LUCIANE TELINSKI WIEDERMANN AGNER

MANUTENÇÃO INCREMENTAL DE VISÕES MATERIALIZADAS EM AMBIENTES *DATA WAREHOUSING*

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Curso de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientadora: Prof.ª Silvia Regina Vergilio

CURITIBA

2000

UFPR
Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática da aluna *Luciane Telinski Wiedermann Agner*, avaliamos o trabalho intitulado **"Manutenção Incremental de Visões Materializadas em Ambientes Datawarehousing "**, cuja defesa foi realizada no dia 06 de outubro de 2000. Após a avaliação, decidimos pela aprovação da Candidata.

Curitiba, 06 de outubro de 2000.

Profª. Dra. Silvia Regina Vergilio
Presidente - DINF/UFPR

Prof. Dr. Mário Jino
DCA-FEEC/UNICAMP

Prof. Dr. Martin Alejandro Musicante
DINF/UFPR

Esta dissertação é dedicada ao meu marido Julio.

Agradecimentos

À minha orientadora, amiga e incentivadora Silvia Regina Vergilio, pela dedicação e apoio.

Ao Julio e a toda a minha família pelo incentivo, amor, carinho e compreensão.

Às amigas Inali e Josiane pela amizade, companheirismo e incentivo.

Aos professores e amigos do Departamento de Informática da Universidade Estadual do Centro-Oeste - UNICENTRO, pela compreensão e apoio.

Aos colegas de mestrado, especialmente ao Aldri, Cristiani, Denis, Fábio, Leonardo, Mozart, Patrícia, Razer, Rudá e Vinícius.

Aos professores do Departamento de Informática da Universidade Federal do Paraná - UFPR, pelo auxílio no processo constante de construção do conhecimento.

Aos funcionários da Universidade Federal do Paraná.

A todos que contribuíram para o desenvolvimento deste trabalho.

E finalmente, gostaria de agradecer a Deus, por guiar meus passos em todos os momentos desta caminhada.

Resumo

Data warehouse é um repositório de dados coletados de fontes de dados distribuídas, autônomas e heterogêneas. A tecnologia *data warehousing* tem sido utilizada em Sistemas de Suporte à Decisão (DSS - *Decision Support Systems*) para auxiliar nos processos decisórios e identificar tendências de mercado. O *data warehouse* armazena uma ou mais visões materializadas dos dados das fontes. A qualidade do processo de tomada de decisão em um DSS depende da correta propagação das atualizações ocorridas nas fontes de dados para as visões materializadas no *data warehouse*. Disso depende a manutenção da consistência dos dados que é em geral um processo complexo. Nos últimos anos, algoritmos de manutenção incremental de visões materializadas em *data warehouse* têm se destacado como uma importante abordagem para o problema. Um estudo comparativo desses algoritmos foi realizado e como conseqüência desse estudo um novo algoritmo, denominado SVM (*Algorithm for **S**cheduling Warehouse **V**iew **M**aintenance*), é aqui proposto. Esse algoritmo combina os aspectos positivos dos algoritmos estudados. Sua principal vantagem é definir intervalos de tempo para propagar as atualizações das fontes no *data warehouse*. Os principais aspectos de implementação do SVM são discutidos e um estudo de caso, composto de diferentes situações que mostram seu funcionamento, é apresentado.

Abstract

A data warehouse is a repository of data collected from distributed, autonomous and heterogeneous sources. Data warehousing technology has been used in Decision Support Systems (DSS) at enabling executives to make better and fast decisions. The data warehouse stores one or more materialized views of the source data. The quality of decisions in a DSS depends on an important issue, that is, the correct propagation of updates at the sources to the views at the data warehouse. Upon it depends on the maintenance of data consistency, which is a complex task. Incremental algorithms for maintenance of views in data warehouses have been proposed recently as an approach to this problem. A study comparing the different algorithms found in the literature was accomplished. As a consequence of this study a new algorithm named SVM is herein proposed. The SVM algorithm combines the positive aspects of the studied algorithms. The main SVM advantage is to permit the definition of time intervals for propagation of the sources updates to the data warehouse. Some SVM implementation aspects are discussed and an example applying the SVM algorithm in different situations is presented.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Exemplos

Capítulo 1

Introdução

1.1. Contexto

Atualmente, o sucesso das organizações está diretamente relacionado com conhecimentos estratégicos e táticos do núcleo do negócio que auxiliam no processo de tomada de decisão. O uso eficiente da informação é fundamental para auxiliar às organizações a atingirem seus objetivos.

Os *data warehouses* ou repositórios de dados surgiram da necessidade de processar, validar, sumarizar e sintetizar os dados da organização de forma a produzir informações que auxiliem nos processos de tomada de decisão e na produção de novos conhecimentos [5].

Data warehouse é um repositório de informações integradas, coletadas de fontes de dados distribuídas, autônomas e possivelmente heterogêneas. Os dados armazenados no *data warehouse* são em geral utilizados em consultas e análises realizadas por sistemas de suporte à decisão (DSS - *Decision Support Systems*) e *data mining* [26].

O *data warehouse* armazena uma ou mais visões materializadas sobre os dados das fontes. Uma visão materializada é uma relação derivada, definida em termos das relações base das fontes, armazenada na base de dados do *data warehouse* [8].

Para haver um processo decisório eficaz é necessário garantir a qualidade e consistência dos dados armazenados no *data warehouse*, pois disso depende o sucesso dos sistemas de apoio à decisão. Nesse sentido, é muito importante realizar a manutenção das visões

materializadas do *data warehouse* que visa a refletir as atualizações ocorridas sobre os dados das relações base armazenadas nas diversas fontes de dados. Um dos principais desafios encontrados neste processo é manter a consistência dos dados do *data warehouse*, especialmente quando as fontes de dados são autônomas e as visões materializam dados de múltiplas fontes [27].

Diversas técnicas oferecem soluções para a manutenção das visões materializadas em ambientes *warehousing* [26]:

- **Reprocessamento**: todos os dados do *data warehouse* são reprocessados periodicamente ou por demanda. Essa técnica consome muito tempo e recursos, particularmente em um ambiente distribuído onde geralmente um grande volume de informações é transferido das fontes para o *data warehouse*. Esta técnica requer que o processamento de consultas seja interrompido no *data warehouse* enquanto a atualização da visão materializada está sendo realizada;
- **Armazenar no *data warehouse* cópias dos dados de todas as relações base envolvidas com a visão materializada (replicação)**: nesse caso, os requisitos de armazenamento são muito altos e os dados replicados armazenados no *data warehouse* precisam ser atualizados sempre que ocorre uma mudança nos dados das fontes. Desta forma, é preciso integrar as atualizações na visão materializada para todos os dados das fontes e não somente para as informações de interesse do sistema de *data warehouse*;
- **Manutenção incremental**: utiliza algoritmos que processam as atualizações na visão materializada em resposta às alterações ocorridas nas relações base, chamados algoritmos de manutenção incremental de visões materializadas. Somente uma parte da visão materializada é modificada em resposta às atualizações ocorridas nas relações base. Manutenção incremental significa que quando ocorrem mudanças nos dados das fontes, novas informações são coletadas das outras fontes de dados relacionadas e integradas no *data warehouse*. Não é necessário interromper o processamento das consultas realizadas pelas aplicações do cliente ao *data warehouse* enquanto a manutenção é realizada, e na maioria dos casos, o tempo requerido para a atualização da visão é menor, se comparado à técnica de reprocessamento. Os benefícios da manutenção incremental sobre a técnica de replicação dos dados das fontes são o menor espaço de armazenamento requerido para o *data warehouse* e o menor número de atualizações a serem processadas no *data warehouse* [8].

A principal desvantagem da manutenção incremental é que esta técnica pode gerar uma anomalia nos dados do *data warehouse*, conforme será mostrado a seguir.

Na manutenção incremental, quando os dados das fontes são modificados é necessário definir quais informações adicionais das fontes relacionadas são necessárias para atualizar a visão materializada. Então, o *data warehouse* deve gerar e enviar consultas a serem avaliadas pelas fontes de dados relacionadas. A Figura 1.1, extraída de [28], ilustra o processamento das mudanças ocorridas nas fontes em um *data warehouse*.

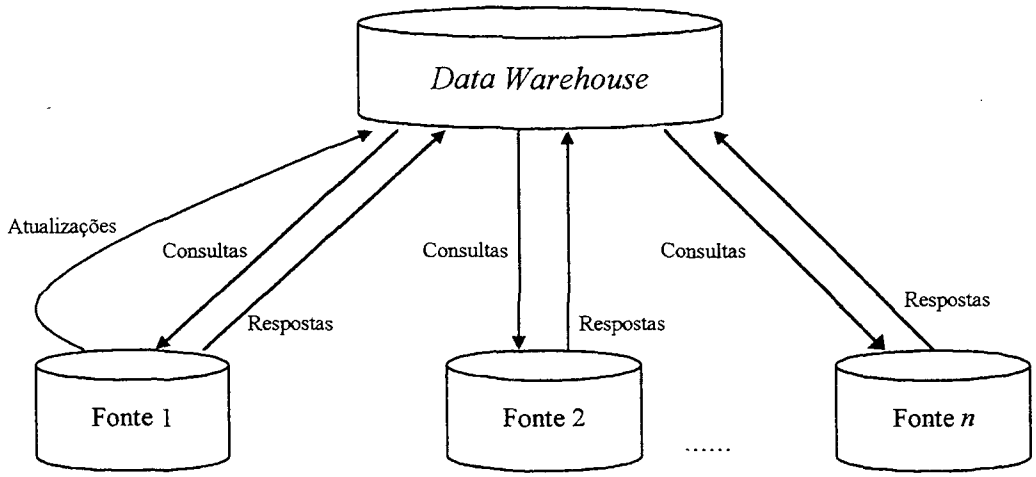

Figura 1.1: Processamento de atualizações em um *data warehouse*

Como novas atualizações podem ocorrer nos dados das fontes enquanto o *data warehouse* está processando uma consulta, esta consulta pode ser avaliada considerando um estado dos dados das fontes que não corresponde ao estado real dos dados quando ocorreu a atualização que gerou a consulta. Anomalias nos dados da visão materializada acontecem quando o *data warehouse* atualiza a visão materializada enquanto a base de dados das fontes está sendo alterada. A inconsistência nos dados ocorre devido à separação entre as fontes, que estão atualizando a base de dados, e o *data warehouse*, que está atualizando a visão [27].

Diversas abordagens foram desenvolvidas para manutenção de visões materializadas em sistemas de base de dados convencionais, centralizadas ou distribuídas [2, 3, 4, 7, 8, 13, 15, 18, 19, 20, 21, 22, 23]. Entretanto, estas abordagens falham em ambientes *data warehousing* pois assumem que a atualização na fonte de dados e a manutenção da visão materializada são realizadas na mesma transação, desta forma, todos os dados das fontes necessários para atualizar as visões materializadas são bloqueados, até que a manutenção seja concluída.

Alguns métodos desenvolvidos para ambientes distribuídos utilizam informações adicionais enviadas pelas fontes de dados para manter as visões materializadas.

Em um ambiente *data warehousing* as fontes podem ser legados ou sistemas não sofisticados, e portanto, possuem as seguintes restrições: 1) Podem informar o *data warehouse* da ocorrência de uma atualização, entretanto, não possuem capacidade de determinar quais os dados pertencentes às fontes de dados relacionadas devem ser integrados com a informação atualizada; 2) Não é possível assumir que elas irão cooperar transmitindo informações adicionais juntamente com as mensagens de atualização.

O Exemplo 1.1, extraído de [27], mostra como a utilização de um algoritmo de manutenção de visões materializadas, proposto para ambientes de base de dados centralizados, pode levar o *data warehouse* a um estado inconsistente. Para este exemplo, são feitas as seguintes suposições: 1) As fontes de dados e o *data warehouse* utilizam o modelo relacional; 2) A visão materializada do *data warehouse* contém os atributos chave para cada relação base que participa da definição da visão; 3) Cada atualização é uma transação separada em uma das fontes de dados. Neste exemplo, a atualização U_2 que ocorre de forma concorrente ao processamento da consulta Q_1 gera uma anomalia na visão materializada, ou seja, os dados da visão materializada resultante são inconsistentes com relação aos dados das fontes.

Para contornar este problema, encontram-se na literatura, alguns algoritmos de manutenção de visões materializadas para ambientes *data warehousing*, destacando-se as famílias de algoritmos ECA e Strobe, propostos por Zhuge et al [26, 27], e os algoritmos SWEEP e Nested SWEEP, propostos por Agrawal et al. [1]. Esses algoritmos diferem principalmente pelo número de fontes relacionadas com o *data warehouse*, pelos diversos cenários de transação considerados e pelos diferentes níveis de consistência que oferecem. Por exemplo, o ECA considera um cenário de *data warehouse* com uma única fonte de dados. Os algoritmos Strobe e Nested SWEEP, projetados para cenários com múltiplas fontes, propagam um conjunto de mudanças das fontes a cada atualização do *data warehouse* (nível de *consistência forte*), no entanto, para atualizar a visão materializada o algoritmo Strobe requer um estado estável do sistema de *data warehouse* e o Nested SWEEP exige que não ocorram atualizações concorrentes. O algoritmo SWEEP proporciona *consistência completa* em cenários com diversas fontes, ou seja, cada atualização das fontes é refletida em um estado distinto do *data warehouse*.

Exemplo 1.1: Anomalia na manutenção incremental da visão

Seja a visão V definida como $V = r_1 \bowtie r_2 \bowtie r_3$, onde r_1, r_2, r_3 são três relações base residindo nas fontes *x*, *y* e *z*, respectivamente. Inicialmente, o estado dos dados das fontes é o seguinte:

r_1: A	B
1	2

r_2: B	C
-	-

r_3: C	D
3	4

MV é a visão materializada no *data warehouse*. Inicialmente, a visão materializada está vazia, ou seja, MV = ∅. No exemplo são consideradas duas atualizações das fontes:

- U_1 = insert (r_2, [2,3]). Insere a tupla [2,3] na relação r_2.
- U_2 = delete (r_1, [1,2]). Remove a tupla [1,2] da relação r_1.

Utilizando um algoritmo de manutenção de visões proposto para ambientes de base de dados centralizados [2], os seguintes eventos podem ocorrer:

- O *data warehouse* recebe U_1 = insert(r_2, [2,3]) da fonte *y*.
- O *data warehouse* gera a consulta $Q_1 = r_1 \bowtie [2,3] \bowtie r_3$. Para avaliar Q_1, o *warehouse* envia primeiramente a consulta $Q_1^1 = r_1 \bowtie [2,3]$ para a fonte *x*.
- O *data warehouse* recebe $A_1^1 = [1,2,3]$ da fonte *x*.
- A consulta $Q_1^2 = [1,2,3] \bowtie r_3$ é enviada para a fonte *z*.
- O *data warehouse* recebe U_2 = delete(r_1, [1,2]) da fonte *x*. Quando o *data warehouse* recebe uma exclusão, ele gera uma ação de exclusão para as tuplas correspondentes em MV, através da comparação dos valores dos campos chave. Como a visão materializada está vazia, nenhuma ação é realizada.
- O *data warehouse* recebe $A_1^2 = [1,2,3,4]$ da fonte *z*. A_1^2 é a resposta final para Q_1. Como não existem mais atualizações a serem processadas, a resposta é inserida em MV e MV = [1,2,3,4]. A visão final é incorreta, pois seus dados são inconsistentes com relação aos dados das fontes.

Algoritmos de manutenção incremental de visões em *data warehouses* são de particular interesse desse trabalho, que foi realizado dentro do contexto de um projeto denominado SAGU (Sistema de Apoio ao Gerenciamento Universitário) [17].

O projeto SAGU tem como objetivo a construção de um sistema de apoio à decisão que deve integrar informações de diversas fontes de dados autônomas e heterogêneas de uma universidade. Este projeto está sendo desenvolvido pelo grupo de pesquisa KDD (*Knowledge Discovery in Database*) do Departamento de Informática da Universidade Federal do Paraná (UFPR) e prevê a integração de informações de diferentes setores desta universidade através da construção de um *data warehouse*, com a finalidade de apoiar e agilizar os processos decisórios e conseqüentemente proporcionar o desenvolvimento acadêmico.

Durante o desenvolvimento do SAGU, os algoritmos de manutenção incremental em ambientes *data warehousing* foram estudados detalhadamente e um novo algoritmo que explora os principais fundamentos e aspectos positivos dos algoritmos estudados foi proposto e é aqui apresentado.

1.2. Motivação

Dado o contexto acima, podem ser resumidos os seguintes itens que serviram como motivação para esse trabalho:

- A tecnologia *data warehousing* é apontada como uma das tendências atuais de mercado, tem despertado grande interesse na comunidade científica e é fundamental para a construção de um sistema de apoio à decisão [5];
- Manter a consistência dos dados do *data warehouse* é um dos principais problemas e desafios relacionados com a tecnologia *data warehousing*;
- Realizar a manutenção incremental das visões materializadas, sem a necessidade de interromper as consultas executadas pelos usuários finais é um dos principais objetivos de um *data warehouse* [25];
- Manter a consistência dos dados do *data warehouse* é um processo complexo, especialmente quando as fontes de dados são autônomas e heterogêneas, e as visões materializam dados de múltiplas fontes [28];

- Um estudo realizado dentro do contexto do projeto SAGU que apontou algumas desvantagens nos algoritmos de manutenção incremental existentes relacionadas ao nível de consistência oferecido, ao número de consultas enviadas para processar as mudanças das fontes e a algumas restrições definidas para efetuar a atualização da visão.

1.3. Objetivos

Este trabalho visa a estudar os principais algoritmos de manutenção incremental de visões materializadas em ambientes *data warehousing* encontrados na literatura e a descrever um novo algoritmo que combina os pontos positivos dos algoritmos existentes. O algoritmo proposto, chamado SVM (*Algorithm for **S**cheduling Warehouse **V**iew **M**aintenance*), atende a um cenário de *data warehouse* com múltiplas fontes, proporciona *consistência forte* que segundo [27] é um quesito adequado à maioria dos *data warehouses* atuais e permite ao usuário definir a periodicidade da manutenção. Aspectos de implementação e validação do algoritmo SVM são também discutidos.

1.4. Organização

Neste capítulo foram introduzidos o problema da manutenção das visões materializadas em *data warehouses*, as motivações e os objetivos deste trabalho.

No Capítulo 2 são descritas as características e considerações do modelo de *data warehouse* adotado e utilizado ao longo deste trabalho. Também é apresentado um resumo de diversos algoritmos para manutenção incremental de visões materializadas em ambientes *warehousing* e que motivaram o desenvolvimento deste trabalho.

O Capítulo 3 descreve o algoritmo SVM, proposto para manutenção incremental em ambientes *warehousing*, e apresenta um comparativo entre o algoritmo proposto e os demais algoritmos de manutenção incremental citados no Capítulo 2.

O Capítulo 4 apresenta os principais aspectos de implementação e um estudo de caso da solução adotada para mostrar o funcionamento do algoritmo SVM. O Capítulo 5 apresenta a conclusão do trabalho. Nele são descritas as principais contribuições do algoritmo e apontadas algumas direções para possíveis trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

Neste capítulo são apresentadas as características e considerações do modelo de ambiente *data warehouse* adotado e utilizado ao longo deste trabalho. Também é apresentada uma revisão bibliográfica dos principias algoritmos de manutenção de visões materializadas para ambientes *data warehousing* existentes na literatura.

2.1. *Data Warehouse*: Arquitetura e Conceitos Relacionados

Warehousing é uma técnica utilizada para recuperar e integrar informações de fontes de dados distribuídas, autônomas e possivelmente heterogêneas. Diversos projetos na área de *warehousing* e visões materializadas são apresentados em [16].

A Figura 2.1 mostra a arquitetura básica de um *data warehouse*. Nesta arquitetura, o *monitor* é responsável por identificar as mudanças ocorridas nos dados das fontes e notificar o *data warehouse*, e o *integrador* deve receber os dados das fontes, integrá-los e manter as visões materializadas.

Esta arquitetura foi implementada no projeto WHIPS - *WareHouse Information Prototype at Stanford* [25]. No escopo deste projeto, Zhuge et al. [27] fazem algumas considerações e definem alguns aspectos sobre um ambiente *data warehousing*, utilizados ao longo deste trabalho e descritos a seguir. Outro modelo de arquitetura de *data warehouse* foi definido por Devlin [5].

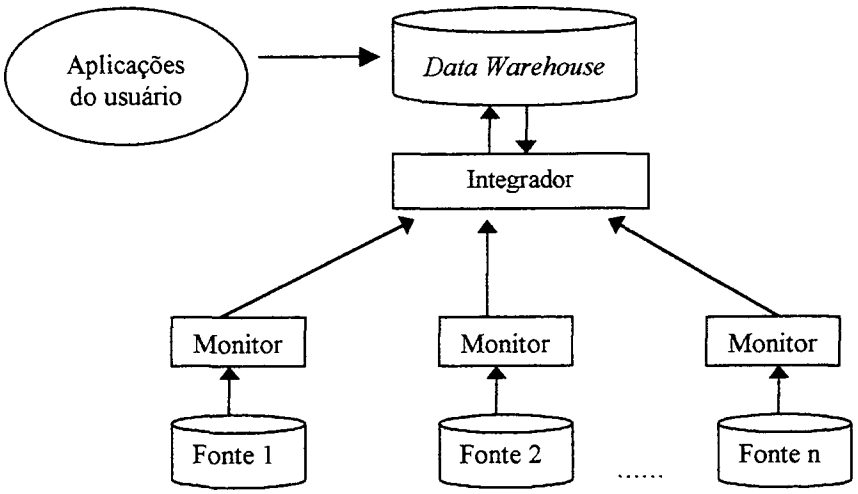

Figura 2.1: Arquitetura de um *data warehouse*

Assume-se que o integrador é fortemente ligado ao *data warehouse*. Portanto, no restante deste trabalho, o termo *warehouse* será utilizado para denotar a combinação entre o integrador e o *data warehouse*.

2.1.1. Cenários de Transação

Os cenários de transação foram definidos com o objetivo de classificar as atualizações que ocorrem nas fontes de dados, sendo eles: *transações de atualização única*, *transações local-fonte* e *transações globais*.

Transações de atualização única

Cada atualização compreende uma transação e é reportada ao *warehouse* separadamente. Desta forma, cada atualização ocorrida nas fontes de dados é processada separadamente pelo *warehouse*.

Transações local-fonte

Uma *transação local-fonte* é uma seqüência de atualizações realizadas em uma mesma fonte de dados que juntas compreendem uma transação. Todas as ações de uma *transação local-fonte* devem ser refletidas atomicamente no *warehouse*. Neste cenário, o *warehouse* processa todas as atualizações executadas em uma fonte de dados através de uma transação como uma

unidade única. Por exemplo, uma transação T_i que engloba *n* atualizações $\{U_1, ..., U_n\}$ ocorridas na fonte de dados *i* é processada com uma unidade única pelo *warehouse*.

Transações globais

Uma *transação global* é uma seqüência de atualizações realizadas em diversas fontes de dados que juntas compreendem uma transação. Todas as ações de uma *transação global* devem ser refletidas atomicamente no *warehouse*. Neste cenário, o *warehouse* processa todas as atualizações executadas em diversas fontes de dados através de uma transação como uma unidade única. Por exemplo, uma transação T_i que engloba *n* atualizações $\{U_1, ..., U_n\}$ ocorridas nas diversas fontes de dados relacionadas com o *warehouse* é processada com uma unidade única pelo *warehouse*.

Para o propósito deste trabalho, assume-se que as atualizações tratadas pelo *warehouse* são *transações de atualização única* e *transações local-fonte*. Atualmente, a maioria dos ambientes *warehousing* suportam cenários de *transações de atualização única*, *transações local-fonte*, ou ambos [28].

2.1.2. Mensagens

Existem dois tipos de mensagens enviadas das fontes de dados para o *warehouse*: reportar uma transação e retornar a resposta referente a uma consulta. Assume-se que as *transações de atualização única* e as *transações local-fonte* são reportadas em uma única mensagem, quando a transação acontece. O *warehouse* envia somente um tipo de mensagem para as fontes de dados: remeter consultas.

Assume-se que a comunicação entre cada fonte de dados e o *warehouse* é confiável e FIFO (*First-in, First-out)*, ou seja, mensagens não são perdidas e são entregues na ordem em que foram enviadas.

Nenhuma restrição é feita sobre a ordem na qual mensagens enviadas por diferentes fontes são entregues ao *warehouse*, pois assume-se que as fontes de dados são autônomas e as atualizações ocorridas em diferentes fontes não são sincronizadas.

2.1.3. Níveis de Consistência

A consistência da manutenção de visões materializadas, para um ambiente *warehousing* com múltiplas fontes de dados, é definida com base nos estados das fontes e do *warehouse*. O estado da fonte é o estado da base de dados da fonte, alterado sempre que a base de dados é atualizada. O estado do *warehouse* é o estado da visão materializada do *warehouse*, modificado sempre que a visão é atualizada. Um algoritmo de manutenção de visões materializadas para ambientes *warehousing* é responsável por trazer a visão para um novo estado consistente depois de processar as mudanças ocorridas nas fontes, desta forma, ele gera uma seqüência de estados *warehouse* enquanto está realizando a manutenção da visão materializada.

Os níveis de consistência dependem da forma como as atualizações das fontes são incorporadas na visão materializada do *warehouse*. Cada nível agrupa todos os níveis antecedentes. Os principais níveis de consistência definidos para ambientes *warehousing* são os seguintes:

- **Convergência**: as mudanças ocorridas nas fontes são eventualmente incorporadas na visão materializada, ou seja, depois da última atualização da visão materializada e quando todas as atividades das fontes tiverem cessado, o estado do *warehouse* é consistente com os estados das fontes;
- **Consistência Forte**: a ordem dos estados de transformação da visão materializada no *warehouse* corresponde à ordem dos estados de transformação dos dados das fontes. Assim, cada estado do *warehouse* reflete um conjunto de estados das fontes e a ordem dos estados do *warehouse* combina com a ordem dos correspondentes estados das fontes;
- **Consistência Completa**: cada estado das fontes de dados é refletido em um estado distinto do *warehouse*. Assim, a ordem dos estados de transformação nas fontes de dados é completamente preservada no *warehouse*.

2.1.4. Visões e Consultas

Assume-se que a visão materializada do *warehouse* é definida através de uma expressão *Project-Select-Join* (PSJ) da álgebra relacional. Assim, se $\{r_1, r_2, ..., r_n\}$ são as *n* relações base

armazenadas nas fontes de dados correspondentes, a visão denotada por V é definida como segue:

$$V = \Pi proj\ (\sigma cond(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n))$$

onde *proj* é um conjunto de nomes de atributos, *cond* é uma expressão condicional, e r_1, r_2, ..., r_n, são as relações base. Qualquer expressão da álgebra relacional construída com as operações *project*, *select* e *join* pode ser transformada em uma expressão equivalente desta forma.

Duas relações base podem residir na mesma fonte de dados ou em fontes distintas. O modelo de dados considerado para as fontes de dados e para o *warehouse* é o modelo de dados relacional.

São considerados dois tipos de atualizações ocorridas nas fontes de dados: inserção e exclusão. Desta forma, a alteração de uma tupla é tratada como uma exclusão da antiga tupla seguida de uma inserção da nova tupla [9].

Quando uma atualização ocorrida na fonte r_i, denominada U_i, é recebida pelo *warehouse*, as mudanças na visão materializada são processadas através de consultas enviadas para as fontes de dados relacionadas. Assim, o *warehouse* gera uma consulta geral, denominada Q, definida como segue:

$$Q = \Pi proj\ (\sigma cond(r_1 \bowtie \ldots \bowtie U_i \bowtie \ldots \bowtie r_n))$$

onde U_i é a tupla t_i atualizada na relação base r_i.

2.2. Algoritmos de Manutenção Incremental

Os algoritmos de manutenção incremental da visão materializada atualizam a visão de forma incremental em resposta às atualizações ocorridas nas fontes de dados. Uma revisão sobre manutenção incremental de visões materializadas pode ser encontrada em [8, 9, 12]. Discussões sobre visões materializadas relacionadas com ambientes *warehousing* podem ser encontradas em [11, 16].

Diversos algoritmos incrementais de manutenção de visões materializadas foram desenvolvidos para ambientes de base de dados centralizados [2, 3, 4, 7, 8, 13, 18, 22, 23]. A maioria destes algoritmos assume que a manutenção da visão é executada por um sistema único que controla todas as relações base, monitorando as atividades das fontes e definindo quais as informações necessárias para atualizar a visão materializada. Estes algoritmos diferem principalmente quanto ao tipo de definição de visão que eles manipulam. Por exemplo, o algoritmo apresentado em [2] considera visões definidas através de expressões *Project-Select-Join* (PSJ) e os algoritmos propostos em [8] trabalham com visões definidas através de qualquer expressão SQL. Alguns destes algoritmos controlam as tuplas duplicadas através das informações dos atributos chave das relações base relacionadas com o *warehouse* [3], enquanto outros armazenam como informação adicional na visão materializada o número de derivações de cada tupla [8]. Estas soluções requerem que os dados das relações base sejam bloqueados enquanto as mudanças são processadas na visão.

Um *warehouse* mantém uma cópia sumarizada dos dados das fontes, desta forma, tem-se um ambiente de banco de dados distribuído com dados replicados [27]. Diversos algoritmos estudam a manutenção de visões em ambientes distribuídos [15, 19, 20, 21], mas estas abordagens requerem que as fontes de dados forneçam informações adicionais, como *timestamps*, para atualizar a visão.

Este trabalho focaliza somente as soluções onde as fontes consideradas são autônomas, podendo ser sistemas não sofisticados ou legados. Assim, não são consideradas abordagens onde é preciso bloquear os dados das fontes enquanto o *warehouse* atualiza as visões ou soluções onde as fontes necessitam fornecer informações adicionais para manter as visões. Nas soluções que serão abordadas, as únicas responsabilidades das fontes são: 1) Notificar o *warehouse* de atualizações relevantes; 2) Responder às consultas enviadas pelo *warehouse*.

Diversos algoritmos de manutenção de visões materializadas para ambientes *warehousing* são apresentados na literatura [1, 26, 27]. Zhuge et al. projetaram duas famílias de algoritmos que oferecem soluções para garantir a consistência dos dados do *warehouse*: a família de algoritmos *Eager Compensating Algorithm* (ECA) [26] e a família de algoritmos *Strobe* [27]. Agrawal et al. [1] desenvolveram os algoritmos SWEEP e Nested SWEEP que são extensões dos algoritmos ECA e Strobe.

2.2.1. Família de Algoritmos ECA

Os algoritmos ECA [26] consideram um cenário restrito, onde todos os dados da visão são provenientes de uma única fonte de dados. A Figura 2.2, extraída de [28], ilustra o processamento de uma mensagem de atualização em um modelo de *data warehouse* com uma única fonte.

A família de algoritmos ECA é formada pelos seguintes algoritmos:

- Algoritmo ECA *(Eager Compensating Algorithm)*;
- Algoritmo ECA-Key (ECA^K);
- Algoritmo ECA-Local (ECA^L).

Os algoritmos da família ECA, implementados no protótipo do sistema WHIPS - *WareHouse Information Prototype at Stanford* [25], proporcionam *consistência forte* em modelos de *data warehouse* com uma única fonte e considerando cenários de *transação de atualização única.*

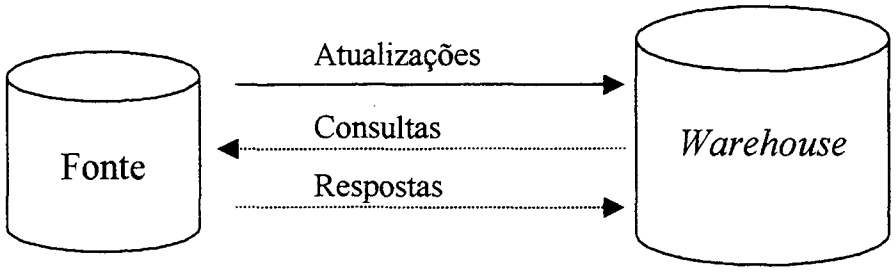

Figura 2.2: Processamento de atualizações em um modelo de *warehouse* com uma fonte

Algoritmo ECA

O ECA (*Eager Compensating Algorithm*) foi desenvolvido com base no algoritmo de manutenção de visões para ambientes de base de dados centralizados [2]. O algoritmo ECA utiliza um conjunto de consultas não respondidas (UQS - *Unanswered Query Set*) para controlar as consultas pendentes, ou seja, as consultas enviadas pelo *warehouse* para a fonte e ainda não respondidas.

Quando o *warehouse* recebe uma notificação de atualização U_i da fonte de dados e o conjunto UQS não está vazio, esta atualização pode fazer com que as consultas Q_j, existentes em UQS, sejam avaliadas incorretamente. Isto ocorre porque as consultas Q_j deveriam ser avaliadas antes da atualização U_i ocorrer, mas estão sendo avaliadas depois de U_i. Assim, as consultas existentes em UQS são processadas considerando um estado da fonte que reflete a atualização U_i. Para resolver este problema, quando o algoritmo ECA gera uma consulta Q_i, em resposta a atualização U_i que ocorreu de forma concorrente ao processamento das consultas Q_j, uma *consulta compensadora* é incorporada em Q_i para cada consulta Q_j existente em UQS. Estas *consultas compensadoras* eliminam os efeitos de U_i refletidos nos resultados das consultas existentes em UQS.

Devido ao uso das *consultas compensadoras*, os resultados das consultas devem ser aplicados na visão materializada somente depois que a resposta a esta consulta e a todas as *consultas compensadoras* relacionadas forem recebidas. O *warehouse* pode apresentar um estado temporariamente inválido se for atualizado para cada consulta recebida. Para evitar estes estados inválidos, o ECA armazena todas as respostas intermediárias em uma relação temporária, denominada COLLECT, e atualiza a visão materializada somente quando UQS estiver vazio (UQS = ∅).

Algoritmo ECA-Key

O algoritmo ECA-Key (ECA^K), um melhoramento do ECA original, assume que a definição da visão inclui os atributos chave na lista de projeção para cada relação base envolvida. As vantagens do ECA^K, sobre o ECA original, são:

- As exclusões são processadas diretamente no *warehouse*, sem a necessidade de enviar consultas para a fonte de dados;
- As inserções são processadas sem a necessidade de incorporar *consultas compensadoras* nas consultas enviadas para a fonte de dados.

Quando uma exclusão U_i é recebida no *warehouse*, nenhuma consulta é enviada para a fonte de dados. Em vez disso, uma ação de exclusão é incorporada diretamente em COLLECT. Desta forma, são excluídas todas as tuplas da visão materializada cujos valores dos atributos, correspondentes aos atributos chave da relação que gerou U_i, sejam iguais aos valores destes atributos em U_i.

Desde que, cada tupla da visão contém os atributos chave para todas as relações base, quando uma tupla *t* de uma relação base é excluída, é possível utilizar os valores dos atributos chave em *t* para identificar quais tuplas na visão foram derivadas utilizando *t*, e que portanto, devem ser excluídas [28].

Desta forma, o ECA^K introduz o conceito de atualizações locais, exclusões são processadas diretamente no *warehouse* sem a necessidade de enviar consultas para a fonte de dados.

A inclusão dos atributos chave de cada relação base na lista de projeção da definição da visão faz com que seja possível executar exclusões diretamente no *warehouse* e elimina a necessidade de incluir *consultas compensadoras* nas consultas enviadas para a fonte no caso de inserções.

Quando uma inserção U_i é recebida no *warehouse*, uma consulta é gerada e enviada para a fonte de dados. Anomalias acontecem devido às atualizações que possam ocorrer de forma concorrente ao processamento desta consulta. As anomalias consistem em tuplas duplicadas ou ausentes na visão materializada. As tuplas duplicadas podem ser identificadas através dos atributos chave, e assim, não são incluídas na visão materializada. As anomalias relacionadas com tuplas ausentes não interferem no resultado final, pois estas tuplas seriam excluídas posteriormente de qualquer forma, durante o processamento das atualizações concorrentes que geraram estas anomalias. Desta forma, elimina-se a necessidade de incorporar *consultas compensadoras* nas consultas enviadas para a fonte de dados.

As respostas das consultas são recebidas e incorporadas em COLLECT, como no algoritmo ECA original. Tuplas duplicadas não são adicionadas ao conjunto COLLECT, pois quando a definição da visão contém os atributos chave para todas as relações base não podem existir tuplas duplicadas. Então, quando uma duplicação é detectada, ela corresponde a uma anomalia e pode ser ignorada. Quando UQS estiver vazio, a visão materializada é atualizada, com base nas ações existentes em COLLECT.

A desvantagem do ECA^K, sobre o ECA original, é a restrição de que a definição da visão materializada deve incluir na lista de projeção os atributos chave de cada relação envolvida.

Algoritmo ECA-Local

O algoritmo ECA^K introduz o conceito de atualizações locais. O algoritmo ECA-Local (ECA^L) combina as *consultas compensadoras* do ECA com as atualizações locais do ECA^K. No ECA^L as atualizações podem ser trabalhadas no *warehouse* localmente ou não localmente. Condições específicas identificam quais atualizações podem ser processadas localmente. Diversos artigos, como [2, 10], descrevem condições nas quais a visão materializada pode ser atualizada sem consultas adicionais às fontes relacionadas, com base na definição da visão ou em uma atualização específica de determinada relação base.

Em situações onde as atualizações não podem ser executadas localmente, o algoritmo ECA original é utilizado, enviando *consultas compensadoras* para as fontes de dados sempre que necessário. No ECA^L nenhuma restrição é feita quanto à definição da visão incluir os atributos chave para todas as relações base envolvidas.

2.2.2. Família de Algoritmos Strobe

Em um cenário que envolve múltiplas fontes de dados, manter a consistência da visão materializada é um processo mais complexo. Quando os dados das fontes são modificados é preciso integrar estes dados com informações das fontes de dados relacionadas, antes de incorporar a mudança na visão materializada do *warehouse*. Durante este processo, mais atualizações podem ocorrer fazendo com que o *warehouse* se torne inconsistente.

Os algoritmos da família Strobe [27] proporcionam *consistência forte*, ou seja, cada estado do *warehouse* reflete um conjunto de estados consistentes das fontes. Cada algoritmo da família Strobe foi projetado para atender a um cenário de transação especifico:

- Algoritmo Strobe: projetado para cenários de *transação de atualização única*;
- Algoritmo Transaction-Strobe: projetado para cenários de *transação local-fonte*;
- Algoritmo Global-Strobe: projetado para cenários de *transação global*.

Os algoritmos Strobe assumem que a definição da visão inclui na lista de projeção os atributos chave para cada relação base envolvida. Este requerimento torna estes algoritmos mais restritivos; entretanto, mais eficientes e fáceis de implementar [28].

Algoritmo Strobe

Assim como os algoritmos ECA, o algoritmo Strobe utiliza o UQS, conjunto de consultas não respondidas, para controlar as consultas pendentes.

Quando o *warehouse* recebe uma mensagem de atualização referente a uma exclusão ocorrida em uma das fontes de dados, o Strobe gera uma ação de exclusão diretamente para as tuplas existentes na visão materializada e relacionadas com a tupla excluída, através da combinação dos valores dos atributos chaves. Como a definição da visão contém os atributos chave para todas as relações base envolvidas com o *warehouse*, quando ocorre a exclusão de uma tupla em uma relação base é possível utilizar os valores dos atributos chave desta tupla para identificar quais as tuplas da visão materializada devem ser excluídas.

Quando a atualização se refere a uma inserção, uma consulta é gerada e enviada para as fontes de dados relacionadas. Para tal, o algoritmo utiliza a função *next_source* que divide a consulta em porções e define qual a próxima fonte a receber a consulta. O critério utilizado para selecionar a próxima fonte para a qual o *warehouse* deve enviar a consulta, é que esta possua uma relação base que possa ser combinada com a expressão condicional da consulta.

A idéia principal do algoritmo Strobe é identificar as atualizações concorrentes que acontecem enquanto uma consulta é processada. Assim, os efeitos destas atualizações podem ser compensados do resultado final da consulta.

Enquanto uma consulta está sendo processada pelas fontes, atualizações podem chegar ao *warehouse* e aplicar o resultado desta consulta na visão materializada pode torná-la inconsistente. Para controlar as atualizações que ocorrem de forma concorrente ao processamento de uma consulta, o algoritmo utiliza o pending(Q), conjunto de atualizações que chegam ao *warehouse* enquanto a consulta Q é processada. Entretanto, somente são inseridas em pending(Q) as atualizações de exclusão. As exclusões que chegam no *warehouse* enquanto existem consultas pendentes são subtraídas do resultado destas consultas. Como o Strobe requer que a definição da visão inclua os atributos chave para cada relação base que participa da visão, tuplas duplicadas existentes não são inseridas na visão materializada. Portanto, as inserções não são incluídas em pending(Q) pois seus efeitos não necessitam ser compensados da resposta de Q, visto que, não interferem no resultado final. Quando o *warehouse* recebe a resposta de uma consulta Q, esta resposta é compensada com base nos dados das atualizações existentes em pending(Q).

As ações de exclusão e os resultados das consultas não são aplicados diretamente na visão materializada do *warehouse*, mas incluídos em uma lista de ações (AL) que controla as operações que efetivamente serão incorporadas no *warehouse*. A visão materializada somente é atualizada, através da execução das ações de AL, quando não existirem consultas sendo processadas pelas fontes de dados relacionadas, ou seja, quando UQS estiver vazio (UQS = ∅). Desta forma, o algoritmo Strobe requer um estado estável do sistema de *data warehouse* para realizar a atualização. Um estado estável do *warehouse* é caracterizado por um período em que não existem consultas sendo processadas e nenhuma nova atualização das fontes é recebida pelo *warehouse*.

O Exemplo 2.1, extraído de [27], ilustra o funcionamento do algoritmo Strobe em um cenário de *data warehouse* com três fontes de dados. Somente as ações executadas pelo *data warehouse* são apresentadas no exemplo. O algoritmo Strobe proporciona *consistência forte* em cenários de *transação de atualização única*.

Algoritmo Transaction-Strobe

O algoritmo Transaction-Strobe (T-Strobe) adapta o algoritmo Strobe para prover *consistência forte* em cenários de *transação local-fonte*. O T-Strobe coleta todas as atualizações executadas através de uma transação e processa estas atualizações como uma unidade única. Agrupar as atualizações de uma transação reduz o número de mensagens de consulta e facilita o processo de atualização do *warehouse*.

O T-Strobe remove todos os pares de inserção e exclusão de uma mesma tupla, existentes na transação a ser processada e definidas nesta ordem: primeiro acontece a inserção e depois a exclusão da tupla. Esta otimização evita o envio de consultas para processar a inserção de uma tupla que posteriormente será excluída.

Depois de receber uma transação T de uma das fontes de dados, o *warehouse* adiciona todas as exclusões existentes em T na lista de ações. Então, o *warehouse* gera uma consulta para processar todas as inserções existentes em T. Como no algoritmo Strobe, os efeitos das atualizações que chegam ao *warehouse*, enquanto uma consulta está sendo processada, são compensados do resultado desta consulta. As ações de AL são aplicadas na visão materializada quando UQS estiver vazio (UQS = ∅), ou seja, quando não existirem consultas sendo processadas pelas fontes de dados.

Exemplo 2.1: Funcionamento do algoritmo Strobe

Seja a visão V definida como $V = r_1 \bowtie r_2 \bowtie r_3$, onde r_1, r_2, r_3 são três relações armazenadas nas fontes *x*, *y* e *z*, respectivamente. O estado inicial dos dados das fontes é o seguinte:

r_1:	A	B
	1	2

r_2:	B	C
	-	-

r_3:	C	D
	3	4

A visão materializada do *warehouse* inicialmente está vazia, ou seja, MV = ∅. São consideradas duas atualizações das fontes:

- U_1 = insert (r_2, [2,3]). Insere a tupla [2,3] na relação r_2.
- U_2 = delete (r_1, [1,2]). Remove a tupla [1,2] da relação r_1.

Ações executadas pelo *data warehouse*:

- A lista de ações é inicializada como vazia (AL = < >).
- Recebe U_1 = insert (r_2, [2,3]) da fonte *y* e gera a consulta $Q_1 = r_1 \bowtie [2,3] \bowtie r_3$. A consulta $Q_1^1 = r_1 \bowtie [2,3]$ é enviada para a fonte *x*.
- Recebe a resposta da consulta Q_1^1 da fonte *x*: $A_1^1 = [1,2,3]$.
- A consulta $Q_2^1 = [1,2,3] \bowtie r_3$ é enviada para a fonte *z*.
- Recebe U_2 = delete (r_1, [1,2]) da fonte *x* e adiciona U_2 em pending(Q_1). Então, uma ação de exclusão é adicionada na lista de ações, resultando em AL = < delete(MV, U_2)>.
- Recebe a resposta da consulta Q_1^2 da fonte *z*: A_1^2= [1,2,3,4]. A_1^2 é a resposta final da consulta Q_1. Como pending(Q_1) contém U_2, o *warehouse* compensa os efeitos de U_2 da resposta A_1^2, resultando na resposta A_1^2 = ∅. Como a resposta A_1^2 está vazia, nada é adicionado na lista de ações (AL).
- Como não existem mais atualizações a serem processadas, atualiza MV aplicando as ações existentes em AL. O resultado final está correto, ou seja, MV = ∅. Os dados da visão materializada são consistentes com relação aos dados das fontes.

Algoritmo Global-Strobe

O algoritmo Global-Strobe (G-Strobe) adapta o algoritmo T-Strobe para prover *consistência forte* em cenários de *transação global*.

O G-Strobe utiliza o conjunto de transações TT, para controlar o conjunto de transações que o *warehouse* recebeu desde a última atualização da visão materializada.

A principal diferença do G-Strobe com relação ao T-Strobe é que ele atualiza a visão materializada somente quando:

- UQS = ∅;
- Para cada transação T_i existente em TT, que depende de outra transação T_j, T_j existe em TT;
- Todas as atualizações das transações em TT foram recebidas e processadas.

Como em um sistema de *data warehouse* os tipos de fontes podem ser diferentes, e conseqüentemente suas capacidades, Zhuge et al. [28] discutem diversas formas para criar o algoritmo G-Strobe, com base na forma de transmissão de uma *transação global* das fontes para o *warehouse*.

O número de mensagens enviadas pelos algoritmos Strobe, T-Strobe e G-Strobe para processar uma atualização é de (n-1) no pior caso, onde n é número de relações base que participam da definição da visão do *warehouse*. Como exclusões são tratadas diretamente no *warehouse*, sem a necessidade de gerar e remeter consultas para as fontes de dados relacionadas, o número de mensagens enviadas para o processamento de múltiplas atualizações é reduzido.

Uma desvantagem dos algoritmos Strobe, T-Strobe e G-Strobe é que eles requerem um estado estável do *warehouse* para atualizar a visão materializada, ou seja, a visão materializada é atualizada somente quando não houver atualizações sendo processadas pelo *warehouse*. Desta forma, a visão materializada não será atualizada enquanto ocorrerem seguidas atualizações nas fontes, ou seja, enquanto UQS não estiver vazio. O algoritmo Complete-Strobe (C-Strobe) foi proposto para contornar este problema [27].

Algoritmo Complete-Strobe

O C-Strobe proporciona *consistência completa* em cenários de *transação de atualização única*, ou seja, cada atualização ocorrida nas fontes é refletida em um estado *warehouse* distinto.

Uma consulta é avaliada para cada inserção ocorrida em uma das fontes de dados e o resultado desta consulta é aplicado no *warehouse*. Entretanto, enquanto uma consulta está sendo processada, outras atualizações podem ocorrer nas fontes de dados e a resposta desta consulta pode conter os efeitos destas atualizações. Os efeitos das atualizações concorrentes devem ser compensados da resposta da consulta, antes de aplicá-la na visão materializada do *warehouse*.

Considerando o caso do processamento da atualização de inserção U_i, o *warehouse* gera e envia uma consulta Q_i para as fontes relacionadas. Entretanto, antes de receber a resposta A_i, referente à consulta Q_i, outras atualizações $(U_{i+1},U_j,...,U_k)$ podem chegar ao *warehouse*. Desta forma, a resposta A_i pode refletir os efeitos destas atualizações concorrentes. Portanto, antes de atualizar a visão materializada com os dados de A_i, o *warehouse* deve compensar os efeitos destas atualizações na resposta A_i.

Se a atualização concorrente for uma inserção, dita U_j, a compensação é realizada através da exclusão das correspondentes tuplas de A_i. Entretanto, se U_j for uma exclusão, é preciso enviar *consultas compensadoras* para as fontes com o objetivo de identificar as tuplas faltantes que devem ser adicionadas em A_i. Durante o processamento das *consultas compensadoras*, novas atualizações podem ocorrer nas fontes de dados. Assim, as respostas destas consultas devem ser compensadas com relação às atualizações que ocorreram de forma concorrente ao seu processamento. Este processo se repete até que todas as *consultas compensadoras* relacionadas com Q_i sejam processadas. Depois de atualizar a visão materializada com relação à atualização U_i, o *warehouse* processa U_{i+1} da mesma forma. Como cada atualização é processada separadamente, não é preciso utilizar a lista de ações.

A principal desvantagem do C-Strobe é o grande número de mensagens de consultas enviadas pelo *warehouse* para cada atualização ocorrida em uma fonte de dados. Assumindo que *n* é o número de relações base relacionadas com o *warehouse* e K é o número máximo de atualizações que podem ocorrer entre o tempo em que uma consulta é enviada e sua resposta é

recebida no *warehouse*. Quando uma inserção U_i é processada, uma consulta Q_i é gerada e enviada para as fontes. Ao receber a resposta A_i pode ser necessário enviar no máximo K *consultas compensadoras* para *n*-2 relações base e para cada uma destas consultas, no máximo K *consultas compensadoras* para *n*-3 relações base podem precisar ser enviadas, e assim por diante. Desta forma, o número total de consultas enviadas é de no máximo K^{n-2}.

O número de *consultas compensadoras* pode ser reduzido agrupando-se as consultas relacionadas com a mesma relação base. Por exemplo, para compensar a consulta Q são enviadas K consultas, onde K é o número máximo de atualizações que podem chegar ao *warehouse* entre o tempo em que uma consulta é enviada e sua resposta é recebida. Entretanto, como existem *n* relações base, é possível agrupar estas consultas em *n*-1 consultas, onde cada consulta agrupada contém todas as consultas geradas para uma mesma relação base. Prosseguindo o processo de agrupar as consultas pelas relações base, o número de consultas adicionais não pode exceder (n-1) x (n-2) x ... x 1 = (n-1)!. Através desta otimização, proposta por Zhuge et al. [27], o número total de consultas enviadas para processar uma atualização é de (n-1)! no pior caso, onde n é o número de relações base que participam da definição da visão do *warehouse*.

Os algoritmos da família Strobe foram implementados no protótipo do sistema WHIPS - *WareHouse Information Prototype at Stanford* [25].

2.2.3. Algoritmos SWEEP e Nested SWEEP

Agrawal et al. [1] desenvolveram os algoritmos SWEEP e Nested SWEEP que são extensões para os algoritmos ECA e Strobe. A principal característica destes algoritmos é que eles utilizam o conceito de *consultas compensadoras*, definido nos algoritmos ECA, para manter as visões materializadas, mas realizam a compensação sem necessitar consultar às fontes de dados. A compensação é efetuada diretamente no *warehouse*, utilizando informações já existentes.

Cada fonte de dados pode armazenar diversas relações base, mas os algoritmos SWEEP e Nested SWEEP assumem que cada fonte possui uma única relação base.

Algoritmo SWEEP

Os algoritmos ECA e Strobe realizam o processamento completo de uma consulta antes de executar qualquer compensação. Desta forma, todas as atualizações que ocorrem durante o tempo em que a consulta está sendo processada são tratadas como concorrentes, mesmo que estas atualizações não interfiram no resultado da consulta. O algoritmo SWEEP compensa os efeitos somente para as atualizações que realmente interferem no resultado da consulta e esta compensação é realizada diretamente no *warehouse*, utilizando os dados disponíveis nas mensagens de atualização enviadas. Este processo é chamado de *correção de erro on-line*.

Quando o *warehouse* recebe a atualização U_i ocorrida na fonte de dados *i*, uma consulta Q_i é gerada e porções desta consulta são enviadas para as fontes relacionadas. Em um ambiente distribuído, uma atualização U_j ocorrida na fonte de dados *j* interfere no resultado da consulta Q_i somente se, U_j ocorre antes do processamento de Q_i pela fonte *j*, ou seja, U_j acontece antes do recebimento da resposta da fonte *j* referente à consulta Q_i. Se, por outro lado, U_j ocorre depois da consulta Q_i ter sido avaliada na fonte *j*, esta atualização não interfere na resposta. Neste caso, a resposta não precisa ser compensada para esta atualização. No algoritmo SWEEP, os efeitos das atualizações concorrentes são compensados, quando necessário, localmente no *warehouse*, utilizando as informações disponíveis nas mensagens de atualização.

Cada mensagem de atualização recebida pelo *warehouse* é armazenada em uma fila de mensagens de atualização. As atualizações existentes na fila de mensagens são processadas seqüencialmente e para cada atualização armazenada nesta fila, uma consulta é gerada e enviada para as fontes de dados relacionadas.

Em um ambiente distribuído, a consulta é dividida em porções e enviada para as fontes de dados relacionadas. Para exemplificar a ordem de processamento das consultas definida pelo SWEEP, assume-se que a visão materializada do *warehouse* é definida através da seguinte expressão:

$$V = (r_1 \bowtie \ldots \bowtie r_{i-1} \bowtie r_i \bowtie r_{i+1} \bowtie \ldots \bowtie r_n)$$

onde $r_1, \ldots, r_{i-1}, r_i, r_{i+1}, \ldots, r_n$ são as *n* relações base armazenadas nas fontes de dados. Supondo que uma atualização U_i da relação r_i é recebida no *warehouse*. As mudanças são processadas

através de consultas enviadas para as fontes de dados relacionadas. Assim, o *warehouse* gera uma consulta geral, denominada Q_i, definida como segue:

$$Q_i = (r_1 \bowtie \ldots \bowtie r_{i-1} \bowtie U_i \bowtie r_{i+1} \bowtie \ldots \bowtie r_n)$$

O algoritmo SWEEP envia primeiramente as porções da consulta Q_i para as relações na direção esquerda da relação r_i onde ocorreu a atualização U_i, considerando a expressão condicional da definição da visão. Depois de enviar a consulta para todas as relações da esquerda, a consulta é enviada para as relações na direção direita de r_i. A consulta é processada iterativamente, como segue: primeiro na direção esquerda de r_i e então prossegue na direção direita de r_i. Desta forma, consulta é primeiramente avaliada em r_{i-1}, depois em r_{i-2}, e assim por diante. O *warehouse* envia a consulta para ser processada em r_{i-1}, e depois de receber a resposta a_{i-1}, continua o processamento enviando a consulta para r_{i-2}. O processamento da consulta acontece da mesma forma na direção direita. A Figura 2.3 representa a ordem de processamento da consulta Q_i.

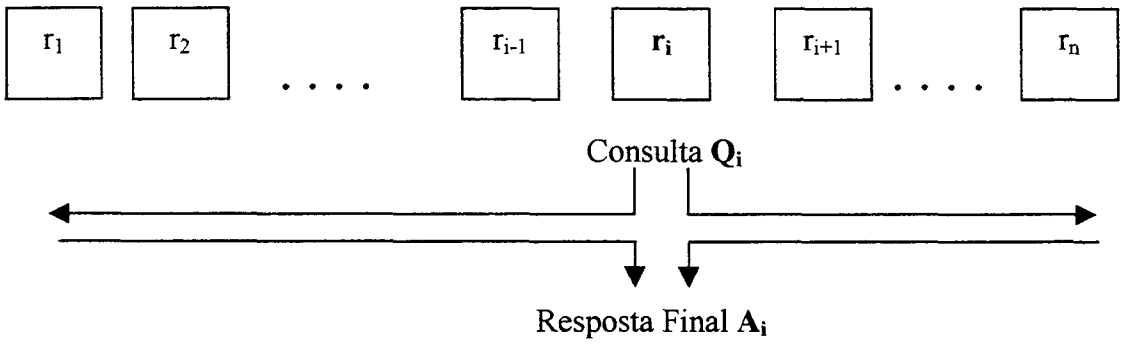

Figura 2.3: Processamento iterativo da consulta

Desta forma, assume-se uma topologia virtual das fontes de dados, definida através da expressão condicional da definição da visão, utilizada para estabelecer a ordem de envio das consultas para as fontes de dados.

Enquanto uma consulta está sendo avaliada, atualizações podem ocorrer nas fontes de dados e as respostas referentes a esta consulta podem conter dados inconsistentes. Por exemplo, quando ocorre uma atualização U_i na relação r_i, a consulta Q_i é gerada e uma porção de Q_i é enviada para ser avaliada primeiramente na relação r_{i-1}. Entretanto, se uma atualização U_{i-1} acontece na relação r_{i-1}, antes da consulta Q_i ser processada em r_{i-1}, é necessário avaliar a

resposta enviada por r_{i-1} e eliminar os possíveis efeitos que U_{i-1} possam ter causado nesta resposta. A compensação é realizada localmente no *warehouse* utilizando os dados existentes na fila de mensagens de atualização. Este procedimento é chamado de *correção do erro on-line* (*on-line error correction*) pois elimina os efeitos das atualizações concorrentes quando elas são detectadas no *warehouse*. Em contraste, os algoritmos Strobe e ECA acumulam estes efeitos até que a consulta seja completamente avaliada por todas as fontes de dados, para então realizar a compensação.

No momento do recebimento da resposta de cada fonte, o algoritmo verifica se existem atualizações concorrentes relacionadas com esta fonte, e compensa os efeitos destas atualizações diretamente na resposta. Por exemplo, depois de processar a consulta na fonte de dados *i*, o algoritmo verifica se durante o período em que a consulta começou a ser processada até o momento em que foi processada por *i*, ocorreram novas atualizações em *i*. O algoritmo compensa os efeitos destas atualizações, se existirem, diretamente na resposta enviada pela fonte *i*, com base nas informações armazenadas na fila de mensagens de atualização. Depois de receber a resposta final referente a uma consulta, o algoritmo atualiza a visão materializada do *warehouse*.

Nenhuma restrição é feita quanto à definição da visão conter os atributos chave para todas as relações base. A multiplicidade das tuplas da visão materializada é mantida através de um campo de controle que armazena o número de derivações possíveis para cada tupla.

As atualizações que ocorrem nas fontes de dados são ordenadas com base na ordem em que estas atualizações são entregues no *warehouse* e a visão materializada é atualizada na ordem destas atualizações.

No SWEEP, o custo de processamento de uma atualização é linear com respeito ao número de relações base, ou seja, somente (n-1) mensagens são necessárias, onde n é o número de relações base que participam da definição da visão do *warehouse*.

O Exemplo 2.2 ilustra o funcionamento do algoritmo SWEEP em um cenário de *data warehouse* com três fontes de dados. Somente as ações executadas pelo *data warehouse* são apresentadas no exemplo.

Exemplo 2.2: Funcionamento do algoritmo SWEEP

Seja a visão V definida como $V = r_1 \bowtie r_2 \bowtie r_3$, onde r_1, r_2, r_3 são três relações armazenadas nas fontes *x*, *y* e *z*, respectivamente. O estado inicial dos dados das fontes é apresentado abaixo:

r_1:

A	B
1	2

r_2:

B	C
-	-

r_3:

C	D
3	4
3	7

A visão materializada do *warehouse* inicialmente está vazia (MV = ∅). São consideradas duas atualizações ocorridas nas fontes de dados relacionadas:

1) U_1 = insert(r_2, [2,3]). Insere a tupla [2,3] na relação r_2.
2) U_2 = delete(r_1, [1,2]). Remove a tupla [1,2] da relação r_1.

Ações executadas pelo *data warehouse*:

- Recebe U_1 da fonte *y* e adiciona U_1 na fila de mensagens.
- U_1 é retirada da fila de mensagens. A consulta $Q_1 = r_1 \bowtie [2,3] \bowtie r_3$ é gerada. A porção $Q_1^1 = r_1 \bowtie [2,3]$ é enviada para a fonte *x*.
- Recebe U_2 da fonte *x* e U_2 é incluída na fila de mensagens.
- Recebe a resposta da consulta Q_1^1 da fonte *x*: $A_1^1 = \emptyset$. Como a atualização U_2 ocorreu na fonte *x* de forma concorrente ao processamento de Q_1, os efeitos de U_2 são eliminados de A_1^1. A atualização U_2 refere-se a uma exclusão, então as tuplas existentes em U_2 devem ser inseridas em A_1^1 ($A_1^1 = A_1^1 + U_2$). Assim, $A_1^1 = \{[1,2,3]\}$.
- A porção $Q_1^2 = [1,2,3] \bowtie r_3$ da consulta é enviada para a fonte *z*.
- Recebe a resposta da consulta Q_1^2 da fonte *z*: $A_1^2 = \{[1,2,3,4],[1,2,3,7]\}$.
- A_1^2 é a resposta final de A_1. As ações de A_1 são aplicadas em MV (MV = MV + insert(MV,A_1)), resultando em MV = {[1,2,3,4],[1,2,3,7]}. Os dados da visão materializada são consistentes com relação aos dados das fontes no momento em que ocorreu a atualização U_1.
- U_2 é retirada da fila de mensagens. A consulta $Q_2 = [1,2] \bowtie r_2 \bowtie r_3$ é gerada e a porção $Q_2^1 = [1,2] \bowtie r_2$ da consulta é enviada para a fonte *y*.
- Recebe a resposta de Q_2^1 da fonte *y*: $A_2^1 = \{[1,2,3]\}$.
- A porção $Q_2^2 = [1,2,3] \bowtie r_3$ da consulta é enviada para a fonte *z*.
- Recebe a resposta da consulta Q_2^2 da fonte *z*: $A_2^2 = \{[1,2,3,4]\ [1,2,3,7]\}$.
- Como A_2^2 é a resposta final de A_2, A_2 é aplicada em MV, ou seja, MV = MV + delete(MV,A_2). O resultado final é MV = ∅. O estado do *warehouse* é consistente com relação ao estado das fontes no momento em que ocorreu a atualização U_2.

Em contraste com os algoritmos Strobe, o SWEEP não requer um estado estável do *warehouse* para incorporar as mudanças na visão. O algoritmo SWEEP proporciona *consistência completa* para cenários de *transação de atualização única.*

Algoritmo Nested SWEEP

O algoritmo Nested SWEEP proporciona *consistência forte* para cenários de *transação de atualização única.* O Nested SWEEP é um algoritmo de manutenção de visão recursivo que processa as mudanças na visão materializada para múltiplas atualizações das fontes coletivamente.

O Nested SWEEP compartilha componentes dos resultados das consultas da atualização que está sendo processada com as atualizações subseqüentes. Quando uma atualização concorrente é detectada durante o processamento de uma consulta, o processamento é interrompido e a parte já avaliada desta consulta é processada recursivamente para a atualização concorrente. Então, o processamento retorna para a consulta original, depois de modificá-la para incluir os itens relevantes da atualização concorrente. Desta forma, a parte já avaliada de uma consulta é incorporada recursivamente para as atualizações que acontecem de forma concorrente à consulta que está sendo processada.

Apesar do Nested SWEEP não exigir que o recebimento de atualizações das fontes seja totalmente interrompido para atualizar a visão materializada, ele necessita de um período durante o qual não aconteçam atualizações efetivamente concorrentes para a terminação do processo de atualização.

O número de mensagens enviadas pelo Nested SWEEP para o processamento de uma atualização é de (n-1) no pior caso, onde n é o número de relações base envolvidas com o *warehouse.* Quando múltiplas atualizações ocorrem de forma concorrente, este algoritmo processa estas atualizações coletivamente. Desta forma, o número de mensagens é reduzido para estas múltiplas atualizações. Maiores detalhes sobre o algoritmo Nested SWEEP podem ser encontrados em [24].

2.3. Considerações Finais

Este capítulo apresentou um resumo dos principais algoritmos de manutenção de visões materializadas para ambientes *warehousing* existentes na literatura, sendo eles:

- Algoritmos da família ECA: ECA, ECA^K e ECA^L;
- Algoritmos da família Strobe: Strobe, T-Strobe, G-Strobe e C-Strobe;
- Algoritmos SWEEP e Nested SWEEP.

Estes algoritmos foram desenvolvidos com base nos aspectos definidos no modelo de *data warehouse* apresentado.

A idéia básica dos algoritmos ECA é adicionar *consultas compensadoras* na consulta enviada para a fonte, com o objetivo de compensar os efeitos das atualizações concorrentes. Os algoritmos ECA^K e ECA^L são melhoramentos do algoritmo ECA original. O ECA^K introduz o conceito de atualizações locais e o ECA^L combina as *consultas compensadoras* do ECA com as atualizações locais do ECA^K.

Os algoritmos da família Strobe foram projetados para atender a diferentes cenários de transação:

- Strobe: projetado para cenários de *transação de atualização única*;
- T-Strobe: projetado para cenários de *transação local-fonte*;
- G-Strobe: projetado para cenários de *transação global*.

Os algoritmos Strobe assumem que a definição da visão inclui os atributos chave para cada relação base envolvida, o que torna estes algoritmos mais restritivos, entretanto, mais eficientes. Os algoritmos da família Strobe proporcionam *consistência forte* em cenários de transação que envolvem múltiplas fontes de dados.

A idéia principal dos algoritmos Strobe é identificar as atualizações concorrentes que acontecem enquanto determinada consulta é processada. Assim, os efeitos destas atualizações podem ser eliminados do resultado final da consulta. O principal problema destes algoritmos é que a visão materializada não pode ser atualizada até que o sistema de *data warehouse* esteja estável, ou seja, o algoritmo espera até que todas as atualizações cessem e somente então as respostas resultantes de todas as atualizações ocorridas nas fontes são incorporadas na visão

materializada. A visão materializada nunca será atualizada se não houver um período de estabilidade no sistema.

Zhuge et al. [27] propuseram o algoritmo C-Strobe para contornar a necessidade de um período de estabilidade no sistema de *data warehouse*. No C-Strobe, cada atualização é avaliada completamente antes de processar as atualizações subseqüentes. O C-Strobe provê *consistência completa*, ou seja, o estado da visão materializada reflete cada estado de transição das fontes. A principal desvantagem do C-Strobe é que o número de mensagens enviadas para processar cada atualização é muito alto.

No algoritmo SWEEP o estado do *warehouse* preserva a ordem de entrega das atualizações ocorridas nas fontes, através da ordenação das mensagens de atualização recebidas. As atualizações que ocorrem nas fontes de dados são ordenadas com base na ordem em que estas atualizações são entregues no *warehouse* e a visão materializada é atualizada na ordem destas atualizações. Desta forma, o SWEEP oferece *consistência completa* para cenários de *transação de atualização única*. O nível de *consistência completa* é considerado muito forte para a maioria dos cenários de *data warehouse* atuais.

O algoritmo Nested SWEEP, uma extensão do algoritmo SWEEP, proporciona *consistência forte* para cenários de *transação de atualização única*. O Nested SWEEP processa as mudanças na visão materializada para múltiplas atualizações das fontes coletivamente, compartilhando componentes dos resultados das consultas da atualização que está sendo processada com as atualizações subseqüentes. O Nested SWEEP requer um período em que não ocorram atualizações concorrentes nas fontes de dados para atualizar a visão materializada.

O algoritmo SVM (*Algorithm for **S**cheduling Warehouse **V**iew **M**aintenance*), apresentado no próximo capítulo, é uma abordagem proposta para garantir a consistência dos dados do *warehouse* que combina os principais fundamentos dos algoritmos apresentados neste capítulo. A principal vantagem do algoritmo proposto é a possibilidade de realizar a manutenção do *warehouse* periodicamente, em intervalos de tempo previamente definidos. Desta forma, não é preciso esperar por um período estável no sistema de *data warehouse*, onde não ocorram mudanças nos dados das fontes, para atualizar a visão materializada. O capítulo seguinte apresenta o algoritmo SVM e faz um comparativo do algoritmo proposto com os demais algoritmos estudados neste capítulo.

Capítulo 3

Algoritmo SVM

O algoritmo SVM (*Algorithm for Scheduling Warehouse View Maintenance*) reúne os principais aspectos positivos dos algoritmos apresentados no Capítulo 2 e visa a realizar a manutenção incremental de visões materializadas em ambientes *warehousing* de forma programada. A proposta do SVM é atualizar a visão materializada periodicamente e incorporar um conjunto de mudanças das fontes a cada atualização.

Este capítulo descreve o algoritmo SVM, seu contexto de aplicação e apresenta os fundamentos utilizados como base para seu desenvolvimento. Também é apresentado um comparativo do algoritmo proposto com os demais algoritmos citados no Capítulo 2.

3.1. Fundamentos Utilizados no Desenvolvimento do SVM

O algoritmo SVM, proposto para manutenção incremental de visões materializadas em ambientes *warehousing*, está fortemente baseado e utiliza os principais fundamentos dos algoritmos Strobe e SWEEP. O modelo de *data warehouse* apresentado por Zhuge et al. [27, 28] foi adotado como base para desenvolvimento do SVM.

O algoritmo proposto utiliza o conceito de lista de ações definido por Zhuge et al. [27]. A lista de ações armazena as operações que serão incorporadas no *warehouse* para a atualização da visão materializada.

Assim como os algoritmos Strobe, o algoritmo SVM assume que a definição da visão inclui na lista de projeção os atributos chave para cada relação base que participa do *warehouse*. Através desta restrição é possível incorporar as operações de exclusão diretamente na lista de ações, sem necessitar gerar e enviar consultas para as fontes de dados relacionadas, o que reduz o número de mensagens necessárias para realizar a atualização da visão em resposta a um conjunto de atualizações das fontes.

A fila de mensagens de atualização definida no algoritmo SWEEP [1] é utilizada pelo algoritmo SVM. As mensagens de atualização recebidas pelo *warehouse* são armazenadas nesta fila e processadas seqüencialmente.

O SVM compensa os efeitos das atualizações que interferem no processamento de uma consulta diretamente no *warehouse*, utilizando os dados disponíveis na fila de mensagens de atualização. Este processo é similar ao processo de *correção de erro on-line* presente nos algoritmos SWEEP [1].

O algoritmo SVM utiliza a expressão condicional da definição da visão para estabelecer a ordem de envio das consultas para as fontes de dados. Esta forma de processamento iterativo das consultas foi definida e utilizada pelos algoritmos SWEEP e Nested SWEEP [1].

3.2. Proposição do Algoritmo SVM

O algoritmo SVM, apresentado na Figura 3.1, possui dois módulos: o monitor do *warehouse* e o monitor da fonte de dados. A Tabela 3.1, ordenada por ordem alfabética dos termos, apresenta a definição dos termos que ilustram os conceitos, variáveis e funções utilizados na descrição do algoritmo proposto. Muitos desses termos encontram-se definidos em [1, 27]. Cada fonte de dados pode armazenar qualquer número de relações base, mas para simplificar a descrição do algoritmo assume-se que cada fonte possui somente uma relação.

Quando acontece uma atualização nos dados da fonte, é responsabilidade do monitor da fonte notificar o *warehouse* através do envio de uma mensagem de aviso de atualização. Outra função do monitor da fonte é processar consultas enviadas pelo *warehouse*, e retornar o resultado do processamento destas consultas ao *warehouse*. Deve existir um monitor para cada fonte de dados envolvida com a visão materializada do *warehouse*.

Algoritmo SVM

Monitor da fonte de dados:

- Quando ocorre uma atualização U_i, envia U_i para o *warehouse*.
- Quando recebe uma consulta Q_i, processa a consulta Q_i e envia a resposta A_i ao *warehouse*.

Monitor do *Warehouse*:

Recebe Mensagens:

- Quando recebe uma atualização U_i da fonte *i*, adiciona U_i na fila MQ e em SUMQ(i).

Processa Mensagens:

- Para todo Ui $\in$ MQ, retira U_i de MQ e de SUMQ(i). Processa cada atualização U_i existente em MQ conforme ordem de chegada.
  - Se U_i for uma exclusão
    - AL = AL + delete(MV,U_i).
  - Se U_i for uma inserção
    - Seja $Q_i = Q<U_i>$, gera Q_i com base nas informações da atualização U_i.
    - Para cada fonte *j* localizada à esquerda da fonte *i*
      - Q_i^j = *ParseQuery*(Q_i, *j*).
      - Envia Q_i^j para a fonte *j*.
      - Após receber a resposta A_i^j da fonte *j*
        - Se $\exists U_k \in$ SUMQ(j), então
          - $\forall U_k$, tal que U_k é uma inserção, faça ($A_i^j = A_i^j - U_k$).
          - $\forall U_k$, tal que U_k é uma exclusão, faça ($A_i^j = A_i^j + U_k$).
      - $A_i = A_i + A_i^j$.
    - Para cada fonte *j* localizada à direita da fonte *i*
      - Q_i^j = *ParseQuery*(Q_i, *j*).
      - Envia Q_i^j para a fonte *j*.
      - Após receber a resposta A_i^j da fonte *j*:
        - Se $\exists U_k \in$ SUMQ(j), então
          - $\forall U_k$, tal que U_k é uma inserção, faça ($A_i^j = A_i^j - U_k$).
          - $\forall U_k$, tal que U_k é uma exclusão, faça ($A_i^j = A_i^j + U_k$).
      - $A_i = A_i + A_i^j$.
    - AL = AL + insert(MV, A_i).

Atualiza o Data Warehouse:

- AL é incializada como uma lista vazia, AL = < >.
- Se *CountTime*( ) for igual a TIME, (*CountTime* = TIME), faça
  - MV = MV + AL, aplica as ações de AL em MV
- Define AL = < >, ou seja, define a lista de ações como vazia.

fim do algoritmo.

Figura 3.1: Algoritmo SVM

Tabela 3.1: Terminologia utilizada no algoritmo SVM

Termo	*Definição*
A_i	Resposta referente a consulta Q_i.
AL	Lista de ações a serem aplicadas no *warehouse*.
CountTime( )	Função que controla a ocorrência de um intervalo de tempo.
insert(MV, U_i)	Função que insere a tupla U_i na visão materializada.
delete(MV, U_i)	Função que exclui a tupla U_i da visão materializada.
MQ	*Message Queue*, fila que armazena o conjunto de mensagens de atualização recebidas pelo *warehouse*.
MV	Visão materializada armazenada no *warehouse*.
ParseQuery(Q_i,j)	Função que retorna a porção Q_i^j da consulta Q_i, a ser avaliada pela fonte *j*.
Q_i	Consulta gerada devido a atualização U_i ocorrida em uma das fontes de dados relacionadas.
$Q<U_i>$	$Q<U_i>$ - denota a expressão da definição da visão V, com a relação base onde ocorreu U_i substituída pela tupla referente à U_i.
TIME	Variável que armazena o valor do intervalo de tempo estabelecido para atualizar a visão materializada.
SUMQ(i)	*Source Update Message Queue*, conjunto de mensagens de atualização ocorridas na fonte *i*. O conjunto SUMQ(i) é um subconjunto do conjunto MQ (SUMQ(i) $\subset$ MQ).
U_i	Atualização ocorrida em uma das fontes de dados relacionadas com o *warehouse*.

Ao receber uma mensagem de aviso de atualização de uma das fontes de dados, o monitor do *warehouse* deve incluí-la na fila de mensagens *Message Queue* (MQ). As mensagens existentes em MQ são retiradas da fila uma a uma e processadas seqüencialmente.

Quando a mensagem refere-se a uma exclusão, o algoritmo adiciona uma ação de exclusão na lista de ações (AL). Para mensagens de inclusão, o monitor do *warehouse* deve gerar uma consulta a ser avaliada pelas fontes de dados relacionadas.

O processamento da consulta é executado da seguinte forma: seja U_i uma atualização ocorrida na relação base *i* e seja Q_i a consulta gerada para processar as mudanças a serem incorporadas na visão materializada referentes à U_i. A função *ParseQuery* divide a consulta Q_i em porções a serem enviadas para as fontes de dados relacionadas. A consulta é processada iterativamente, como segue: primeiro as porções de Q_i são avaliadas pelas relações base localizadas na direção esquerda de *i* e então o processamento prossegue nas relações base

localizadas na direção direita de *i*, considerando a expressão condicional da definição da visão. Assim, primeiramente Q_i é avaliada na relação *i*-1, depois em *i*-2, até que todas as relações da esquerda tenham avaliado a consulta. Então, a consulta é processada na direção direita da mesma forma. Assume-se uma localização virtual das fontes de dados estabelecida com base nas informações de cada fonte que podem ser unidas com dados de outras fontes.

Enquanto uma consulta está sendo processada pelas fontes de dados, novas atualizações podem ocorrer. Para eliminar o efeito das atualizações concorrentes, o monitor do *warehouse* ao receber a resposta de uma fonte de dados, referente à porção de uma consulta avaliada, verifica se não ocorreram novas atualizações nesta fonte do instante em que a consulta foi gerada até o momento em que a consulta foi processada pela fonte. Para verificar a existência de atualizações concorrentes é utilizado o conjunto SUMQ(j) que guarda as atualizações ocorridas na fonte de dados *j* e que ainda não foram processadas pelo *warehouse*. Por exemplo, ao receber a resposta A_r^j, referente à consulta Q_r^j avaliada pela fonte *j*, o monitor do *warehouse* deve verificar se existem atualizações concorrentes em SUMQ(j) e então eliminar os efeitos destas atualizações da resposta A_r^j. A compensação é realizada localmente no *warehouse* através das informações existentes na fila de mensagens, sem a necessidade de gerar e enviar novas consultas para as fontes de dados relacionadas.

Os resultados das consultas e as ações de exclusão não são aplicados diretamente no *warehouse*. Em vez disso, o algoritmo armazena todas as ações a serem executadas na visão materializada na lista de ações. Desta forma, é possível realizar uma manutenção programada da visão materializada, através da definição de intervalos de tempo para aplicar as ações de AL no *warehouse*.

O processo de atualização da visão materializada é realizado periodicamente pela função *CountTime*, que utiliza como parâmetro o valor do intervalo de tempo a ser considerado para incorporar as mudanças das fontes no *warehouse*, armazenado na variável TIME. A periodicidade da manutenção da visão materializada é definida pelo usuário do sistema de *data warehouse* conforme as especificações e requisitos do sistema.

Os processos de recebimento de mensagens, processamento de mensagens e atualização do *data warehouse* são executados em paralelo pelo monitor do *warehouse*.

O algoritmo SVM proporciona *consistência forte* em cenários de *transação de atualização única*, ou seja, o estado do *warehouse* reflete um conjunto de estados consistentes das fontes de dados. O SVM pode ser facilmente adaptado para cenários de *transação local-fonte*.

O Exemplo 3.1 ilustra o funcionamento do algoritmo SVM em um cenário de *data warehouse* com três fontes de dados. Somente as ações executadas pelo monitor do *warehouse* são apresentadas no exemplo.

No exemplo, a atualização U_2 acontece de forma concorrente ao processamento da consulta Q_1, pois U_2 ocorre na fonte de dados x antes da consulta Q_1 ser avaliada por esta fonte. O SVM compensa os efeitos de U_2 sobre o processamento de Q_1 utilizando as informações existentes em MQ. Da mesma forma, os efeitos da atualização concorrente U_3 são compensados da resposta de Q_2. O exemplo mostra que utilizando o algoritmo SVM, as mudanças ocorridas nas fontes de dados são incorporadas periodicamente na visão materializada do *warehouse* de forma consistente. Os efeitos causados pelas atualizações que acontecem de forma concorrente ao processamento das consultas são compensados e o estado da visão materializada depois de incorporar as ações existentes em AL é consistente com o estado das fontes de dados.

O intervalo de tempo definido para a atualizar a visão é variável e depende das configurações do sistema de *data warehouse*, entretanto, sempre que a visão materializada for atualizada ela deve refletir somente as mudanças ocorridas nas fontes de dados e já processadas pelo *warehouse*. Desta forma, considerando a atualização U_i, o algoritmo SVM assegura que os efeitos de todas as atualizações que chegaram antes de U_i serão refletidos na visão materializada, mas nenhum efeito das atualizações que chegaram depois de U_i será incluído.

O Exemplo 3.1 mostra que o *warehouse* pode ser atualizado a qualquer momento, pois somente serão inseridos os efeitos das mensagens já processadas. Por exemplo, incorporando as ações de AL na visão materializada depois de processar U_2, mas antes de processar U_3, somente as mudanças referentes às atualizações U_1 e U_2 são incluídas e nenhum efeito da atualização U_3 é adicionado. Neste caso, o resultado a atualização da visão materializada é MV = {[1,2,3,4],[1,2,3,7] [8,2,3,4],[8,2,3,7]}. O estado do *warehouse* é consistente com o conjunto de estados das fontes considerando as atualizações U_1 e U_2.

Exemplo 3.1: Funcionamento do algoritmo SVM

Seja a visão V definida como $V = r_1 \bowtie r_2 \bowtie r_3$, onde r_1, r_2, r_3 são três relações armazenadas nas fontes *x*, *y* e *z*, respectivamente. O estado inicial dos dados das fontes é apresentado abaixo:

r_1: A	B
1	2

r_2: B	C
-	-

r_3: C	D
3	4
3	7

A visão materializada do *warehouse* inicialmente está vazia (MV = ∅). A lista de ações é inicializada como vazia (AL = < >) e a variável TIME é inicializada com o intervalo de tempo definido para atualizar MV.

São consideradas três atualizações ocorridas nas fontes de dados relacionadas: 1) U_1 = insert(r_2, [2,3]); 2) U_2 = insert(r_1, [8,2]); 3) U_3 = delete(r_3, [3,7]).

Ações executadas pelo monitor do *warehouse*:

- Recebe U_1 da fonte *y*, adiciona U_1 em MQ e em SUMQ(y).
- U_1 é retirada de MQ e de SUMQ(y). A consulta $Q_1 = r_1 \bowtie [2,3] \bowtie r_3$ é gerada e a porção $Q_1^1 = r_1 \bowtie [2,3]$ é enviada para a fonte *x*.
- Recebe U_2 da fonte *x* e U_2 é incluída em MQ e em SUMQ(x).
- Recebe a resposta da consulta Q_1^1 da fonte *x*: A_1^1 ={[1,2,3],[8,2,3]}. Como SUMQ(x) contém U_2, os efeitos de U_2 precisam ser eliminados de A_1^1. A atualização U_2 refere-se a uma inserção, então as tuplas existentes em U_2 devem ser removidas de A_1^1 ($A_1^1 = A_1^1 - U_2$). Assim, A_1^1 = {[1,2,3]}.
- A porção $Q_1^2 = [1,2,3] \bowtie r_3$ da consulta é enviada para a fonte *z*.
- Recebe a resposta da consulta Q_1^2 da fonte *z*: A_1^2 = {[1,2,3,4],[1,2,3,7]}. Como SUMQ(z) está vazio nenhum procedimento é executado.
- Como A_1^2 é a resposta final de A_1, as ações de A_1 são inseridas em AL (AL = AL + insert(MV,A_1)).
- U_2 é retirada de MQ e de SUMQ(x). A consulta $Q_2 = [8,2] \bowtie r_2 \bowtie r_3$ é gerada e a porção $Q_2^1 = [8,2] \bowtie r_2$ da consulta é enviada para a fonte *y*.
- Recebe a resposta de Q_2^1 da fonte *y*: A_2^1 ={[8,2,3]}. Como SUMQ(y) está vazio nenhum procedimento é executado.
- A porção $Q_2^2 = [8,2,3] \bowtie r_3$ da consulta é enviada para a fonte *z*.
- Recebe U_3 da fonte *z* e U_3 é incluída em MQ e em SUMQ(z).
- Recebe a resposta da consulta Q_2^2 da fonte *z*: A_2^2 = {[8,2,3,4]}. Como SUMQ(z) contém U_3, os efeitos de U_3 precisam ser eliminados de A_2^2. A atualização U_3 refere-se a uma exclusão, então as tuplas existentes em U_3 devem ser inseridas de A_2^2 ($A_2^2 = A_2^2 + U_3$).
  Assim, A_2^2 = {[8,2,3,4],[8,2,3,7]}.
- Como A_2^2 é a resposta final de A_2, as ações de A_2 são aplicadas em AL, ou seja, AL = AL + insert(MV,A_2).
- U_3 é retirada de MQ e de SUMQ(z). Como U_3 é uma exclusão, uma ação de exclusão relacionada a U_3 é inserida em AL, ou seja, AL = AL + delete(MV,U_3).
- Supondo que *CountTime*( ) = TIME, então as ações de AL são aplicadas em MV, ou seja, MV = MV + AL. O resultado final é MV = {[1,2,3,4][8,2,3,4]}.
- AL é reinicializada (AL = < >).

A principal vantagem do SVM é a possibilidade de se determinar o momento em que as mudanças ocorridas nos dados das fontes serão incorporadas na visão materializada do *warehouse*. O período de tempo a ser considerado para realizar a manutenção da visão é definido pelo usuário do sistema de *data warehouse* e irá depender das características específicas do sistema e das necessidades do usuário. O usuário pode atualizar o *warehouse* diariamente ou semanalmente, por exemplo, conforme as especificações do sistema. Isto torna o algoritmo mais flexível, podendo ser adaptado em diferentes sistemas de *data warehouse*.

3.3. Comparação do Algoritmo SVM com os demais Algoritmos

O algoritmo SVM não pode ser comparado diretamente com o algoritmo ECA, visto que, o ECA atende a um cenário de *data warehouse* bastante restrito, onde a visão materializada está relacionada com uma única fonte de dados.

Em um cenário de *data warehouse* mais abrangente, com diversas fontes de dados relacionadas, o processo de manter a consistência dos dados da visão torna-se mais complexo. Os algoritmos Strobe, T-Strobe, SWEEP, Nested SWEEP e SVM são aplicados em cenários de *data warehouse* onde existem diversas fontes de dados relacionadas e cada atualização ocorrida nas fontes de dados é reportada separadamente ao *warehouse* ou reportada através de uma transação que engloba diversas atualizações ocorridas em uma mesma fonte de dados.

Os algoritmos SWEEP e C-Strobe proporcionam *consistência completa* em cenários de *transação de atualização única*. *Consistência completa* determina que cada estado do *warehouse* deve refletir um estado de transição das fontes de dados, ou seja, cada atualização ocorrida em uma fonte é refletida em um estado distinto da visão materializada. Entretanto, *consistência completa* pode ser um requisito muito forte e desnecessário na maioria dos cenários de *data warehouse*, pois a visão materializada é atualizada para cada mudança ocorrida nas fontes de dados [27]. Os algoritmos Strobe, T-Strobe, Nested SWEEP e SVM proporcionam *consistência forte*, um requisito desejável na maioria dos cenários de *data warehouse* atuais. *Consistência forte* requer que a visão materializada incorpore o efeito de várias atualizações das fontes coletivamente [28].

A vantagem do SWEEP sobre o algoritmo C-Strobe, considerando que ambos oferecem *consistência completa*, é que no SWEEP o número de mensagens envolvidas no

processamento de uma atualização é significativamente menor. No C-Strobe o número total de consultas enviadas para processar uma atualização é de (n-1)! no pior caso, contra (n-1) mensagens necessárias utilizando o SWEEP, onde n é o número de relações base que participam da definição da visão do *warehouse*.

A principal vantagem do algoritmo SVM sobre os outros algoritmos que oferecem *consistência forte*, é a possibilidade de determinar quando a visão materializada do *warehouse* deve ser atualizada. Nos algoritmos Strobe e T-Strobe a visão materializada é atualizada somente quando não existem consultas sendo processadas, portanto, se ocorrerem seguidas modificações nas fontes de dados o *warehouse* não será atualizado. Para incorporar na visão materializada as mudanças ocorridas nas fontes, o Nested SWEEP requer que não existam atualizações efetivamente concorrentes sendo processadas, o que é um requisito mais suave que o exigido pelo Strobe e T-Strobe. Desta forma, é possível dizer que os algoritmos Strobe, T-Strobe e Nested SWEEP dependem de algum fator determinante, relacionado ao volume de mensagens de atualização recebidas, para incorporar na visão materializada as mudanças ocorridas nas fontes de dados. Utilizando o algoritmo SVM é possível atualizar a visão em períodos de tempo previamente definidos, sem necessitar que o recebimento de mensagens de atualização seja interrompido por um período de tempo no *warehouse*. Outro benefício da manutenção programada é que o usuário do sistema de *data warehouse* é quem determina a periodicidade de manutenção da visão materializada.

Os algoritmos SWEEP, Nested SWEEP e SVM realizam a compensação das anomalias somente para as atualizações efetivamente concorrentes, ou seja, para as atualizações que ocorreram na fonte de dados i antes da consulta que está sendo processada no *warehouse* ser avaliada na fonte i. Isto não acontece com os algoritmos Strobe que avaliam totalmente o resultado de uma consulta antes de executar qualquer compensação. Utilizando os algoritmos Strobe, a compensação é realizada para todas as atualizações que ocorreram durante o período de processamento da consulta e não somente para as atualizações que realmente interferem no resultado da consulta.

Uma restrição dos algoritmos Strobe e SVM é que a definição da visão deve incluir os atributos chave para cada relação base que participa do *warehouse*. Entretanto, esta restrição permite reduzir o número de mensagens necessárias para processar as mudanças na visão, visto que, as operações de exclusão são executadas diretamente na visão materializada do *warehouse*, sem a necessidade de gerar e enviar consultas para as fontes de dados

relacionadas. A maioria das aplicações inclui os atributos chave para as relações base, entretanto, para as aplicações onde os atributos chave não fazem parte da lista de projeção da definição da visão é possível adicionar estes atributos através do *warehouse* [27].

Os algoritmos ECA, Strobe, C-Strobe, SWEEP, Nested SWEEP e SVM foram projetados para cenários de *transação de atualização única*. O algoritmo T-Strobe foi projetado para cenários de *transação local-fonte* e o algoritmo G-Strobe para cenários de *transação global*. Entretanto, os algoritmos SWEEP, Nested SWEEP e SVM podem ser facilmente adaptados para trabalhar com *transações local-fonte*.

A Tabela 3.2 apresenta um resumo do comparativo das propriedades dos algoritmos de manutenção incremental de visões materializadas para ambientes *warehousing*.

Tabela 3.2: Comparativo dos algoritmos de manutenção de visões materializadas

Algoritmo	Número de fontes	Nível de Consistência	Custo de Mensagens (p/ atualização)	Cenários de Transação	Visão inclui atributos chave	Manutenção da visão materializada
ECA	1	*forte*	O(1)	*atualização única*	SIM	Requer um estado estável do sistema de *data warehouse*
Strobe	*n*	*forte*	O(*n*)	*atualização única*	SIM	Requer um estado estável do sistema de *data warehouse*
T-Strobe	*n*	*forte*	O(*n*)	*local-fonte*	SIM	Requer um estado estável do sistema de *data warehouse*
G-Strobe	*n*	*forte*	O(*n*)	*global*	SIM	Requer um estado estável do sistema de *data warehouse*
C-Strobe	*n*	*completa*	O(*n*!)	*atualização única*	SIM	Atualiza a visão depois de cada atualização processada
SWEEP	*n*	*completa*	O(*n*)	*atualização única*	NÃO	Atualiza a visão depois de cada atualização processada
Nested SWEEP	*n*	*forte*	O(*n*)	*atualização única*	NÃO	Requer um período em que cessem as atualizações concorrentes
SVM	*n*	*forte*	O(*n*)	*atualização única*	SIM	Manutenção Programada

3.4. Considerações Finais

Este capítulo apresentou a descrição do algoritmo SVM proposto para manutenção incremental de visões materializadas em ambientes *warehousing*. O SVM é fortemente baseado nos fundamentos definidos nos algoritmos Strobe e SWEEP, como: lista de ações, fila de mensagens de atualização, *correção de erro on-line* e processamento iterativo da consulta.

O algoritmo SVM, assim como os algoritmos Strobe, é restritivo pois requer que a definição da visão inclua os atributos chave para cada relação base envolvida com o *warehouse*. Entretanto, esta restrição torna os algoritmos mais eficientes, pois as exclusões podem ser executadas localmente no *warehouse*.

Os algoritmos SWEEP e Nested SWEEP proporcionam *consistência completa*, um requisito muito forte para a maioria dos sistemas de *data warehouse* atuais.

Os algoritmos SVM, Strobe e Nested SWEEP proporcionam *consistência forte* em cenários de *transação de atualização única*. A principal vantagem do algoritmo proposto sobre estes algoritmos é a possibilidade de realizar a manutenção da visão materializada de forma programada. O *warehouse* é atualizado periodicamente, conforme intervalo de tempo previamente definido. Também, o SVM não requer um período de estabilidade do sistema em que o recebimento de mensagens tenha cessado para atualizar a visão. Isto torna o algoritmo adequado para diferentes sistemas de *data warehouse* com características e necessidades diversas.

O monitor do *warehouse* definido no algoritmo proposto foi implementado com o objetivo de mostrar seu funcionamento. O Capítulo 4 apresenta os principais aspectos de implementação e um estudo de caso do SVM.

Capítulo 4

Implementação e Utilização do SVM

O algoritmo SVM foi avaliado através da implementação do monitor do *warehouse* descrito no capítulo anterior. Este capítulo apresenta os principais aspectos de implementação e as restrições da solução implementada. Dados de teste, gerados considerando a definição de um ambiente de base de dados distribuído, são utilizados em um estudo de caso sobre a utilização do algoritmo SVM.

4.1. Aspectos de Implementação

Foram implementadas as principais funções relacionadas com o monitor do *warehouse*, utilizando-se a linguagem C, com o objetivo de avaliar o funcionamento do algoritmo SVM. Desenvolveu-se uma solução que está estruturada em módulos, conforme apresentado na Figura 4.1. Os principais aspectos de implementação desses módulos são apresentados a seguir.

4.1.1. Módulo 1 - *Prepara Dados*

O Módulo 1 (*Prepara Dados*) é responsável por preparar estruturas de dados com as informações referentes às fontes de dados relacionadas e à definição da visão do *warehouse*. Estas estruturas de dados são utilizadas pelos outros módulos da solução implementada.

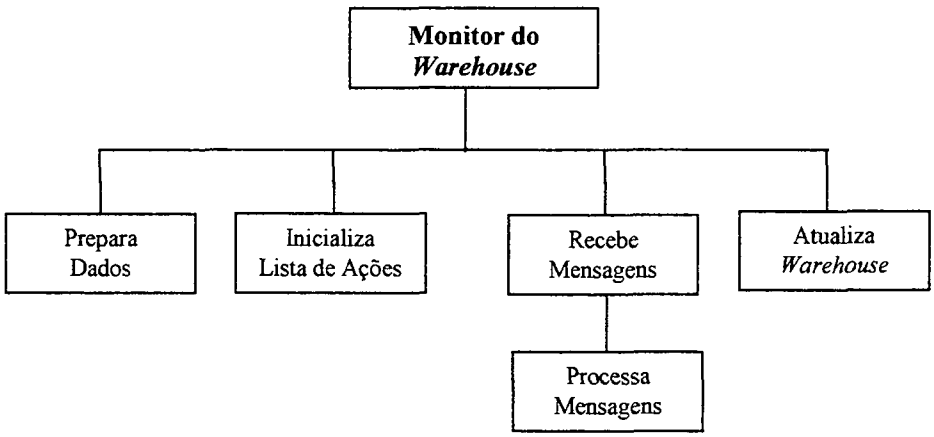

Figura 4.1: Módulos do SVM

Para a implementação da solução, assume-se que a visão do *warehouse* é definida utilizando-se a linguagem SQL (*Structured Query Language*) e que a estrutura da expressão SQL consiste em três cláusulas: *select*, *from* e *where*.

No Capítulo 2, a visão do *warehouse* foi definida através de uma expressão *Project-Select-Join* (PSJ) da álgebra relacional. A álgebra relacional é uma linguagem formal que fornece uma notação concisa para representar consultas. No entanto, sistemas comerciais de bancos de dados requerem uma linguagem mais "amigável ao usuário". A SQL foi utilizada como a linguagem de definição da visão do *warehouse* para a implementação do algoritmo SVM, pois é a linguagem padrão de banco de dados relacional utilizada como base pelos principais SGBDs (*Sistemas Gerenciadores de Bancos de Dados*) relacionais comerciais [14]. Outras vantagens da SQL são: 1) Declarações SQL podem ser embutidas diretamente no código do programa fonte de diversas linguagens de programação como C, Fortran e Pascal; 2) Uma expressão PSJ pode ser traduzida em uma expressão SQL equivalente [2].

Os dados da definição da visão são armazenados na estrutura de dados *View*. A função *gravaV* prepara a estrutura *View* para ser utilizada pelos outros módulos da solução. "Preparar" significa criar a estrutura *View* e atribuir valores a ela. Estes valores são lidos de um arquivo de definição da visão do *warehouse* e atribuídos à estrutura *View*.

A Figura 4.2 exemplifica o conteúdo de um arquivo de definição da visão V, onde R1, R2 e R3 são as três relações base armazenadas nas fontes de dados 1, 2 e 3, respectivamente.

```
create view V
  select R1.A, R2.C, R3.D
  from R1, R2, R3
  where R1.B = R2.B, R2.C = R3.C;
```

Figura 4.2: Arquivo de definição da visão V

A expressão PSJ da álgebra relacional apresentada abaixo, é equivalente a expressão SQL apresentada na Figura 4.2.

$$V = \pi_{R1.A,\, R2.C,\, R3.D}(R1 \bowtie_{R1.B = R2.B} R2 \bowtie_{R2.C = R3.C} R3)$$

A estrutura *View* é implementada como um vetor de listas ligadas com quatro elementos. Cada elemento do vetor aponta para uma lista ligada que contém os itens de uma cláusula da definição SQL. Desta forma, as cláusulas da definição da visão do *warehouse* são armazenadas em ordem no vetor, sendo elas: *create view*, *select*, *from* e *where*.

Cada elemento existente em uma cláusula é armazenado em um item distinto da lista ligada correspondente. Para armazenar cada elemento de uma cláusula em um item da lista, o Módulo *Prepara Dados* utiliza a função *parse_str* que divide a *string* de texto, definida em cada cláusula SQL, em elementos distintos e os armazena na respectiva lista. A Figura 4.3 exemplifica a estrutura *View* criada com base nos dados do arquivo de entrada apresentado na Figura 4.2.

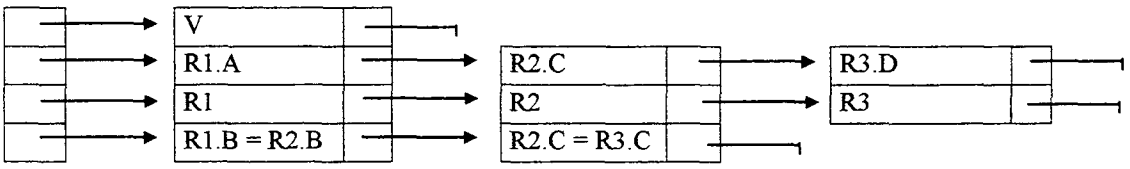

Figura 4.3: Estrutura *View* criada com base no arquivo de definição da visão V

A função *gravaR* prepara a estrutura de dados *Rel* para ser utilizada pelos outros módulos pertencentes à solução implementada. "Preparar" significa criar a estrutura *Rel* e atribuir valores a ela. Estes valores são lidos de um arquivo de dados com informações sobre as relações base relacionadas com a visão do *warehouse* e atribuídos à estrutura *Rel*.

A estrutura *Rel* é implementada como um vetor de listas ligadas com *n* elementos, onde *n* é o número de relações base envolvidas com o *warehouse*. As informações de cada relação

base são armazenadas em uma lista distinta. Os itens das listas armazenam informações relevantes sobre os campos das relações, como:

- Nome do campo;
- Tipo do campo;
- O campo participa da cláusula *select* da definição da visão;
- O campo participa da cláusula *where* da definição da visão;
- O campo compõe a chave primária na relação.

4.1.2. Módulo 2 - *Inicializa Lista de Ações*

A função *gravaAL* prepara a estrutura de dados AL para ser utilizada pelos outros módulos da solução. A estrutura AL armazena a lista de ações a serem aplicadas para atualizar o *warehouse* e é implementada como um vetor de listas ligadas. Como a lista de ações pode armazenar diversas operações, cada elemento do vetor irá conter um apontador para uma lista referente a uma operação a ser executada na visão materializada.

4.1.3. Módulo 3 - *Recebe Mensagens*

O Módulo 3 (*Recebe Mensagens*) controla o recebimento de mensagens de atualização enviadas pelas fontes de dados relacionadas. Este módulo é responsável por:

- Monitorar o recebimento de mensagens de atualização (U_i) enviadas pelas fontes de dados relacionadas;
- Armazenar as mensagens de atualização em uma fila de mensagens.

Para verificar o recebimento de mensagens de atualização utiliza-se a função *recU*, responsável por monitorar um arquivo que contém as mensagens de atualização enviadas pelas fontes de dados. A função *recU* monitora este arquivo e cada mensagem detectada é inserida na fila de mensagens MQ (*Message Queue*). Os dados do arquivo de mensagens devem ser gravados pelos monitores das fontes de dados quando da ocorrência de uma atualização. A estrutura de dados MQ é implementada como uma fila de listas ligadas, onde cada lista armazena uma mensagem de atualização. As informações são gravadas e retiradas de MQ utilizando o conceito FIFO (*First-in, First-out*), ou seja, as primeiras mensagens

recebidas são as primeiras a serem processadas. A ordem do recebimento de mensagens é controlada através de um número de identificação atribuído a cada mensagem recebida pelo *warehouse*.

O conjunto SUMQ(i) - *Source Update Message Queue*, citado no Capítulo 3, não foi implementado em uma estrutura de dados específica. SUMQ(i) é o conjunto de mensagens de atualização ocorridas na fonte de dados *i* e ainda não processadas pelo *warehouse*. As informações de SUMQ(i) podem ser obtidas diretamente da fila de mensagens, pois SUMQ(i) é um subconjunto de MQ.

4.1.4. Módulo 4 - *Processa Mensagens*

O Módulo 4 (*Processa mensagens*) é responsável por processar as mensagens de atualização enviadas pelas fontes de dados relacionadas com o *warehouse*. Este módulo executa as seguintes tarefas:

- Retira da fila de mensagens (MQ) a próxima mensagem a ser processada;
- Gera a consulta Q_i a ser enviada para as fontes de dados relacionadas, com base nas informações da mensagem de atualização U_i que está sendo processada;
- Inicializa A_i, a resposta da consulta Q_i;
- Define a ordem de envio da consulta para as fontes de dados relacionadas;
- Gera a porção Q_i^j da consulta Q_i, e envia Q_i^j para a fonte de dados *j*;
- Recebe a resposta A_i^j, enviada pela fonte *j*, e atualiza a resposta A_i com base nas informações de A_i^j;
- Depois de receber a resposta de todas as fontes de dados relacionadas, atualiza AL com base nas informações de A_i.

A função *gravaUi* retira de MQ a próxima mensagem a ser processada. Os dados desta mensagem são atribuídos a uma estrutura de dados que armazena informações relevantes sobre a atualização ocorrida, como:

- Nome da relação onde ocorreu a atualização;
- Tipo da atualização ocorrida (inserção ou exclusão);

- Campos da relação relevantes para a atualização do *warehouse*;
- Informações atualizadas.

As mensagens de atualização são processadas uma a uma pelo monitor do *warehouse*, desta forma, somente depois de processar completamente uma mensagem de atualização é que uma nova mensagem é retirada de MQ.

Quando a mensagem de atualização (U_i) refere-se a uma exclusão, os dados desta atualização são inseridos diretamente na lista de ações. A função *gravaALdel* insere em AL uma ação para excluir todas as tuplas da visão materializada que possuírem os valores dos atributos chave iguais aos valores apresentados em U_i.

Quando a mensagem de atualização (U_i) refere-se a uma inserção, o próximo passo a ser executado é gerar a consulta Q_i com base nos dados da atualização U_i. Esta tarefa é realizada pela função *gravaQ*. A função *gravaQ* gera a consulta Q_i utilizando as informações existentes na mensagem U_i e na estrutura de dados *View*. A consulta gerada é armazenada em uma estrutura de dados que utiliza a mesma definição da estrutura *View*.

A função *gravaQ* utiliza as seguintes funções para preparar a consulta Q_i: *gravaQi*, *replace_select*, *replace_where* e *replace_from*. A função *gravaQi* realiza uma cópia dos dados existentes na estrutura de dados *View* para Q_i. As funções *replace_select* e *replace_from* percorrem as listas em Q_i referentes às cláusulas *select* e *from*, respectivamente, excluindo os campos referentes à relação que gerou U_i. A função *replace_where* percorre a lista de Q_i referente à cláusula *where* e substitui os campos de Q_i pelos valores destes campos referenciados na mensagem de atualização U_i.

Depois de gerar a consulta Q_i, o Módulo *Recebe Mensagens* chama a função *gravaA*. Esta função prepara a estrutura de dados A_i que irá armazenar a resposta referente à consulta Q_i. Os valores dos campos da relação que gerou U_i, relevantes para a atualização da visão do *warehouse*, são atribuídos à estrutura A_i. Estes valores devem ser unidos com os valores das repostas enviadas pelas fontes de dados relacionadas para compor a atualização a ser executada no *warehouse*. A estrutura de dados A_i é implementada como um vetor de listas ligadas. Como a resposta A_i pode possuir diversas tuplas, cada elemento do vetor da estrutura A_i irá conter um apontador para uma lista referente a uma tupla da resposta. Cada item da lista armazena dados sobre um dos campos referenciados na cláusula *select* da consulta Q_i.

A função *ParseQuery* seleciona a próxima fonte de dados para a qual a consulta Q_i deve ser enviada. As consultas são processadas primeiramente por todas as relações localizadas na direção esquerda da relação onde ocorreu a atualização e depois por todas as relações localizadas na direção direita, considerando a expressão condicional da definição da visão. Para executar o envio das consultas nesta ordem, a função *ParseQuery* implementa dois laços *for*. O primeiro laço envia as consultas para as fontes que armazenam as relações da esquerda e o segundo laço para as fontes que armazenam as relações da direita.

A função *gravaQij* gera a porção Q_i^j, da consulta Q_i, a ser enviada para a fonte de dados *j*, com base nas informações da consulta Q_i e da estrutura de dados *Rel*. A consulta Q_i^j é armazenada em uma estrutura de dados que utiliza a mesma definição da estrutura *View*.

A função *gravaQij* utiliza as seguintes funções para gerar Q_i^j: *selectQij*, *fromQij* e *whereQij*. As funções *selectQij*, *fromQij* e *whereQij* percorrem as listas da estrutura de dados Q_i referentes às cláusulas *select*, *from* e *where*, respectivamente, selecionando e gravando em Q_i^j somente os campos relacionados com a relação para a qual a consulta será enviada. A função *selectQij* utiliza a função *ins_where* para inserir na cláusula *select* da consulta Q_i^j, os campos referenciados na cláusula *where* da definição da visão do *warehouse* e relacionados com a relação para a qual a consulta será enviada. Os valores destes campos são necessários para atualizar a cláusula *where* da consulta Q_i, depois de receber a resposta de cada fonte de dados relacionada.

Para verificar o recebimento das respostas enviadas pelas fontes de dados, a função *recAi* monitora um arquivo de respostas. Os dados deste arquivo devem ser gravados pelos monitores das fontes de dados quando do envio da resposta referente a uma consulta. A função *recAi* lê as informações existentes no arquivo de respostas, atribui os valores da resposta à estrutura de dados A_i^j e exclui estes dados do arquivo. A estrutura A_i^j armazena as informações enviadas por uma fonte em resposta a uma consulta recebida e utiliza a mesma definição da estrutura de dados A_i.

Antes de atualizar a resposta A_i, com os dados da resposta A_i^j, é preciso verificar a ocorrência de atualizações concorrentes e compensar seus efeitos. Depois do monitor do *warehouse* receber a resposta A_i^j da fonte de dados *j*, a função *compQij* verifica se existe em MQ alguma atualização de inserção ou exclusão relacionada com a fonte *j*.

Se a atualização encontrada em MQ for uma inserção, os valores dos campos chave desta atualização são comparados com os valores dos campos chave das tuplas existentes em A_i^j. Caso os valores da tupla de MQ sejam iguais aos valores de alguma tupla existente em A_i^j, esta atualização é efetivamente concorrente e seus efeitos precisam ser compensados do resultado final da resposta A_i^j. Neste caso, a resposta possui uma tupla que não existia no momento em que a atualização U_i ocorreu, portanto, esta tupla é excluída de A_i^j.

Entretanto, se a atualização encontrada em MQ for uma exclusão, a função *compQij* verifica se os valores dos campos atualizados, que participam da cláusula *where* da definição da visão, correspondem aos valores especificados para estes campos na cláusula *where* da consulta Q_i^j. Se estes valores forem iguais, a tupla da atualização existente em MQ deve ser inserida na resposta A_i^j.

Desta forma, a resposta A_i^j é atualizada com base nas informações de MQ, eliminado-se os efeitos causados pelas atualizações concorrentes, se necessário. Então, a resposta A_i é atualizada com base nos dados existentes em A_i^j pela função *recAi*. Os campos da fonte *j* que participam da cláusula *where* da consulta Q_i, também são atualizados com os valores retornados em A_i^j.

Depois de receber a resposta de todas as fontes relacionadas, os dados existentes em A_i são inseridos na lista de ações (AL) pela função *gravaALins*.

4.1.5. Módulo 5 - *Atualiza o Data Warehouse*

O Módulo 5 (*Atualiza o Data Warehouse*) é responsável por atualizar a visão materializada do *warehouse* periodicamente. As operações de atualização existentes na lista de ações (AL) são gravadas em um arquivo de dados pela função *CountTime*, considerando o intervalo de tempo definido para atualizar o *warehouse*. Então, estas operações são excluídas de AL.

A manutenção da visão materializada acontece de forma paralela ao processamento dos outros módulos da solução.

4.2. Restrições

A solução implementada engloba os módulos apresentados na Seção 4.1 deste capítulo. Estes módulos executam tarefas relacionadas com o monitor do *warehouse*. Conforme apresentado no Capítulo 3, o SVM é composto de duas partes principais: o monitor da fonte de dados e o monitor do *warehouse*.

A principal função do monitor da fonte de dados é realizar a comunicação das fontes de dados com o monitor do *warehouse*, de duas formas distintas: 1) Avisar o monitor do *warehouse* da ocorrência de uma atualização; 2) Enviar a resposta referente a uma consulta recebida do monitor do *warehouse*. Para implementar a solução adotada neste trabalho, todas as funções executadas pelo monitor da fonte de dados foram simuladas. A troca de informações entre as fontes de dados e o monitor do *warehouse* foi simulada através da utilização de arquivos de dados contendo as informações recebidas e enviadas pelas fontes. Os conjuntos de dados de teste, utilizados em um estudo de caso do algoritmo SVM, foram definidos e previamente gravados em arquivos de dados, simulando desta forma, o envio de informações por parte das fontes de dados.

Foram implementadas as principais funções relacionadas com o monitor do *warehouse*. As funções do monitor do *warehouse* responsáveis por realizar a comunicação do *warehouse* com as fontes de dados, bem como, a interação da solução com a base de dados do *warehouse* não foram implementadas. O envio das consultas para as fontes de dados foi simulado através da gravação dos dados das consultas em arquivos de dados. Também, a atualização da visão materializada do *warehouse* foi simulada da mesma forma, ou seja, as operações contidas na lista de ações são gravadas em um arquivo de dados.

O SVM não foi testado em um ambiente *warehousing* real. Um ambiente fictício de base de dados distribuído de um sistema bancário foi definido com o objetivo de realizar um estudo de caso para validar o algoritmo proposto. Através da definição deste ambiente foram criados os conjuntos de dados de teste utilizados neste estudo de caso, entretanto, este ambiente não foi implementado. Os dados de teste utilizados foram previamente gravados em arquivos de dados com o objetivo de simular a comunicação entre o *warehouse* e as fontes de dados.

A tarefa principal de um algoritmo de manutenção de visões materializadas para ambientes *warehousing* é realizar o processamento das mensagens de atualização recebidas e incorporar as mudanças ocorridas nas fontes de dados de forma consistente no *warehouse*. Através da solução implementada foi possível avaliar o funcionamento do algoritmo proposto, pois todas as funções relacionadas com o processamento de uma mudança ocorrida nas fontes de dados, desde o recebimento da mensagem de atualização pelo *warehouse* até a atualização da visão materializada, foram implementadas. Somente foram simuladas as funções responsáveis pela comunicação entre o *warehouse* e as fontes, e a interação com a base de dados do *warehouse*, visto que, o sistema bancário utilizado para avaliar a solução adotada não foi criado em um ambiente real.

As fontes de dados podem armazenar qualquer número de relações base, mas este trabalho assume que cada fonte possui uma única relação base que participa da definição da visão materializada do *warehouse*. Múltiplas relações base em uma única fonte podem ser tratadas separadamente. Assim, as consultas são enviadas para cada fonte em diferentes tempos, de forma separada, com o objetivo de obter a cada consulta os dados de uma única relação base. A fonte é consultada conforme o número de relações que participam da visão e o número de vezes que cada relação aparece na definição da visão materializada do *warehouse*. Outra forma de tratar diversas relações base em uma única fonte, é agrupar as consultas das relações base localizadas lado a lado na expressão condicional da definição da visão e que pertencem à mesma fonte de dados.

4.3. Um Estudo de Caso

A validação da solução implementada foi realizada através de um estudo de caso, desenvolvido com base na definição de um ambiente fictício de base de dados distribuído, criado com o objetivo de simular a manutenção de uma visão materializada do *warehouse* de uma instituição bancária. Este ambiente não foi implementado, sendo utilizado apenas como referência para definir os dados de teste usados neste estudo de caso.

Para a proposta deste capítulo, assume-se um sistema bancário com três fontes de dados distintas que residem em diferentes localizações:

- Fonte 1 (*cliente*);
- Fonte 2 (*conta corrente*);
- Fonte 3 (*conta poupança*).

A Figura 4.4 apresenta o Diagrama Entidade-Relacionamento (Diagrama ER) do sistema bancário, cuja definição foi utilizada como base para avaliar o funcionamento do SVM. O Diagrama ER possui 5 entidades: *cliente*, *conta corrente*, *movimenta conta corrente*, *conta poupança* e *movimenta poupança*. A entidade cliente pertence à fonte de dados 1 e as entidades *conta corrente* e *movimenta conta corrente* pertencem à fonte de dados 2. A fonte de dados 3 engloba as entidades *conta poupança* e *movimenta poupança*.

Conforme definido no Diagrama ER, um cliente pode não ter nenhuma conta corrente, bem como, possuir uma ou mais contas corrente. Da mesma forma, um cliente pode possuir uma ou mais contas poupança, ou ainda, não ter nenhuma conta poupança.

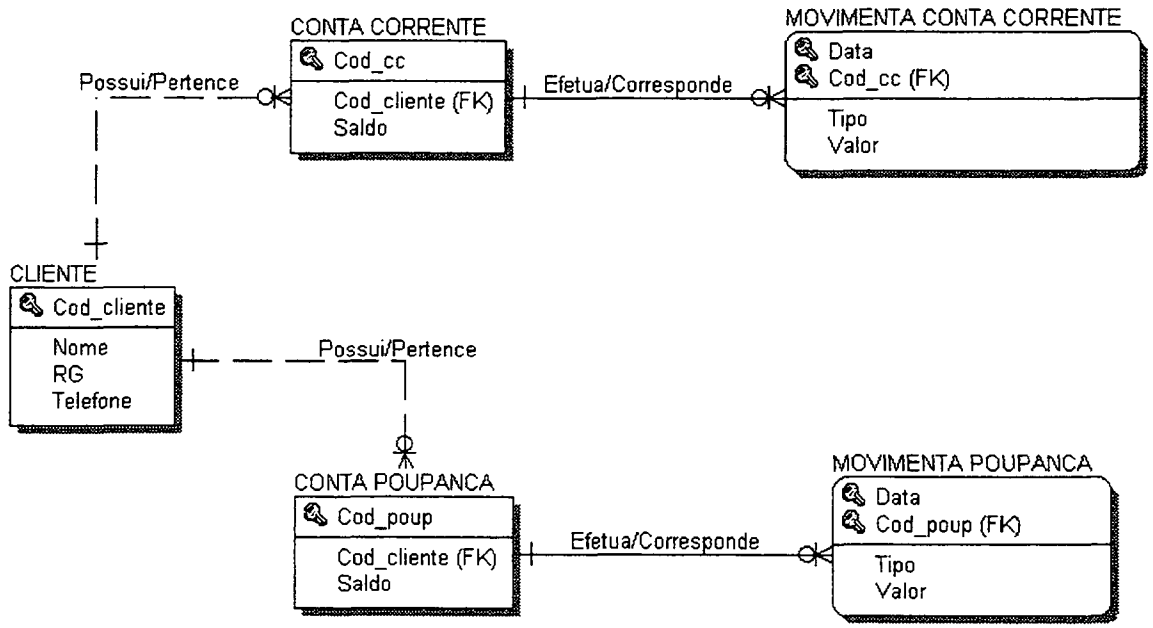

Figura 4.4: Diagrama ER do sistema bancário

O esquema da base de dados relacional do sistema bancário é apresentado na Figura 4.5. Os nomes dos atributos chave de cada relação base estão em negrito.

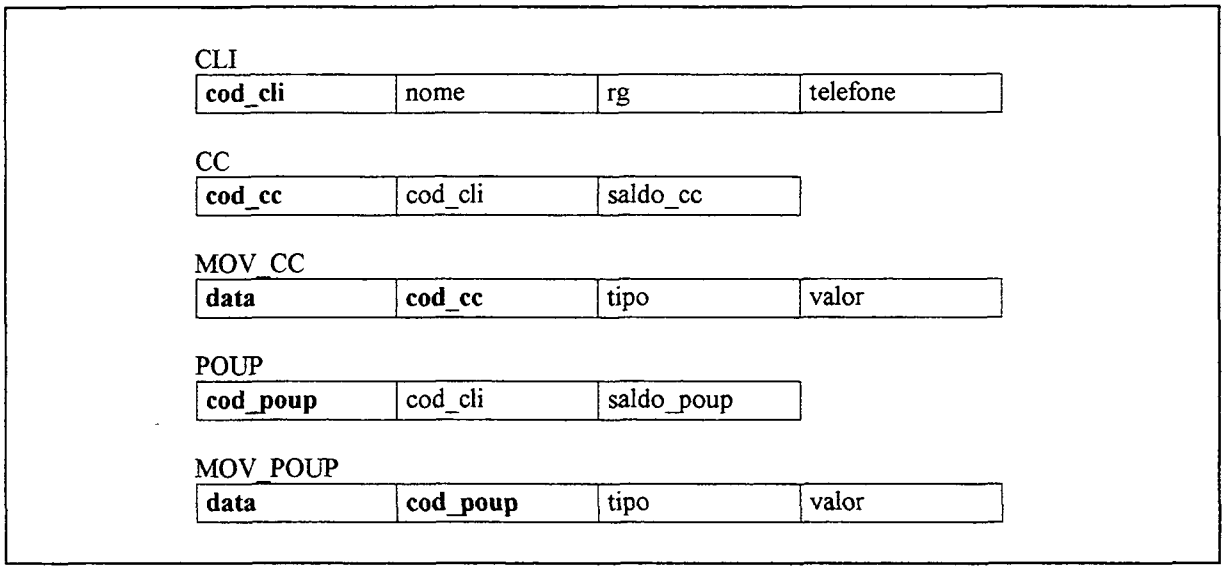

Figura 4.5: Esquema da base de dados do sistema bancário

A Tabela 4.1 mostra um dicionário de dados resumido das relações base do sistema bancário. A fonte de dados 1 (*cliente*) possui uma relação base, chamada CLI, que mantém dados referentes aos clientes. A fonte de dados 2 (*conta corrente*) possui duas relações base que armazenam dados referentes às contas correntes e às movimentações financeiras realizadas nestas contas, chamadas de CC (*conta corrente*) e MOV_CC (*movimentação da conta corrente*), respectivamente. A fonte de dados 3 (*conta poupança*) possui duas relações base que armazenam dados referentes às contas poupança e às movimentações financeiras realizadas nestas contas, chamadas de POUP (*conta poupança*) e MOV_POUP (*movimentação da conta poupança*), respectivamente.

A Figura 4.6 apresenta o esquema da visão do *warehouse* do sistema bancário, chamada BANCO. A visão do *warehouse* foi definida com base nas seguintes relações base existentes nas fontes de dados 1, 2 e 3: CLI, CC e POUP. Os nomes dos atributos chave da visão materializada estão em negrito.

BANCO

cod_cli	nome	telefone	**cod_cc**	saldo_cc	**cod_poup**	saldo_poup

Figura 4.6: Esquema da visão do *data warehouse* do sistema bancário

Tabela 4.1: Dicionário de dados do sistema bancário

Relação Base	Nome do Campo	Campo Chave	Descrição
CLI	cod_cli	sim	Código do cliente
	nome	não	Nome do cliente
	rg	não	Número do Registro de Identidade (R.G.) do cliente
	telefone	não	Telefone do Cliente
CC	cod_cc	sim	Código da conta corrente
	cod_cli	não	Código do cliente responsável pela conta corrente
	saldo_cc	não	Saldo da conta corrente
MOV_CC	data	sim	Data da movimentação da conta corrente
	cod_cc	sim	Conta corrente que sofreu movimentação
	tipo	não	Tipo da movimentação realizada na conta corrente
	valor	não	Valor da movimentação na conta corrente
POUP	cod_poup	sim	Código da conta poupança
	cod_cli	não	Código do cliente responsável pela conta poupança
	saldo_poup	não	Saldo da conta poupança
MOV_POUP	data	sim	Data da movimentação da conta poupança
	cod_poup	sim	Conta poupança que sofreu movimentação
	tipo	não	Tipo da movimentação realizada na conta poupança
	valor	não	Valor da movimentação na conta poupança

A Figura 4.7 apresenta a definição da visão materializada do *warehouse*, denominada BANCO, utilizando a linguagem SQL.

```
create view banco
 select cli.cod_cli, cli.nome, cli.telefone, cc.cod_cc, cc.saldo_cc, poup.cod_poup, poup.saldo_poup
 from cli, cc, poup
 where cli.cod_cli = cc.cod_cli, cc.cod_cli  = poup.cod_cli;
```

Figura 4.7: Definição da visão materializada do *warehouse* BANCO

4.3.1. Dados de Teste

Foram gerados vários conjuntos de dados de teste com o objetivo de criar diversas situações propícias para avaliar o funcionamento do algoritmo SVM. Um destes conjuntos será definido a seguir, a fim de exemplificar o estudo de caso. A notação utilizada é a mesma apresentada na Tabela 3.1 do Capítulo 3.

A Figura 4.8 apresenta o estado inicial dos dados das relações base CLI, CC e POUP, relevantes para a manutenção da visão materializada do *warehouse* BANCO. A Figura 4.9 apresenta o estado inicial dos dados da visão materializada do *warehouse* do sistema bancário.

CLI	cod_cli	nome	rg	telefone
	100	João Azevedo	9999999-5	0412536611
	110	Luiza do Amaral	8888888-6	0412650311
	120	Carla Ferraz	7777777-4	0412663554

CC	cod_cc	cod_cli	saldo_cc
	10100	100	500,50
	10110	110	250,50
	10120	120	650,00
	10121	120	250,00

POUP	cod_poup	cod_cli	saldo_poup
	20100	100	1500,50
	20110	110	2250,50

Figura 4.8: Estado inicial das relações CLI, CC e POUP

BANCO	cod_cli	nome	telefone	cod_cc	saldo_cc	cod_poup	saldo_poup
	100	João Azevedo	0412536611	10100	500,50	20100	1500,50
	110	Luiza do Amaral	0412650311	10110	250,50	20110	2250,50

Figura 4.9: Estado inicial da visão materializada do *warehouse* BANCO

São consideradas duas atualizações que ocorrem nas fontes de dados 3 e 2:

1. U_1 = insert(POUP, [20120; 120; 300,00]);
2. U_2 = delete(CC, [10121;120;250,00]).

A atualização U_1 é uma inserção da tupla [20120; 120; 300,00] na relação POUP e a atualização U_2 é uma exclusão da tupla [10121;120;250,00] na relação CC.

Inicialmente, a função *gravaV* (Módulo *Prepara Dados*) prepara a estrutura de dados *View*, a partir de um arquivo de dados que contém a definição da visão do *warehouse* BANCO. A função *gravaR* (Módulo *Prepara Dados*) prepara a estrutura de dados *Rel*, com base em informações das relações CLI, CC e POUP, armazenadas em um arquivo de dados sobre as relações base relacionadas com o *warehouse*.

A função *gravaAL* (Módulo *Inicializa Lista de Ações*) cria e inicializa a estrutura de dados AL, que irá armazenar a lista de ações a serem executadas para atualizar a visão materializada do *warehouse*.

A função *recU* (Módulo *Recebe Mensagens*) verifica o recebimento da mensagem de atualização U_1 enviada pela fonte de dados 3 e armazena informações referentes a U_1 na fila de mensagens de atualização (MQ).

A função *gravaUi* (Módulo *Processa Mensagens*) retira de MQ a próxima mensagem de atualização a ser processada, mensagem U_1. A Figura 4.10 apresenta a mensagem de atualização em processamento (U_1) e a consulta Q_1 gerada pela função *gravaQ* (Módulo *Processa Mensagens*).

```
-------------------------------------------------
MENSAGEM DE ATUALIZACAO EM PROCESSAMENTO: U1
-------------------------------------------------
Ocorrida na Relacao: poup
Tipo de Atualizacao: 1 - insercao
Informacoes atualizadas
    Campo: poup.cod_poup, Informacao: 20120
    Campo: poup.cod_cli, Informacao: 120
    Campo: poup.saldo_poup, Informacao: 300,00

GERA A CONSULTA Q1 REFERENTE A ATUALIZACAO U1
    select  cli.cod_cli , cli.nome , cli.telefone , cc.cod_cc ,
cc.saldo_cc
    from cli , cc
    where cli.cod_cli=cc.cod_cli , cc.cod_cli=120 ;
```

Figura 4.10: Mensagem de atualização U_1 e consulta Q_1

A resposta A_1 é inicializada pela função *gravaA* (Módulo *Processa Mensagens*) com base nos dados de U_1. A função *recU* (Módulo *Recebe Mensagens*) identifica o recebimento de uma nova mensagem de atualização, a mensagem U_2, ocorrida na fonte de dados 2. Os dados de U_2 são armazenados em MQ. A Figura 4.11 apresenta a saída gerada pela solução no momento em que o monitor do *warehouse* recebe a mensagem de atualização U_2.

```
MONITOR DO DATA WAREHOUSE RECEBE MENSAGEM DE ATUALIZACAO
FILA DE MENSAGENS DE ATUALIZACAO - (MQ)
Atualizacao U2
Ocorrida na Relacao: cc
Tipo de Atualizacao: 2 - exclusao
Informacoes atualizadas
    Campo: cc.cod_cc, Informacao: 10121
    Campo: cc.cod_cli, Informacao: 120
    Campo: cc.saldo_cc, Informacao: 250,00
```

Figura 4.11: Monitor do *warehouse* recebe mensagem de atualização U_2

A função *ParseQuery* (Módulo *Processa Mensagens*) seleciona a próxima fonte de dados que deverá receber a consulta Q_1. Como as consultas são enviadas primeiramente para todas as fontes localizadas na direção esquerda da fonte onde ocorreu U_1, considerando a expressão condicional da definição da visão, a próxima fonte a receber a consulta é a fonte de dados 2. A Figura 4.12 apresenta a ordem de processamento iterativo da consulta Q_1, considerando a definição da visão e a atualização U_1 ocorrida na fonte de dados 3.

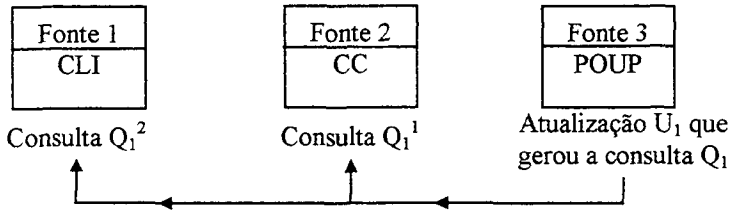

Figura 4.12: Processamento iterativo da consulta Q_1

A função *gravaQij* (Módulo *Processa Mensagens*) gera a porção Q_1^1 da consulta Q_1, a ser enviada para fonte de dados 2. A consulta Q_1^1 gerada pela solução é apresentada na Figura 4.13.

```
select  cc.cod_cc , cc.cod_cli , cc.saldo_cc
from cc
where cc.cod_cli=120;
```

Figura 4.13: Consulta Q_1^1

A função *recAi* (Módulo *Processa Mensagens*) verifica o recebimento da resposta A_1^1 enviada pela fonte de dados 2. A Figura 4.14 apresenta a resposta A_1^1.

```
FONTE 2 ENVIA RESPOSTA A11
  A11[1]:
  cc.cod_cc : 10120
  cc.cod_cli : 120
  cc.saldo_cc : 650,00
```

Figura 4.14: Reposta A_1^1

Depois de receber A_1^1, a função *compQij* (Módulo *Processa Mensagens*) verifica se não existem atualizações concorrentes em MQ relacionadas com a fonte de dados 2. A fila MQ contém a atualização U_2, uma exclusão relacionada com a fonte de dados 2. Como a atualização U_2 aconteceu enquanto a consulta Q_1^1 estava sendo processada, a resposta A_1^1 pode não corresponder ao estado dos dados da fonte 2 no momento em que U_1 ocorreu. Para verificar se U_2 interferiu no resultado da resposta A_1^1, o valor do campo cc.cod_cli existente em U_2 é comparado com o valor especificado na expressão condicional da cláusula *where* da consulta Q_1^1 relacionada com este campo. Como estes valores são iguais, é preciso compensar os efeitos da atualização U_2 em A_1^1. A compensação é realizada através da inclusão da tupla de U_2 na resposta A_1^1. A Figura 4.15 apresenta a resposta A_1^1, depois de compensar os efeitos causados pela atualização concorrente U_2.

```
FORAM ENCONTRADAS ATUALIZACOES CONCORRENTES NA RELACAO: cc
COMPENSA EFEITO DAS ATUALIZACOES CONCORRENTES EM A11
A11[1]:
  cc.cod_cc : 10120
  cc.cod_cli : 120
  cc.saldo_cc : 650,00
A11[2]:
  cc.cod_cc : 10121
  cc.cod_cli : 120
  cc.saldo_cc : 250,00
```

Figura 4.15: Reposta A_1^1 depois de compensar os efeitos de U_2

A resposta A_1 é atualizada pela função *recAi*, com base nos dados de A_1^1. A função *ParseQuery* envia então a porção Q_1^2 da consulta Q_1 para a fonte de dados 1. Depois de receber a resposta A_1^2 da fonte 1, a função *compQij* verifica a existência de atualizações em MQ relacionadas com a fonte 1. Como a fila de mensagens está vazia, ou seja, não possui nenhuma mensagem de atualização, a função *recAi* atualiza a resposta A_1 com base nos dados

de A_1^2. Neste momento, A_1 é a resposta final de Q_1, pois a consulta Q_1 já foi enviada para todas as fontes de dados relacionadas.

A função *gravaALins* (Módulo *Processa Mensagens*) atualiza AL com base nos dados de A_1, ou seja, as ações de A_1 são incorporadas na lista de ações. A Figura 4.16 apresenta o conteúdo de AL, depois de incorporar os dados de A_1.

Depois do processamento da mensagem de atualização U_1, a função *gravaUi* (Módulo *Processa Mensagens*) retira U_2, a próxima mensagem a ser processada, da fila de mensagens. Como U_2 refere-se a uma exclusão ocorrida na relação CC, uma ação de exclusão para todas as tuplas da visão materializada do *warehouse* que possuem o valor do atributo cc.cod_cc, atributo chave de CC, igual ao valor deste atributo referenciado em U_2, é inserida diretamente em AL pela função *gravaALdel*. O conteúdo da lista de ações depois de processar a mensagem de atualização U_2 é apresentado na Figura 4.17.

O Módulo *Atualiza o Data Warehouse* aplica as ações existentes em AL na visão materializada do *warehouse* periodicamente, conforme o intervalo de tempo previamente definido.

```
-----------------------------------------------------
AL - LISTA DE ACOES A SER APLICADA NO DATA WAREHOUSE
-----------------------------------------------------
AL[1]:
  INSERT INTO BANCO
  VALUES  ('120','CarlaFerraz','0412663554','10120','650,00','20120','300,00');
AL[2]:
  INSERT INTO BANCO
  VALUES ('120','Carla Ferraz','0412663554','10121','250,00','20120','300,00');
```

Figura 4.16: AL depois de processar U_1

```
-----------------------------------------------------
AL - LISTA DE ACOES A SER APLICADA NO DATA WAREHOUSE
-----------------------------------------------------
AL[1]:
  INSERT INTO BANCO
  VALUES ('120','Carla  Ferraz','0412663554','10120','650,00','20120','300,00');
AL[2]:
  INSERT INTO BANCO
  VALUES ('120','Carla Ferraz','0412663554','10121','250,00','20120','300,00');
AL[3]:
  DELETE FROM BANCO
  WHERE cc.cod_cc = '10121';
```

Figura 4.17: AL depois de processar U_2

Considerando que o intervalo de tempo definido para a atualização do *warehouse* tenha ocorrido depois de processar U_1, mas antes de terminar o processamento de U_2. A Figura 4.18 apresenta o estado dos dados da visão materializada do *warehouse* BANCO, aplicando-se as ações de AL depois de processar U_1, considerando o conteúdo de AL apresentado na Figura 4.16. O estado da visão materializada é consistente com o estado dos dados das fontes no momento em que ocorreu U_1.

BANCO	cod_cli	nome	telefone	cod_cc	saldo_cc	cod_poup	saldo_poup
	100	João Azevedo	0412536611	10100	500,50	20100	1500,50
	110	Luiza do Amaral	0412650311	10110	250,50	20110	2250,50
	120	Carla Ferraz	0412663554	10120	650,00	20120	300,00
	120	Carla Ferraz	0412663554	10121	250,00	20120	300,00

Figura 4.18: Estado da visão materializada do *warehouse* BANCO depois de aplicar AL

Considerando esta mesma situação, onde AL é aplicada na visão materializada do *warehouse* depois de completar o processamento de U_1, para exemplificar uma anomalia nos dados da visão do *warehouse*. Se a compensação dos efeitos das atualizações concorrentes não fosse realizada pelo algoritmo SVM (função *compQij* - Módulo *Processa Mensagens*), o *warehouse* seria atualizado considerando um estado dos dados da fonte 2 que não corresponde ao real estado desta fonte quando ocorreu a atualização U_1. A Figura 4.19 exemplifica o estado da visão materializada do *warehouse* BANCO, caso o algoritmo SVM não realizasse a compensação dos efeitos das atualizações concorrentes. Este estado da visão é inconsistente com o estado das fontes relacionadas no momento em que ocorreu U_1. É importante ressaltar que, depois de processar U_2 e na próxima atualização da visão materializada do *warehouse*, o estado final da visão seria consistente com os dados da fonte, entretanto, teria passado por estados intermediários inconsistentes.

BANCO	cod_cli	nome	telefone	cod_cc	saldo_cc	cod_poup	saldo_poup
	100	João Azevedo	0412536611	10100	500,50	20100	1500,50
	110	Luiza do Amaral	0412650311	10110	250,50	20110	2250,50
	120	Carla Ferraz	0412663554	10120	650,00	20120	300,00

Figura 4.19: Estado inconsistente da visão materializada do *warehouse* BANCO

Sempre que o *warehouse* for atualizado, o estado dos dados da visão materializada deve refletir um conjunto de estados consistentes das fontes de dados, conforme proposto na

definição do algoritmo (nível de *consistência forte*). Ainda, a visão materializada do *warehouse* não pode assumir estados intermediários inconsistentes.

A saída completa gerada pela solução implementada, considerando este conjunto de dados de teste, é apresentada no Apêndice A. A saída mostra os passos de processamento das mensagens de atualização e apresenta o conteúdo da lista de ações depois de processar cada atualização. Este apêndice também apresenta a descrição de outros dois conjuntos de dados de teste criados, bem como, a saída gerada para estes dados.

4.4. Considerações Finais

Neste capítulo foram apresentados os módulos do algoritmo SVM implementados, utilizando a linguagem C, com o objetivo de mostrar o funcionamento deste algoritmo. Um estudo de caso foi definido com base em um ambiente de base de dados distribuído fictício para testar a solução implementada.

Foram implementadas as funções do monitor do *warehouse*. As funções que realizam a comunicação entre as fontes e o *warehouse* foram simuladas, utilizando-se arquivos de dados com informações previamente gravadas. Também, o envio de consultas para as fontes e a interação da solução adotada com a base de dados do *warehouse* foram simulados através da utilização de arquivos de dados nos quais as informações a serem enviadas foram armazenadas.

Através da solução implementada foi possível verificar que o algoritmo SVM processa as mensagens de atualização que chegam das fontes de dados relacionadas seqüencialmente, atualizando a visão materializada periodicamente e produzindo uma seqüência de estados *warehouse* consistentes. Os efeitos das mudanças que acontecem nas fontes enquanto o monitor do *warehouse* está processando uma consulta são eliminados da resposta desta consulta.

O estudo de caso aqui apresentado e conseqüente validação do funcionamento do SVM constitui-se uma motivação para a utilização do algoritmo em um ambiente *warehousing* real. O Capítulo 5 apresenta as principais conclusões deste trabalho e as sugestões de trabalhos futuros.

Capítulo 5

Conclusões

A manutenção incremental das visões materializadas em *data warehouses* deve refletir as atualizações ocorridas sobre os dados das fontes. As vantagens da manutenção incremental sobre as técnicas de reprocessamento e replicação dos dados são principalmente o menor tempo requerido para atualizar a visão se comparada à técnica de reprocessamento e o menor espaço de armazenamento requerido com relação à técnica de replicação.

Este trabalho realizou um estudo dos principais algoritmos de manutenção incremental de visões materializadas para ambientes *warehousing*. Como resultado desse estudo um novo algoritmo foi proposto, o algoritmo SVM que reúne os principais fundamentos dos algoritmos encontrados na literatura. O SVM foi projetado para ambientes *warehousing* com múltiplas fontes de dados e proporciona *consistência forte* em cenários de *transação de atualização única*, ou seja, cada estado *warehouse* reflete um conjunto de estados das fontes. *Consistência forte* é um requisito de consistência adequado para a maioria dos cenários de *data warehouse* atuais. Através do SVM é possível realizar uma manutenção programada da visão materializada, ou seja, o *data warehouse* é atualizado periodicamente. A periodicidade da manutenção do *warehouse* é definida pelo usuário final.

O algoritmo SVM atende a um cenário mais complexo se comparado ao algoritmo ECA, projetado para cenários de *data warehouse* com uma única fonte de dados. Com relação aos outros algoritmos que proporcionam *consistência forte* para múltiplas fontes, algoritmos Strobe e Nested SWEEP, o principal benefício do SVM é a possibilidade de determinar o momento em que a visão materializada deve ser atualizada, sem depender do volume de

mensagens de atualização recebidas pelo *warehouse*. Os algoritmos Strobe requerem um estado estável do sistema de *data warehouse* e o Nested SWEEP requer um período em que não ocorram atualizações concorrentes com relação à consulta que está sendo processada.

O algoritmo C-Strobe não requer um estado estável do sistema de *data warehouse* para atualizar a visão materializada, entretanto, o número de mensagens de consultas necessárias para processar cada atualização é muito alto, se comparado ao número de mensagens enviadas pelo algoritmo SVM.

Uma das vantagens do algoritmo SVM sobre os algoritmos SWEEP e Nested SWEEP, é que as exclusões são tratadas diretamente no *warehouse*, sem necessitar gerar e enviar consultas para as fontes de dados relacionadas. Assim, o número de consultas enviadas às fontes para processar múltiplas atualizações é menor.

Os algoritmos SWEEP e C-Strobe proporcionam *consistência completa*. *Consistência completa* é um requisito muito forte para a maioria dos cenários de *data warehouse* atuais, pois implica que cada atualização das fontes seja refletida em um estado distinto do *warehouse*.

O algoritmo SVM é composto pelo monitor do *warehouse* e pelo monitor da fonte. O monitor do *warehouse* foi implementado, utilizando a linguagem de programação C, com o objetivo de avaliar o funcionamento do SVM. Um estudo de caso foi realizado, considerando um ambiente de base de dados distribuído fictício, para validar a solução implementada.

O monitor da fonte não foi implementado e suas funções principais, comunicação entre as fontes de dados e o *warehouse*, foram simuladas. Decidiu-se por não implementar o monitor das fontes, pois as especificações deste monitor variam conforme as características de cada fonte de dados. Por exemplo, o modelo utilizado pelas fontes de dados pode ser o relacional ou hierárquico. No caso de fontes relacionais, diferentes Sistemas Gerenciadores de Banco de Dados (SGBDs) podem ser utilizados.

Resumindo, têm-se as seguintes características do algoritmo SVM:

- Projetado para cenários de *data warehouse* com múltiplas fontes de dados;
- Oferece nível de *consistência forte*;

- Compensação dos efeitos causados pelas atualizações concorrentes realizada localmente no *data warehouse*, o que elimina a necessidade de enviar *consultas compensadoras* para as fontes de dados relacionadas;
- As atualizações de exclusão ocorridas nas fontes de dados são processadas diretamente no *data warehouse*, sem precisar enviar consultas para as fontes de dados relacionadas. Isto reduz o número de consultas processadas para múltiplas atualizações das fontes;
- Manutenção programada da visão materializada do *data warehouse*;
- O algoritmo proposto pode ser utilizado em diferentes ambientes *data warehousing*.

Além do algoritmo SVM proposto também destacam-se as seguintes contribuições:

- Revisão bibliográfica do estado da arte dos principais algoritmos de manutenção incremental de visões materializadas em ambientes *warehousing*;
- Este trabalho pode ser utilizado como fonte de consulta e referência para estudo e definição da forma de gerenciamento e manutenção da consistência dos dados de um *data warehouse*.

5.1. Trabalhos Futuros

Uma sugestão para trabalho futuro é a implementação do algoritmo SVM utilizando um caso real, com o objetivo de coletar métricas, tais como:

- Desempenho com relação ao número médio de consultas necessárias para processar múltiplas atualizações;
- Desempenho com relação ao volume de dados transferido.

Um ambiente real que pode ser utilizado para avaliar o SVM é o *data warehouse* do projeto SAGU. Desta forma, um estudo de viabilidade de utilização do algoritmo SVM como abordagem para garantir a consistência dos dados da visão materializada do *data warehouse* da UFPR necessita ser realizado. Este estudo deve coletar informações como:

- Freqüência de atualizações das fontes de dados relacionadas;
- Nível de consistência requerido pelo sistema de *data warehouse*;
- Periodicidade desejada para realizar a manutenção da visão materializada.

- Modelo de dados utilizado pelas fontes de dados relacionadas;
- Sistema Gerenciador de Banco de Dados (SGBD) utilizado pelas fontes de dados relacionadas;

Para a implementação do monitor da fonte de dados e do monitor do *warehouse* que utilizam o modelo relacional, é possível relacionar algumas sugestões:

- Conforme já foi visto, declarações SQL podem ser embutidas diretamente no código do programa fonte de diversas linguagens de programação como C, Fortran e Pascal.
- As fontes de dados relacionais podem se comunicar com o *data warehouse* através de *triggers*, enviar as mensagens avisando da ocorrência de atualizações ou remeter as respostas das consultas processadas. Também, o *warehouse* pode enviar consultas para as fontes de dados através de *triggers*.
- A linguagem de programação Microsoft Visual C++, da Microsoft Corporation, oferece o *Microsoft Foundation Class Library* (MFC), uma biblioteca de classes utilizadas para construir aplicações específicas, dentre elas aplicações de banco de dados. A classes MFC oferecem ferramentas para criar aplicações que acessam às bases de dados, executando consultas ou realizando atualizações. A conexão com as bases de dados é realizada através de *drivers* ODBC (*Open Database Connectivity*). As classes MFC possibilitam acessar uma grande variedade de SGBDs para os quais os *drivers* ODBC estão disponíveis, como por exemplo: Microsoft SQL Server, SYBASE® SQL Server e ORACLE® Server.
- Para os monitores das fontes de dados e do *warehouse* que utilizam o Oracle® Server como SGBD é possível utilizar o précompilador PRO*C/C++. Este précompilador é uma ferramenta de programação que possibilita a inserção de declarações SQL em um programa fonte escrito em linguagem de programação C ou C++. Os comandos SQL são utilizados para acessar bases de dados Oracle®. O PRO*C/C++ converte as declarações SQL embutidas no programa fonte em declarações do código C padrão.

Outra sugestão para trabalho futuro é estender o algoritmo SVM para tratar *transações local-fonte*, com base nos fundamentos apresentados pelo algoritmo T-Strobe [27].

Referências Bibliográficas

[1] D. Agrawal, A.E. Abbadi, A.K. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 417-427, May 1997.

[2] J.A. Blakeley, P.-A. Larson, and F.W. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61-71, June 1986.

[3] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Base (VLDB)*, pages 577-589, September 1991.

[4] L.S. Colby, T. Griffin, L. Libkin, I.S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 469-480, June 1996.

[5] B. Devlin. *Data Warehouse from Architecture to Implementation.* Addison-Wesley, 1997.

[6] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* Addison-Wesley Publishing Company, 1994.

[7] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 328-339, May 1995.

[8] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 157-166, May 1993.

[9] A. Gupta and I. Mumick. Maintenance of Materialized Views: Problems, techniques, and applications. *IEEE Bulletin of the Technical Committee on Data Engineering*, 18(2):3-18, June 1995.

[10] A. Gupta and J.A. Blakeley. Using partial information to update materialized views. *Information Systems*, 20(8):641-662, November 1995.

[11] A. Gupta and I. Mumick, editors. *Proceedings of the workshop on materialized views: techniques and applications*, June 1996.

[12] A. Gupta and I. Mumick, editors. *Materialized Views*. MIT Press, 1998.

[13] J.V. Harrison and S.W. Dietrich. Maintenance of materialized views in a deductive database: An update propagation approach. In *Proceedings of the 1992 JICLSP Workshop on Deductive Databases*, pages 56-65, 1992.

[14] H. F. Korth and A. Silberschatz. *Sistemas de Bancos de Dados*. Segunda Edição. MAKRON Books, 1993.

[15] B. Lindsay, L.M. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A snapshot differential refresh algorithm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1986.

[16] D. Lomet and J. Widom, editors. *Special Issue on Materialized Views and Data Warehousing*, IEEE Data Engineering Bulletin 18(2), June 1995.

[17] A.T. Pozo et al. SAGU - Sistema de Apoio ao Gerenciamento Universitário. Relatório técnico do Projeto, DINF - Departamento de Informática, UFPR, agosto 2000.

[18] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3): 337-341, September 1991.

[19] A. Segev and W. Fang. Currency-based updates to distributed materialized views. In *Proceedings of the 6th International Conference on Data Engineering (ICDE)*, pages 512-520, February 1990.

[20] A. Segev and W. Fang. Optimal update policies for distributed materialized views. *Management Science*, 37(7):851-70, July 1991.

[21] A. Segev and J. Park. Updating distributed materialized views. In *IEEE Transactions on Knowledge and Data Engineering*, 1(2):173-184, June 1989.

[22] O. Shmueli and A. Itai. Maintenance of views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 240-255, May 1984.

[23] M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. In *Proceedings of the 22nd International Conference on Very Large Data Base (VLDB)*, pages 75-86, September 1996.

[24] T. Yurek. Efficient View Maintenance at Data Warehouses. Master thesis, University of California at Santa Barbara, Department of Computer Science, UCSB, Santa Barbara, CA 93106, 1997.

[25] J.L. Wiener, H. Gupta, W.J. Labio, Y. Zhuge, H. Garcia-Molina, and J. Window. A system prototype for warehouse view maintenance. In *Proceedings of the Workshop on Materialized Views, Techniques and Applications*, pages 26-33, June 1996.

[26] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing enviroment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316-327, May 1995.

[27] Y. Zhuge, H. Garcia-Molina, and J.L. Wiener. Consistency algorithms for multi-source warehouse view maintenance. *Distributed and Parallel Databases*, 6(1):7-40, January 1998.

[28] Y. Zhuge. Incremental Maintenance of Consistent Data Warehouses. Phd thesis, Stanford University, Department of Computer Science, Stanford, CA 94305-2140, June 1999.

Apêndice A

Conjuntos de Dados de Teste

Para o estudo de caso do algoritmo SVM, apresentado no Capítulo 4, foram gerados diversos conjuntos de dados de teste definidos com base no ambiente de base de dados do sistema bancário. Três destes conjuntos de dados de teste são apresentados neste apêndice. Eles diferem basicamente quanto ao tipo das atualizações consideradas e representam três diferentes situações a serem tratadas pela solução implementada.

Para todos os conjuntos de dados de testes são apresentados o estado inicial dos dados da visão materializada do *warehouse* e das relações base bem como as atualizações consideradas.

A saída gerada pela solução implementada é apresentada para cada conjunto de dados de teste. Esta saída é armazenada em um arquivo de dados. A saída gerada demonstra os passos de processamento das atualizações; entretanto, o momento em que a visão materializada é atualizada não é explicitamente definido. Como a periodicidade da manutenção pode variar de um sistema para outro, o importante é verificar que o *warehouse* pode ser atualizado em qualquer instante do processamento, pois somente as atualizações já processadas são refletidas no *warehouse*. Como os casos de teste consideram poucas atualizações, apenas duas atualizações por caso de teste, não foi necessário definir o instante em que a manutenção deve ocorrer. É importante observar que, a visão materializada pode ser atualizada depois de processar a primeira mensagem de atualização (U_1) ou ainda depois de processar todas as mensagens recebidas (U_1 e U_2). Somente os efeitos das atualizações já processadas são incorporados no *warehouse* e nenhum efeito de uma atualização concorrente é refletido. Portanto, a atualização pode ocorrer a qualquer tempo.

A.1. Conjunto de Dados de Teste 1

A Figura A.1 apresenta o estado inicial dos dados das relações CLI, CC e POUP, residentes nas fontes de dados 1, 2 e 3, respectivamente. A Figura A.2 apresenta o estado inicial da visão do *warehouse* BANCO.

São consideradas duas atualizações que ocorrem de forma concorrente, nas fontes de dados 3 e 2, sendo elas:

1. U_1 = insert(poup, [20120;120;300,00]);
2. U_2 = delet(cc, [10121; 120; 250,00]).

CLI	cod_cli	nome	rg	telefone
	100	João Azevedo	9999999-5	0412536611
	110	Luiza do Amaral	8888888-6	0412650311
	120	Carla Ferraz	7777777-4	0412663554

CC	cod_cc	cod_cli	saldo_cc
	10100	100	500,50
	10110	110	250,50
	10120	120	650,00
	10121	120	250,00

POUP	cod_poup	cod_cli	saldo_poup
	20100	100	1500,00
	20110	110	2250,50

Figura A.1: Estado inicial das relações - conjunto de dados de teste 1

BANCO	cod_cli	nome	telefone	cod_cc	saldo_cc	cod_poup	saldo_poup
	100	João Azevedo	0412536611	10100	500,50	20100	1500,00
	110	Luiza do Amaral	0412650311	10110	250,50	20110	2250,50

Figura A.2: Estado inicial do *warehouse* - conjunto de dados de teste 1

A.1.1. Saída Gerada para o Conjunto de Dados de Teste 1

```
DEFINICAO DA VISAO DO DATA WAREHOUSE
   create view  banco
      select  cli.cod_cli, cli.nome, cli.telefone, cc.cod_cc, cc.saldo_cc,
              poup.cod_poup, poup.saldo_poup
      from cli, cc, poup
      where cli.cod_cli=cc.cod_cli, cc.cod_cli=poup.cod_cli ;

--------------------------------------------------
MENSAGEM DE ATUALIZACAO EM PROCESSAMENTO: U1
--------------------------------------------------
Ocorrida na Relacao: poup
Tipo de Atualizacao: 1 - insercao
Informacoes atualizadas
    Campo: poup.cod_poup, Informacao: 20120
    Campo: poup.cod_cli, Informacao: 120
    Campo: poup.saldo_poup, Informacao: 300,00

GERA A CONSULTA Q1 REFERENTE A ATUALIZACAO U1
    select  cli.cod_cli, cli.nome, cli.telefone, cc.cod_cc, cc.saldo_cc
    from cli, cc
    where cli.cod_cli=cc.cod_cli, cc.cod_cli=120 ;

INICIALIZA A RESPOSTA A1
A1[1]:
Tipo de atualizacao: 1 - insercao
    cli.cod_cli :
    cli.nome :
    cli.telefone :
    cc.cod_cc :
    cc.saldo_cc :
    poup.cod_poup : 20120
    poup.saldo_poup : 300,00

MONITOR DO DATA WAREHOUSE RECEBE MENSAGEM DE ATUALIZACAO
FILA DE MENSAGENS DE ATUALIZACAO - (MQ)
Atualizacao U2
Ocorrida na Relacao: cc
Tipo de Atualizacao: 2 - exclusao
Informacoes atualizadas
    Campo: cc.cod_cc, Informacao: 10121
    Campo: cc.cod_cli, Informacao: 120
    Campo: cc.saldo_cc, Informacao: 250,00

PROXIMA FONTE A ENVIAR A CONSULTA Q1: 2
ENVIA CONSULTA Q11 PARA A FONTE: 2
    select  cc.cod_cc , cc.cod_cli , cc.saldo_cc
    from cc
    where cc.cod_cli=120 ;
FONTE 2 ENVIA RESPOSTA A11
A11[1]:
    cc.cod_cc : 10120
    cc.cod_cli : 120
    cc.saldo_cc : 650,00
FORAM ENCONTRADAS ATUALIZACOES CONCORRENTES NA RELACAO: cc
```

```
COMPENSA EFEITO DAS ATUALIZACOES CONCORRENTES EM A11
A11[1]:
     cc.cod_cc : 10120
     cc.cod_cli : 120
     cc.saldo_cc : 650,00
 A11[2]:
     cc.cod_cc : 10121
     cc.cod_cli : 120
     cc.saldo_cc : 250,00

 ATUALIZA A1 COM BASE NOS DADOS DE A11
 A1[1]:
 Tipo de atualizacao: 1 - insercao
     cli.cod_cli :
     cli.nome :
     cli.telefone :
     cc.cod_cc : 10120
     cc.saldo_cc : 650,00
     poup.cod_poup : 20120
     poup.saldo_poup : 300,00
 A1[2]:
 Tipo de atualizacao: 1 - insercao
     cli.cod_cli :
     cli.nome :
     cli.telefone :
     cc.cod_cc : 10121
     cc.saldo_cc : 250,00
     poup.cod_poup : 20120
     poup.saldo_poup : 300,00

 PROXIMA FONTE A ENVIAR A CONSULTA Q1: 1
 ENVIA CONSULTA Q12 PARA A FONTE: 1
     select  cli.cod_cli, cli.nome, cli.telefone
     from cli
     where cli.cod_cli=120 ;
 FONTE 1 ENVIA RESPOSTA A12
 A12[1]:
     cli.cod_cli : 120
     cli.nome : Carla Ferraz
     cli.telefone : 0412663554

 ATUALIZA A1 COM BASE NOS DADOS DE A12
 A1[1]:
 Tipo de atualizacao: 1 - insercao
     cli.cod_cli : 120
     cli.nome : Carla Ferraz
     cli.telefone : 0412663554
     cc.cod_cc : 10120
     cc.saldo_cc : 650,00
     poup.cod_poup : 20120
     poup.saldo_poup : 300,00
 A1[2]:
 Tipo de atualizacao: 1 - insercao
     cli.cod_cli : 120
     cli.nome : Carla Ferraz
     cli.telefone : 0412663554
     cc.cod_cc : 10121
     cc.saldo_cc : 250,00
     poup.cod_poup : 20120
     poup.saldo_poup : 300,00
```

```
ATUALIZA AL COM OS DADOS DE  A1 (AL = AL + A1)

-----------------------------------------------------
AL - LISTA DE ACOES A SER APLICADA NO DATA WAREHOUSE
-----------------------------------------------------
AL[1]:
  INSERT INTO BANCO
  VALUES ('120','Carla Ferraz','0412663554','10120','650,00','20120',
         '300,00');
AL[2]:
  INSERT INTO BANCO
  VALUES ('120','Carla Ferraz','0412663554','10121','250,00','20120',
         '300,00');

-----------------------------------------------------
MENSAGEM DE ATUALIZACAO EM PROCESSAMENTO: U2
-----------------------------------------------------
Ocorrida na Relacao: cc
Tipo de Atualizacao: 2 - exclusao
Informacoes atualizadas
    Campo: cc.cod_cc, Informacao: 10121
    Campo: cc.cod_cli, Informacao: 120
    Campo: cc.saldo_cc, Informacao: 250,00

U2 REFERE-SE A UMA ATUALIZACAO DE EXCLUSAO
ATUALIZA AL COM DADOS DE U2 (AL = AL+ U2)

-----------------------------------------------------
AL - LISTA DE ACOES A SER APLICADA NO DATA WAREHOUSE
-----------------------------------------------------
AL[1]:
  INSERT INTO BANCO
  VALUES ('120','Carla Ferraz','0412663554','10120','650,00','20120',
         '300,00');
AL[2]:
  INSERT INTO BANCO
  VALUES ('120','Carla Ferraz','0412663554','10121','250,00','20120',
         '300,00');
AL[3]:
  DELETE FROM BANCO
  WHERE cc.cod_cc = '10121';
```

A.2. Conjunto de Dados de Teste 2

A Figura A.3 apresenta o estado inicial dos dados das relações CLI, CC e POUP, residentes nas fontes de dados 1, 2 e 3, respectivamente. A Figura A.4 apresenta o estado inicial da visão do *warehouse* BANCO.

São consideradas duas atualizações que ocorrem de forma concorrente, nas fontes de dados 3 e 2, sendo elas:

1. U_1 = insert(poup, [20120;120;300,00]);
2. U_2 = insert(cc, [10121; 120; 250,00]).

CLI	cod_cli	nome	rg	telefone
	100	João Azevedo	9999999-5	412536611
	110	Luiza do Amaral	8888888-6	412650311
	120	Carla Ferraz	7777777-4	412663554

CC	cod_cc	cod_cli	saldo_cc
	10100	100	500,50
	10110	110	250,50
	10120	120	650,00

POUP	cod_poup	cod_cli	saldo_poup
	20100	100	1500,00
	20110	110	2250,50

Figura A.3: Estado inicial das relações - conjunto de dados de teste 2

BANCO	cod_cli	nome	telefone	cod_cc	saldo_cc	cod_poup	saldo_poup
	100	João Azevedo	412536611	10100	500,50	20100	1500,00
	110	Luiza do Amaral	412650311	10110	250,50	20110	2250,50

Figura A.4: Estado inicial do *warehouse* - conjunto de dados de teste 2

A.2.1. Saída Gerada para o Conjunto de Dados de Teste 2

```
DEFINICAO DA VISAO DO DATA WAREHOUSE
   create view  banco
      select  cli.cod_cli, cli.nome, cli.telefone, cc.cod_cc, cc.saldo_cc,
              poup.cod_poup, poup.saldo_poup
      from cli, cc, poup
      where cli.cod_cli=cc.cod_cli, cc.cod_cli=poup.cod_cli ;

-----------------------------------------------------
MENSAGEM DE ATUALIZACAO EM PROCESSAMENTO: U1
-----------------------------------------------------
Ocorrida na Relacao: poup
Tipo de Atualizacao: 1 - insercao
Informacoes atualizadas
    Campo: poup.cod_poup, Informacao: 20120
    Campo: poup.cod_cli, Informacao: 120
    Campo: poup.saldo_poup, Informacao: 300,00

GERA A CONSULTA Q1 REFERENTE A ATUALIZACAO U1
    select  cli.cod_cli, cli.nome, cli.telefone, cc.cod_cc, cc.saldo_cc
    from cli, cc
    where cli.cod_cli=cc.cod_cli, cc.cod_cli=120 ;

INICIALIZA A RESPOSTA A1
A1[1]:
Tipo de atualizacao: 1 - insercao
    cli.cod_cli :
    cli.nome :
    cli.telefone :
    cc.cod_cc :
    cc.saldo_cc :
    poup.cod_poup : 20120
    poup.saldo_poup : 300,00

MONITOR DO DATA WAREHOUSE RECEBE MENSAGEM DE ATUALIZACAO
FILA DE MENSAGENS DE ATUALIZACAO - (MQ)
Atualizacao U2
Ocorrida na Relacao: cc
Tipo de Atualizacao: 1 - insercao
Informacoes atualizadas
    Campo: cc.cod_cc, Informacao: 10121
    Campo: cc.cod_cli, Informacao: 120
    Campo: cc.saldo_cc, Informacao: 250,00

PROXIMA FONTE A ENVIAR A CONSULTA Q1: 2
ENVIA CONSULTA Q11 PARA A FONTE: 2
    select  cc.cod_cc, cc.cod_cli, cc.saldo_cc
    from cc
    where cc.cod_cli=120 ;
```

```
FONTE 2 ENVIA RESPOSTA A11
A11[1]:
    cc.cod_cc : 10120
    cc.cod_cli : 120
    cc.saldo_cc : 650,00
A11[2]:
    cc.cod_cc : 10121
    cc.cod_cli : 120
    cc.saldo_cc : 250,00
FORAM ENCONTRADAS ATUALIZACOES CONCORRENTES NA RELACAO: cc
COMPENSA EFEITO DAS ATUALIZACOES CONCORRENTES EM A11
A11[1]:
    cc.cod_cc : 10120
    cc.cod_cli : 120
    cc.saldo_cc : 650,00

ATUALIZA A1 COM BASE NOS DADOS DE A11
A1[1]:
Tipo de atualizacao: 1 - insercao
    cli.cod_cli :
    cli.nome :
    cli.telefone :
    cc.cod_cc : 10120
    cc.saldo_cc : 650,00
    poup.cod_poup : 20120
    poup.saldo_poup : 300,00

PROXIMA FONTE A ENVIAR A CONSULTA Q1: 1
ENVIA CONSULTA Q12 PARA A FONTE: 1
    select  cli.cod_cli, cli.nome, cli.telefone
    from cli
    where cli.cod_cli=120 ;
FONTE 1 ENVIA RESPOSTA A12
A12[1]:
    cli.cod_cli : 120
    cli.nome : Carla Ferraz
    cli.telefone : 0412663554

ATUALIZA A1 COM BASE NOS DADOS DE A12
A1[1]:
Tipo de atualizacao: 1 - insercao
    cli.cod_cli : 120
    cli.nome : Carla Ferraz
    cli.telefone : 0412663554
    cc.cod_cc : 10120
    cc.saldo_cc : 650,00
    poup.cod_poup : 20120
    poup.saldo_poup : 300,00

ATUALIZA AL COM OS DADOS DE  A1 (AL = AL + A1)
-----------------------------------------------------
AL - LISTA DE ACOES A SER APLICADA NO DATA WAREHOUSE
-----------------------------------------------------
AL[1]:
  INSERT INTO BANCO
  VALUES ('120','Carla Ferraz','0412663554','10120','650,00','20120',
         '300,00');
```

```
-------------------------------------------------
MENSAGEM DE ATUALIZACAO EM PROCESSAMENTO: U2
-------------------------------------------------
Ocorrida na Relacao: cc
Tipo de Atualizacao: 1 - insercao
Informacoes atualizadas
    Campo: cc.cod_cc, Informacao: 10121
    Campo: cc.cod_cli, Informacao: 120
    Campo: cc.saldo_cc, Informacao: 250,00
INICIALIZA A RESPOSTA A2
A2[1]:
Tipo de atualizacao: 1 - insercao
    cli.cod_cli :
    cli.nome :
    cli.telefone :
    cc.cod_cc : 10121
    cc.saldo_cc : 250,00
    poup.cod_poup :
    poup.saldo_poup :

GERA A CONSULTA Q2 REFERENTE A ATUALIZACAO U2
    select  cli.cod_cli, cli.nome, cli.telefone, poup.cod_poup,
            poup.saldo_poup
    from cli, poup
    where cli.cod_cli=120, 120=poup.cod_cli ;

PROXIMA FONTE A ENVIAR A CONSULTA Q2: 1
ENVIA CONSULTA Q21 PARA A FONTE: 1
    select  cli.cod_cli, cli.nome, cli.telefone
    from cli
    where cli.cod_cli=120 ;
FONTE 1 ENVIA RESPOSTA A21
A21[1]:
    cli.cod_cli : 120
    cli.nome : Carla Ferraz
    cli.telefone : 0412663554

ATUALIZA A2 COM BASE NOS DADOS DE A21
A2[1]:
Tipo de atualizacao: 1 - insercao
    cli.cod_cli : 120
    cli.nome : Carla Ferraz
    cli.telefone : 0412663554
    cc.cod_cc : 10121
    cc.saldo_cc : 250,00
    poup.cod_poup :
    poup.saldo_poup :

PROXIMA FONTE A ENVIAR A CONSULTA Q2: 3
ENVIA CONSULTA Q22 PARA A FONTE: 3
    select  poup.cod_poup, poup.cod_cli, poup.saldo_poup
    from poup
    where 120=poup.cod_cli ;
FONTE 3 ENVIA RESPOSTA A22
A22[1]:
    poup.cod_poup : 20120
    poup.cod_cli : 120
    poup.saldo_poup : 300,00
```

```
ATUALIZA A2 COM BASE NOS DADOS DE A22
A2[1]:
Tipo de atualizacao: 1 - insercao
    cli.cod_cli : 120
    cli.nome : Carla Ferraz
    cli.telefone : 0412663554
    cc.cod_cc : 10121
    cc.saldo_cc : 250,00
    poup.cod_poup : 20120
    poup.saldo_poup : 300,00

ATUALIZA AL COM OS DADOS DE  A2 (AL = AL + A2)

-----------------------------------------------------
AL - LISTA DE ACOES A SER APLICADA NO DATA WAREHOUSE
-----------------------------------------------------
AL[1]:
  INSERT INTO BANCO
  VALUES ('120','Carla Ferraz','0412663554','10120','650,00','20120',
         '300,00');
AL[2]:
  INSERT INTO BANCO
  VALUES ('120','Carla Ferraz','0412663554','10121','250,00','20120',
         '300,00');
```

A.3. Conjunto de Dados de Teste 3

A Figura A.5 apresenta o estado inicial dos dados das relações CLI, CC e POUP, residentes nas fontes de dados 1, 2 e 3, respectivamente. A Figura A.6 apresenta o estado inicial da visão do *warehouse* BANCO.

São consideradas duas atualizações que ocorrem de forma concorrente, nas fontes de dados 2 e 3, sendo elas:

1. U_1 = insert(cc, [10100; 100; 300,00]);
2. U_2 = delet(poup, [20100;100;1500,00]).

CLI	cod_cli	nome	rg	telefone
	100	João Azevedo	9999999-5	412536611
	110	Luiza do Amaral	8888888-6	412650311
	120	Carla Ferraz	7777777-4	412663554

CC	cod_cc	cod_cli	saldo_cc
	10110	110	250,50
	10120	120	650,00

POUP	cod_poup	cod_cli	saldo_poup
	20100	100	1500,00
	20110	110	2250,50
	20120	120	350,00

Figura A.5: Estado inicial das relações - conjunto de dados de teste 3

BANCO	cod_cli	nome	telefone	cod_cc	saldo_cc	cod_poup	saldo_poup
	110	Luiza do Amaral	412650311	10110	250,50	20110	2250,50
	120	Carla Ferraz	412663554	10120	650,00	20120	350,00

Figura A.6: Estado inicial do *warehouse* - conjunto de dados de teste 3

A.3.1. Saída Gerada para o Conjunto de Dados de Teste 3

```
DEFINICAO DA VISAO DO DATA WAREHOUSE
   create view  banco
      select  cli.cod_cli, cli.nome, cli.telefone, cc.cod_cc, cc.saldo_cc,
              poup.cod_poup, poup.saldo_poup
      from cli, cc, poup
      where cli.cod_cli=cc.cod_cli, cc.cod_cli=poup.cod_cli ;

---------------------------------------------------
MENSAGEM DE ATUALIZACAO EM PROCESSAMENTO: U1
---------------------------------------------------
Ocorrida na Relacao: cc
Tipo de Atualizacao: 1 - insercao
Informacoes atualizadas
    Campo: cc.cod_cc, Informacao: 10100
    Campo: cc.cod_cli, Informacao: 100
    Campo: cc.saldo_cc, Informacao: 300,00

GERA A CONSULTA Q1 REFERENTE A ATUALIZACAO U1
    select  cli.cod_cli, cli.nome, cli.telefone, poup.cod_poup,
            poup.saldo_poup
    from cli, poup
    where cli.cod_cli=100, 100=poup.cod_cli ;

INICIALIZA A RESPOSTA A1
A1[1]:
Tipo de atualizacao: 1 - insercao
    cli.cod_cli :
    cli.nome :
    cli.telefone :
    cc.cod_cc : 10100
    cc.saldo_cc : 300,00
    poup.cod_poup :
    poup.saldo_poup :

MONITOR DO DATA WAREHOUSE RECEBE MENSAGEM DE ATUALIZACAO
FILA DE MENSAGENS DE ATUALIZACAO - (MQ)
Atualizacao U2
Ocorrida na Relacao: poup
Tipo de Atualizacao: 2 - exclusao
Informacoes atualizadas
    Campo: poup.cod_poup, Informacao: 20100
    Campo: poup.cod_cli, Informacao: 100
    Campo: poup.saldo_poup, Informacao: 1500,00

PROXIMA FONTE A ENVIAR A CONSULTA Q1: 1
ENVIA CONSULTA Q11 PARA A FONTE: 1
    select  cli.cod_cli, cli.nome, cli.telefone
    from cli
    where cli.cod_cli=100 ;
FONTE 1 ENVIA RESPOSTA A11
A11[1]:
    cli.cod_cli : 100
    cli.nome : Joao Azevedo
    cli.telefone : 0412536611
```

```
ATUALIZA A1 COM BASE NOS DADOS DE A11
A1[1]:
Tipo de atualizacao: 1 - insercao
    cli.cod_cli : 100
    cli.nome : Joao Azevedo
    cli.telefone : 0412536611
    cc.cod_cc : 10100
    cc.saldo_cc : 300,00
    poup.cod_poup :
    poup.saldo_poup :

PROXIMA FONTE A ENVIAR A CONSULTA Q1: 3
ENVIA CONSULTA Q12 PARA A FONTE: 3
    select  poup.cod_poup, poup.cod_cli, poup.saldo_poup
    from poup
    where 100=poup.cod_cli ;
FONTE 3 ENVIA RESPOSTA A12

FONTE 3 ENVIOU A RESPOSTA A12 VAZIA
FORAM ENCONTRADAS ATUALIZACOES CONCORRENTES NA RELACAO: poup
COMPENSA EFEITO DAS ATUALIZACOES CONCORRENTES EM A12
A12[1]:
    poup.cod_poup : 20100
    poup.cod_cli : 100
    poup.saldo_poup : 1500,00

ATUALIZA A1 COM BASE NOS DADOS DE A12
A1[1]:
Tipo de atualizacao: 1 - insercao
    cli.cod_cli : 100
    cli.nome : Joao Azevedo
    cli.telefone : 0412536611
    cc.cod_cc : 10100
    cc.saldo_cc : 300,00
    poup.cod_poup : 20100
    poup.saldo_poup : 1500,00

ATUALIZA AL COM OS DADOS DE  A1 (AL = AL + A1)

-----------------------------------------------------
AL - LISTA DE ACOES A SER APLICADA NO DATA WAREHOUSE
-----------------------------------------------------
AL[1]:
 INSERT INTO BANCO
 VALUES ('100','Joao Azevedo','0412536611','10100','300,00','20100',
        '1500,00');

-----------------------------------------------------
MENSAGEM DE ATUALIZACAO EM PROCESSAMENTO: U2
-----------------------------------------------------
Ocorrida na Relacao: poup
Tipo de Atualizacao: 2 - exclusao
Informacoes atualizadas
    Campo: poup.cod_poup, Informacao: 20100
    Campo: poup.cod_cli, Informacao: 100
    Campo: poup.saldo_poup, Informacao: 1500,00

U2 REFERE-SE A UMA ATUALIZACAO DE EXCLUSAO
ATUALIZA AL COM DADOS DE U2 (AL = AL+ U2)
```

```
-----------------------------------------------------
AL - LISTA DE ACOES A SER APLICADA NO DATA WAREHOUSE
-----------------------------------------------------
AL[1]:
 INSERT INTO BANCO
 VALUES ('100','Joao Azevedo','0412536611','10100','300,00','20100',
        '1500,00');
AL[2]:
 DELETE FROM BANCO
 WHERE poup.cod_poup = '20100';
```