_region_fk = region_pk_oldnew;

  UPDATE customer SET c_region_fk = region_pk,
       c_region = SUBSTRING(c_region FROM '[A-Z]*.[A-Z]*') || ' ' ||
region_pk
  FROM temp_region
  WHERE c_initialvt = initialvt
    AND c_region_fk = region_pk_oldnew;

  INSERT INTO temp_supplier
  SELECT * FROM supplier
  WHERE s_initialvt <> initialvt
    AND s_finalvt IS NULL
    AND s_region_fk IN (SELECT region_pk_oldnew FROM temp_region);

  UPDATE supplier SET s_finalvt = finalvt WHERE s_suppkey IN (SELECT
s_suppkey FROM temp_supplier);

  UPDATE temp_supplier SET s_initialvt = initialvt, s_region_fk =
region_pk,
       s_region = SUBSTRING(s_region FROM '[A-Z]*.[A-Z]*') || ' ' ||
region_pk
  FROM temp_region
  WHERE s_region_fk = region_pk_oldnew;

  SELECT INTO pk MAX(s_suppkey) FROM supplier;
  pk := pk + 1;

  OPEN cur_supplier FOR
    SELECT s_suppkey FROM temp_supplier ORDER BY s_suppkey;
  LOOP
    FETCH cur_supplier INTO var_supplier;
    EXIT WHEN NOT FOUND;
    UPDATE temp_supplier SET s_suppkey = pk WHERE s_suppkey =
var_supplier;
    pk := pk + 1;
  END LOOP;
  CLOSE cur_supplier;

  INSERT INTO supplier SELECT * FROM temp_supplier;

  RAISE NOTICE 'SUPPLIERS UPDATED!';

  INSERT INTO temp_customer
  SELECT * FROM customer
  WHERE c_initialvt <> initialvt
    AND c_finalvt IS NULL
    AND c_region_fk IN (SELECT region_pk_oldnew FROM temp_region);

  UPDATE customer SET c_finalvt = finalvt WHERE c_custkey IN (SELECT
c_custkey FROM temp_customer);

  UPDATE temp_customer SET c_initialvt = initialvt, c_region_fk =
region_pk,
       c_region = SUBSTRING(c_region FROM '[A-Z]*.[A-Z]*') || ' ' ||
region_pk
  FROM temp_region
  WHERE c_region_fk = region_pk_oldnew;

  SELECT INTO pk MAX(c_custkey) FROM customer;
```

```
  pk := pk + 1;

  OPEN cur_customer FOR
    SELECT c_custkey FROM temp_customer ORDER BY c_custkey;
  LOOP
    FETCH cur_customer INTO var_customer;
    EXIT WHEN NOT FOUND;
    UPDATE temp_customer SET c_custkey = pk WHERE c_custkey =
var_customer;
    pk := pk + 1;
  END LOOP;
  CLOSE cur_customer;

  INSERT INTO customer SELECT * FROM temp_customer;

  RAISE NOTICE 'CUSTOMERS UPDATED!';

  UPDATE lineorder SET lo_suppkey = s_suppkey
  FROM temp_supplier
  WHERE lo_suppid = s_suppid
    AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

  RAISE NOTICE 'TABLE LINEORDER UPDATED 1!';

  UPDATE lineorder SET lo_custkey = c_custkey
  FROM temp_customer
  WHERE lo_custid = c_custid
    AND lo_year >= EXTRACT(YEAR from initialvt AT TIME ZONE 'GMT');

  RAISE NOTICE 'TABLE LINEORDER UPDATED 2!';

  DELETE FROM temp_supplier;
  DELETE FROM temp_customer;
  DELETE FROM temp_nation;
  DELETE FROM temp_region;

  RAISE NOTICE 'THE END!';
END;
$$ LANGUAGE plpgsql;

/*
Remoção das tabelas auxiliares.
*/

CREATE OR REPLACE FUNCTION f_city_drop () RETURNS VOID AS $$
BEGIN
  RAISE NOTICE '*** FUNCTION f_city_drop ***';

  DROP TABLE temp_supplier;
  DROP TABLE temp_customer;

  DROP TABLE as_alter;
  DROP TABLE ac_alter;

  DROP TABLE s_alter;
  DROP TABLE s_alter1;

  DROP TABLE s_split;
  DROP TABLE s_split1;
  DROP TABLE s_split2;
```

```
  DROP TABLE temp_street1;
  DROP TABLE temp_street2;
  DROP TABLE temp_street;
  DROP TABLE temp_city;
  DROP TABLE temp_nation;
  DROP TABLE temp_region;

  DROP TABLE c_alter1;
  DROP TABLE c_split1;
  DROP TABLE c_split2;
  DROP TABLE c_merge1;

  DROP TABLE c_alter;
  DROP TABLE c_split;
  DROP TABLE c_merge;
END;
$$ LANGUAGE plpgsql;



-- *** CHAMADA DAS FUNÇÕES PARA MODIFICAÇÃO DOS OBJETOS ESPACIAIS *** --

-- ### PRIMEIRA ALTERAÇÃO ### --

-- Alteração de 5% dos endereços dos fornecedores
SELECT f_adds_select(0.05);
SELECT f_adds_create();
SELECT f_adds_update1(TIMESTAMP WITH TIME ZONE '1993-01-01 -0');
SELECT f_adds_update2(TIMESTAMP WITH TIME ZONE '1992-12-31 23:59:59 -0',
TIMESTAMP WITH TIME ZONE '1993-01-01 -0');
SELECT f_adds_drop();

-- Alteração de 5% dos endereços dos clientes
SELECT f_addc_select(0.05);
SELECT f_addc_create();
SELECT f_addc_update1(TIMESTAMP WITH TIME ZONE '1993-01-01 -0');
SELECT f_addc_update2(TIMESTAMP WITH TIME ZONE '1992-12-31 23:59:59 -0',
TIMESTAMP WITH TIME ZONE '1993-01-01 -0');
SELECT f_addc_drop();

-- Alteração de 5% das ruas
SELECT f_str_select1(0.05);
SELECT f_str_select2(0.05);
SELECT f_str_create();
SELECT f_str_update_alter1(TIMESTAMP WITH TIME ZONE '1993-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1994-01-01 -0');
SELECT f_str_update_alter2(TIMESTAMP WITH TIME ZONE '1993-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1994-01-01 -0');
SELECT f_str_update_split1(TIMESTAMP WITH TIME ZONE '1993-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1994-01-01 -0');
SELECT f_str_update_split2(TIMESTAMP WITH TIME ZONE '1993-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1994-01-01 -0');
SELECT f_str_update_merge1(TIMESTAMP WITH TIME ZONE '1993-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1994-01-01 -0');
SELECT f_str_update_merge2(TIMESTAMP WITH TIME ZONE '1993-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1994-01-01 -0');
SELECT f_str_drop();

-- Alteração de 5% das cidades
SELECT f_city_select1(0.05);
SELECT f_city_select2(0.05);
```

```
SELECT f_city_create();
SELECT f_city_update_alter1(TIMESTAMP WITH TIME ZONE '1994-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1995-01-01 -0');
SELECT f_city_update_alter2(TIMESTAMP WITH TIME ZONE '1994-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1995-01-01 -0');
SELECT f_city_update_split1(TIMESTAMP WITH TIME ZONE '1994-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1995-01-01 -0');
SELECT f_city_update_split2(TIMESTAMP WITH TIME ZONE '1994-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1995-01-01 -0');
SELECT f_city_update_merge1(TIMESTAMP WITH TIME ZONE '1994-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1995-01-01 -0');
SELECT f_city_update_merge2(TIMESTAMP WITH TIME ZONE '1994-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1995-01-01 -0');
SELECT f_city_update_final(TIMESTAMP WITH TIME ZONE '1994-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1995-01-01 -0');
SELECT f_city_drop();

-- ### SEGUNDA ALTERAÇÃO ### --

-- Alteração de 5% dos endereços dos fornecedores
SELECT f_adds_select(0.05);
SELECT f_adds_create();
SELECT f_adds_update1(TIMESTAMP WITH TIME ZONE '1996-01-01 -0');
SELECT f_adds_update2(TIMESTAMP WITH TIME ZONE '1995-12-31 23:59:59 -0',
TIMESTAMP WITH TIME ZONE '1996-01-01 -0');
SELECT f_adds_drop();

-- Alteração de 5% dos endereços dos clientes
SELECT f_addc_select(0.05);
SELECT f_addc_create();
SELECT f_addc_update1(TIMESTAMP WITH TIME ZONE '1996-01-01 -0');
SELECT f_addc_update2(TIMESTAMP WITH TIME ZONE '1995-12-31 23:59:59 -0',
TIMESTAMP WITH TIME ZONE '1996-01-01 -0');
SELECT f_addc_drop();

-- Alteração de 5% das ruas
SELECT f_str_select1(0.05);
SELECT f_str_select2(0.05);
SELECT f_str_create();
SELECT f_str_update_alter1(TIMESTAMP WITH TIME ZONE '1996-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1997-01-01 -0');
SELECT f_str_update_alter2(TIMESTAMP WITH TIME ZONE '1996-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1997-01-01 -0');
SELECT f_str_update_split1(TIMESTAMP WITH TIME ZONE '1996-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1997-01-01 -0');
SELECT f_str_update_split2(TIMESTAMP WITH TIME ZONE '1996-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1997-01-01 -0');
SELECT f_str_update_merge1(TIMESTAMP WITH TIME ZONE '1996-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1997-01-01 -0');
SELECT f_str_update_merge2(TIMESTAMP WITH TIME ZONE '1996-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1997-01-01 -0');
SELECT f_str_drop();

-- Alteração de 5% das cidades
SELECT f_city_select1(0.05);
SELECT f_city_select2(0.05);
SELECT f_city_create();
SELECT f_city_update_alter1(TIMESTAMP WITH TIME ZONE '1997-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1998-01-01 -0');
SELECT f_city_update_alter2(TIMESTAMP WITH TIME ZONE '1997-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1998-01-01 -0');
```

```
SELECT f_city_update_split1(TIMESTAMP WITH TIME ZONE '1997-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1998-01-01 -0');
SELECT f_city_update_split2(TIMESTAMP WITH TIME ZONE '1997-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1998-01-01 -0');
SELECT f_city_update_merge1(TIMESTAMP WITH TIME ZONE '1997-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1998-01-01 -0');
SELECT f_city_update_merge2(TIMESTAMP WITH TIME ZONE '1997-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1998-01-01 -0');
SELECT f_city_update_final(TIMESTAMP WITH TIME ZONE '1997-12-31 23:59:59 -
0', TIMESTAMP WITH TIME ZONE '1998-01-01 -0');
SELECT f_city_drop();
```