MAC 5701 -Tópicos em Ciência da Computação

Algoritmos para Consistência de Atualizações em Datawarehouse

Bianka Maria Moura Teixeira Gonçalves

Orientador: Prof Dr. João Eduardo Ferreira

Departamento de Ciência da Computação

Instituto de Matemática e Estatística

Universidade de São Paulo

2 de julho de 2001

Resumo

A maioria dos sistemas comerciais de warehousing, foca no armazenamento de dados para acesso eficiente e em facilidades de consultas no warehouse. Eles ignoram o problema de consistência de integrar novos dados assumindo que estes acontecem "off-line", ou seja quando as consultas não estão sendo executadas.Muitas das aplicações de warehouse possuem operações de sincronismo nas quais o domínio da aplicação existente não exige uma atualização rigorosa dos dados. Com a evolução da representação semântica dos dados em ambientes operacionais, as análises realizadas no Data Warehouse passaram a ter necessidades outras de sincronismo. Por isso existe um crescente interesse em sistemas de warehouse que possam absorver as atualizações(dados operacionais)sem parar o processo de consulta(dados analíticos). Com esses sistemas poderia se diminuir o tempo de sincronização entre os dados analíticos e os operacionais tornando as consultas no warehouse cada vez mais confiáveis.

1 Introdução

Um data warehouse é um repositório de informações integradas de fontes distribuídas, autônomas e possivelmente heterogêneas para consultas e análises eficientes [ZGMW98]. Um data warehouse é um conjunto de bases de dados orientadas a conteúdo, integradas, variantes no tempo e não voláteis.O termo orientado a conteúdo reflete o fato que as informações no warehouse estão dispostas de acordo com a área de negócios da organização que o utiliza. O termo integrado refere-se ao fato que o warehouse integra dados derivados de vários sistemas funcionais diferentes da organização e provê uma única e consistente visão de toda a situação da organização. O termo variante no tempo significa que o warehouse é capaz de representar variação temporal dos dados, e as consultas podem ser baseadas em períodos de tempo. Os dados são não-voláteis, dado que eles sempre são incluídos no warehouse e raramente são apagados do mesmo mantendo todo o histórico da organização [MSRK98]. Uma visão é uma relação derivada em termos das relações base armazenadas. Deste modo uma visão define uma função de um conjunto de tabelas bases para uma tabela derivada. Esta função é tipicamente recomputada toda vez que a visão é referenciada. Uma visão é materializada quando armazenamos as tuplas da visão no data warehouse. Consequentemente, o acesso a visão materializada pode ser bem mais rápido que recomputar toda a visão. O conteúdo do warehouse pode ser utilizado para tomada de decisões e análise, para isso são utilizadas sobre ele ferramentas OLAP (On-Line Analytical Processing). Essas ferramentas apresentam os dados para as aplicações dos usuários de maneira multi-dimensional, podendo assim serem construídas consultas complexas que atendem as necessidades da organização.

A arquitetura de um warehouse é descrito pela figura 1.

Em cada fonte, um mediador coleta os dados e os envia para o warehouse. Os mediadores são os responsáveis por identificar as mudanças ocorridas nas fontes, transformá-las e filtrá-las. Se a fonte possui *triggers* relacionais(programas que repassam as informações) os mediadores podem simplesmente repassar a informação. No warehouse o integrador recebe os dados das fontes, executa alguma integração ou translação necessária, adiciona alguma informação extra tais como rótulos de tempo para análises históricas, e então o warehouse é informado para que faça o armazenamento dos dados. O integrador combina os dados enviados pelas diferentes fontes, resolvendo os conflitos entre os dados e mantendo os dados na forma consistente.Como efeito o data warehouse armazena uma visão materializada das fontes [ZGMW98].O metadata contém informações adicionais sobre a criação, gerenciamento e uso do warehouse. Ele serve como uma ponte entre os usuários do warehouse e os dados nele contidos. Os dados do warehouse são acessados pelo servidor de OLAP para dispor os dados de maneira multidimensional para as aplicações de *front-end* [1] tais como (ferramentas de análise, gerador de relatórios, planilhas eletrônicas, ferramentas de data mining). Basicamente, o servidor de OLAP interpreta as consultas das aplicações clientes e converte-as em complexas consultas SQL requeridas para acessar o data warehouse. Ele pode também acessar os dados das fontes de dados se as consultas

[1] aplicações que são utilizadas pelos usuários

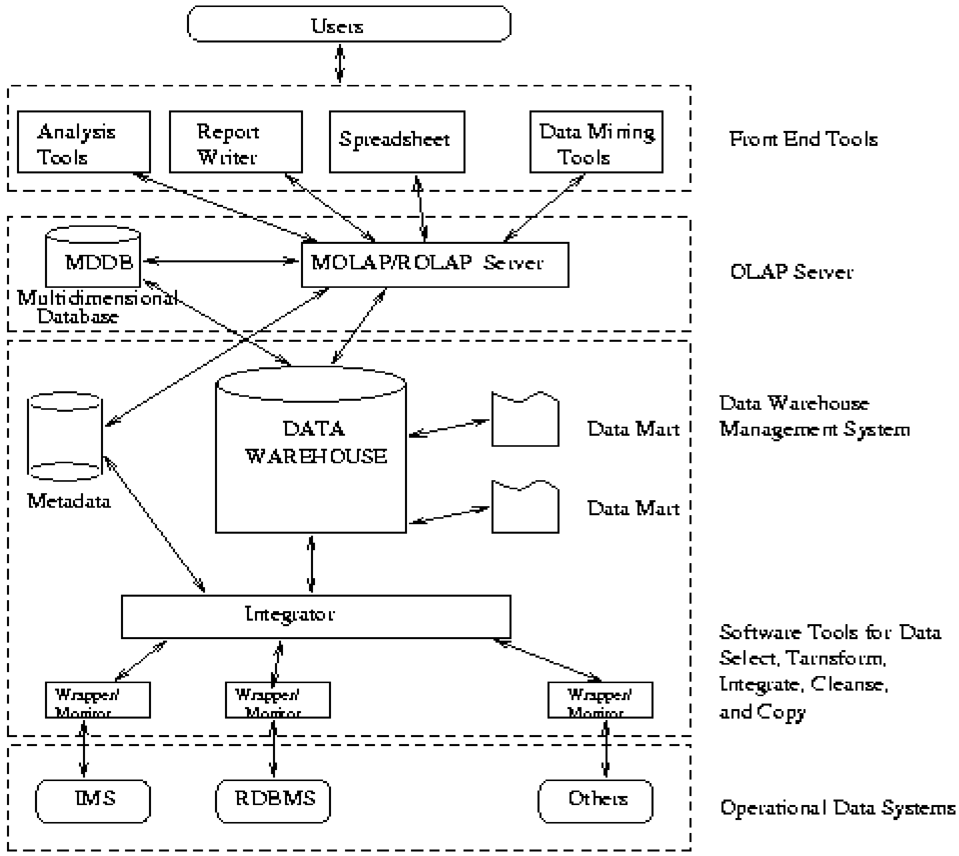

Figura 1: Arquitetura de Warehouse

assim necessitarem.Quando as consultas são respondidas o servidor OLAP passa as visões multidimensionais para as ferramentas de *front-end* e estas irão formatar os dados de acordo com as necessidades das aplicações clientes [SMKK99].

Após o processo inicial de carga do warehouse, ou seja seu povoamento, mudanças irão ocorrer nas fontes e as visões do warehouse ficarão desatualizadas em relação as fontes de dados. Para manter as visões do warehouse consistentes, as visões devem ser incrementalmente mantidas propagando as mudanças das fontes para as visões do warehouse. Assume-se a técnica onde apenas uma parte da visão muda em resposta a atualização nas relações bases, pois na maioria das vezes torna-se muito caro recomputar todas as visões do warehouse a partir do início para cada atualização ocorrida [GMS].

A manutenção de visões materializadas difere da manutenção de visões tradicional no fato que a definição da visão e as bases de dados estão desacopladas [ZGMHW95]. Usualmente as mudanças nas fontes de dados são propagadas periodicamente para as visões [QW97]. O processo de carga segundo [AV98] e o fator de maior complexidade no warehouse, sendo que este fato permanece imutável. Nos sistemas de warehouse comerciais todas as alterações ocorridas nas fontes são acumuladas e periodicamente propagadas para o warehouse (por exemplo diariamente). Essas alterações estão contidas em uma grande transação de atualização em lote(*batch*) chamada de transação de manutenção. A transação de manutenção é tipicamente a única transação de atualização para as visões do warehouse [QW97]. Os warehouses comerciais focam no armazenamento de dados para acesso eficiente e em facilidades de consultas. Desta maneira devem executar as consultas e a transação de manutenção de forma que não gerem dados inconsistentes e não bloqueiem uma a outra. Para evitar esses problemas quando se executa a transação de manutenção o warehouse fica indisponível para consultas(*off-line*)e as fontes para atualizações [ZGMW98]. Com a finalização do processo de carga (manutenção da visão) o warehouse conterá todas as atualizações geradas nas fontes de dados.

Muitos das aplicações de warehouse possuem operações de sincronismo nas quais o domínio da aplicação existente não exige uma atualização rigorosa dos dados podendo ser usados os warehouse convencionais. Porém com a evolução da representação semântica dos dados em ambientes operacionais, as análises realizadas no Data Warehouse passaram a ter necessidades outras de sincronismo. Entre alguns problemas que justificam esta nova necessidade pode-se citar a globalização das companhias não permitindo que o warehouse fique indisponível por uma grande janela de tempo, e o limitante da mesma janela ao número e ao tamanho das visões a serem atualizadas já que o tempo de atualização é reduzido. Por isso existe um crescente interesse em sistemas de warehouse que possam absorver as atualizações dos dados operacionais sem interromper o processo de consultas aos dados analíticos, através dos novos algoritmos de sincronismo apresentados nesse trabalho poderia-se diminuir o tempo de sincronização entre os dados analíticos e os operacionais tornando as consultas no warehouse cada vez mais confiáveis e sincronizadas com os dados operacionais.

2 Manutenção de Visão

Quando os dados nos sistemas operacionais sofrem alterações, o data warehouse precisa ser atualizado de acordo com tal mudança. O processo de atualizar a visão materializada em resposta as mudanças nas fontes de dados é chamado de *manutenção de visão*. A manutenção de visão em ambientes distribuídos faz com que apareçam inconsistências já que existe um espaço de tempo finito e limitado necessário para propagar as mudanças das fontes de dados para o data warehouse e para computar as visões.

De uma forma geral o sincronismo de dados pode ser mantido em três passos segundo [SMKK99]:

1. Propagando as mudanças das fontes de dados para o data warehouse para garantir que cada visão represente um estado consistente da base de dados;
2. Computar as visões em resposta a essas mudanças usando o estado das fontes em que foi propagado a atualização;
3. Instalar a atualizações da visão no data warehouse na mesma ordem que as mudanças ocorreram nas fontes.

   As inconsistências vão aparecer no warehouse desde que as mudanças nas fontes sejam randômicas e dinâmicas. Antes que o warehouse esteja pronto para computar a atualização da visão para mudanças já ocorridas, o estado das fontes já pode ter mudado e esse fato pode causar inconsistência nos dados. Para exemplificar o problema da inconsistência analisaremos o exemplo abaixo proposto por [ZGMW98]. Neste exemplo por questão de simplificação assume-se que todas as fontes utilizam o modelo entidade-relacionamento e a visão materializada mantida no warehouse contém as chaves participantes de cada relação. Cada atualização é tratada como uma transação em cada uma das fontes de dados.

2.1 Exemplo 1

Seja V a visão do warehouse definida como sendo V= (r1 $\bowtie$ r2 $\bowtie$ r3) onde r1, r2 e r3 são relações residentes nas fontes x, y, z. Inicialmente as relações estão inicializadas como:

	A	B		B	C		C	D
r1	1	2	r2	-	-	r3	3	4

A visão materializada no warehouse é $MV = \phi$. Vamos considerar duas atualizações das fontes U_1= *insert(r2, [2,3])* e U_2= *delete(r1,[1,2])*. Usando um algoritmo de manutenção incremental convencional [BLT86], os seguintes eventos podem ocorrer no warehouse:

1. O warehouse recebe a atualização U_1= *insert* [2]*(r2,[2,3])* da fonte y. Ele gera uma consulta Q_1= r1 $\bowtie$ [2,3] $\bowtie$ r3. Para responder a Q_1 primeiro o warehouse envia consulta Q_1^1= r1 $\bowtie$ [2,3] para fonte x.

2. O warehouse recebe a resposta A_1^1= [1,2,3] da fonte$x$. A consulta Q_1^2= [1,2,3] $\bowtie$ r3 é enviada para a fonte z para ser respondida.

3. O warehouse recebe U_2= *delete* [3]*(r1, [1,2])* da fonte x. Como a visão corrente no warehouse é vazia, nenhuma ação é tomada para este *delete*.

4. O warehouse recebe A_1^2= [1,2,3,4] da fonte z, que é a resposta final para Q_1. Como não há consultas pendentes ou atualizações, a resposta é inserida na MV e então MV= [1,2,3,4], sendo que esta visão final é incorreta.

No exemplo acima o intercalamento da consulta Q_1 com as atualização U_2 chegando da fonte x causou a visão incorreta.

2.2 Métodos para se Previnir a Inconsistência em Data Warehouse

Existem algumas maneiras de se prevenir a inconsistência dos dados:

- A primeira maneira consiste em tornar as visões "self-maintenance" [GM95, GMS], ou seja elas fazem "auto-manutenção" sem que haja comunicação instantânea com as fontes de dados.Uma das técnicas para alcançar este objetivo é fazer com que a manutenção da visão se torne local ao warehouse. Para isso é necessário armazenar cópias de todas relações no warehouse. Desta maneira as atualizações poderiam ser feitas diretas no warehouse. Esta técnica possui um série de desvantagens: o espaço de armazenamento no warehouse para guardar todas as relações seria muito alto e poderiam aparecer inconsistências por causa de redundância de dados. Além do mais dependendo do tipo de dados armazenados como por exemplo os dados de

[2]evento que significa uma inserção de tupla ocorrida em uma base de dados. Será utilizada a palavra em inglês devido ser mais comum a sua ocorrência e entendimento.

[3]evento que significa apagar uma tupla de uma base de dados, será utilizada a palavra em inglês pelo mesmo motivo de *insert*.

companhias, o warehouse teria de armazenar os dados de todas as companhias no mundo, pois se forem copiadas apenas as companhias de interesse no momento, no futuro poderia não se conseguir responder um consulta sobre uma nova companhia. O warehouse teria também que integrar os dados de todas as fontes não somente dos dados que lhe são interessantes.

- A segunda maneira para prevenir inconsistências é executar cada atualização e todas as ações necessárias para a sua integração incremental no warehouse como uma transação distribuída envolvendo o warehouse e todas as fontes de dados com quem se relaciona. Porém transações distribuídas requerem um mecanismo de controle de concorrência global envolvendo todas as fontes, e tal mecanismo pode não existir. Mesmo que este mecanismo exista, as fontes devem estar preparadas para tolerar os atrasos que aparecem com o controle de concorrência global.

- Uma terceira maneira de se garantir a consistência de dados, é executar as atualizações de forma incremental "on-line" ou seja no momento que as atualizações ocorrem nas fontes elas são propagadas para o warehouse. Como mostrado acima essa estratégia pode gerar inconsistências nos dados. É nesta manutenção de sincronismo de dados que esta o enfoque dos algoritmos descritos a seguir. Cada atualização que chega deve ser integrada com os dados de outras fontes antes de ser armazenada no warehouse e outras atualizações que são submetidas ao data warehouse até o fim deste processo devem ser tratadas de forma a não gerar resultados inválidos para a atualização anterior.

3 Algoritmos para Manutenção de Consistência

3.1 Algoritmo ECA [ZGMHW95]

O algoritmo de Compensação Eager (ECA) evita recomputar a relação e armazenar cópias das relações base no warehouse. A idéia básica do algoritmo é adicionar às consultas enviadas para as fontes, consultas de compensação para corrigir às atualizações concorrentes.Nesta solução assume-se que as mensagens são entregues na ordem, ou seja quando uma atualização chega ao warehouse e são necessárias informações adicionais das fontes, esta consulta as fontes tem precedência sobre novas atualizações que chegam. Desta maneira o algoritmo consegue detectar as anomalias que irão ocorrer caso outra atualização chegue antes da resposta a consulta. É também assumido que as fontes manuseiam pedidos feitos pelas consultas automaticamente.

3.1.1 Definições de Conceitos Básicos

Para entender melhor os algoritmos da família ECA torna-se necessário estabelecer alguns conceitos.

- As visões consideradas pelos algoritmos ECA são formadas por um conjunto de *select-project-join(SPJ)* da forma V= (Π $proj(\sigma cond($r1 $\bowtie$ r2 $\bowtie$ ... rn)) onde *proj* é o conjunto de nomes de atributos, *cond* é a expressão booleana e *ri* são as relações que por motivo de simplificação são consideradas distintas.

- O algoritmo assume que para fazer a manutenção de visão é utilizada a técnica de atualização imediata.

- O algoritmo explora a semântica das aplicações(visões relacionais) para garantir um certo tipo consistência sem o uso dos locks [4] tradicionais. Em [BGMS92] encontra-se trabalhos relacionadas ao assunto.

3.1.2 Eventos

São seqüências de operações que ocorrem no mesmo *site* [5]. Assume-se que os eventos são atômicos, isto é existe uma mecanismo de controle de concorrência local em cada site que garante que operações conflitantes não se sobreponham.

[4] termo usado em controle de concorrência que significa que um recurso encontra-se alocado (travado) para uma determinada transação

[5] local onde são gerados os eventos, podendo ser fontes ou data warehouse

Além disso dentro de um evento as ações ocorrem sempre realizando primeiro a ação no site depois enviando a notificação ou resposta a esta ação. Existem dois tipos de eventos de fonte e dois tipos de eventos de warehouse:

Eventos de Fonte

S_up: A fonte executa uma atualização U e envia notificação de atualização para o warehouse.

S_qu: a fonte executa a consulta Q usando o estado corrente das relações bases e envia a relação resposta A para o warehouse.

Eventos do Warehouse

W_up: O warehouse recebe uma atualização U gera uma consulta Q e envia Q para a fonte onde será executada.

W_ans: o warehouse recebe a relação resposta A da consulta Q e atualiza a visão baseada em A.

- Um evento arbitrário é denotado por "e", "se" denota um evento de fonte, "we" denota um evento da warehouse. Tipos de evento são subscritos para indicar um evento específico, Ex: S_up_i, W_up_j. Para cada evento "e", informações relevantes sobre o evento são denotadas por query(e), update(e), e answer(e) que denotam respectivamente a consulta, a atualização e resposta associada a este evento. Se o evento "e" é causado por outro evento então trigger(e) denota o evento que disparou "e". O estado dos dados após um evento é denotado por state(e).

- Existe um operador binário de evento "<" que significa *ocorre antes de* ;

- Assume-se que as mensagens são entregues em ordem e são processadas em ordem;

- Os estado de uma fonte após um evento se_j é ss_j, o estado do warehouse após os evento trigger we_i é ws_i ;

- No warehouse o estado de uma visão materializada MV após um evento we_i é dado por $V[ws_i]$ onde V é a definição de visão;

- Nas fontes $Q[ss_i]$ é o resultado de se responder uma consulta Q nas relações bases existentes após o evento se_i;

3.1.3 Niveis de Correção (Correctness)

Quando se define a noção de convergência é considerada que as execuções são finitas, ou seja existe uma última atualização U_n, e últimos eventos se_p e we_k. Desta forma o algoritmo para manutenção de visão de warehouse deve exibir as seguintes propriedades abaixo:

- Convergência: para todas as finitas execuções $V[ws_q] = V\ [ss_p]$. Que significa que após a última atualização e após todas atividades terem cessado, a visão é consistente com as relações da fonte.

- Consistência Fraca: Para todas execuções e para todo ws_i, existe um ss_j tal que $V[ws_i] = V[ss_j]$, ou seja para cada estado do warehouse corresponde a um estado válido das relações da fonte.

- Consistência: Para todas as execuções e para todo $ws_i \leq ws_j$ existe $ss_k < ss_l$ tal que $V[ws_i] = V[ss_k]$ e $V[ws_j] = V[ss_l]$, ou seja para cada estado da visão corresponde um estado válido da fonte e na ordem correspondente.

- Consistência Forte: Consistência e Convergência.

- Completude (Completeness): Consistência forte e para todo ss_i existe um ws_j tal que $V[ws_j] = V[ss_i]$, ou seja existe um completa preservação de ordem mapeada entre os estados da visão e os estados da fonte. Apesar de propriedade ser importante acredita-se que esta é muito forte e na prática é desnecessária nos ambientes de warehouse.

3.1.4 Sinais e Consultas

- Serão tratados dois tipos de atualização: inserções(*insert*) e deleções(*delete*). As alterações podem ser representadas por deleções [6] seguidas de inserções.

- São anexados sinais as tuplas, onde "+" significa uma tupla existente ou a ser inserida e "−" uma tupla a ser deletada. Os sinais das tuplas são propagados através das operações relacionais. Esta técnica é similar a usada em [BLT86].

- Existem dois operadores binários "+" e "−" que operam sobre a relações com sinais nas tuplas. Os operadores são comutativos e associativos. O produto cartesiano x é distributivo sobre "+" e "−".

[6] é a ação que se executa ao se remover uma tupla de uma base de dados

- Para manter a visão sobre as relações o algoritmo ECA vai gerar consultas que contém uma coleção de termos na seguinte forma:

  T=$\Pi proj(\sigma cond(\H{r}_1 \bowtie \H{r}_2 \bowtie ... \H{r}_n))$ onde $\H{r}_i$ é uma relação ou uma tupla de atualização *ti* de *ri*. A consulta é formada pelo somatório dos termos:

$$Q= \sum_i Ti$$

3.1.5 Descrição Algoritmo ECA

O algoritmo ECA é um algoritmo de manutenção de visão incremental baseado no algoritmo de manutenção de visão centralizado em [BLT86]. O algoritmo ECA antecipa as anomalias que aparecem devido o desasacoplamento entre as atualizações da fontes de dados da base e a modificação da visão efetuando compensações para as anomalias para garantir a correta manutenção da visão. Neste algoritmo o qwarehouse possui um conjunto consultas "pendentes"que é a seguir definido.

Quando um evento we é processado no warehouse, seja o conjunto de consultas não respondidas de we (*UQS(we)* - unanswered query set)o conjunto de todas as consultas que foram enviadas pelo warehouse antes que we ocorresse mas cujas respostas não foram recebidas antes de we chegasse ao warehouse. Quando o warehouse receber uma atualização Ui e o conjunto UQS não for vazio, esta atualização pode causar o resultado errado das consultas Qj que estão em UQS aguardando resposta. Essa incorreção aparece pois as consultas Qj deveriam ter sido computadas antes de Ui porém foram computadas depois de Ui. Dessa forma todas as consultas em UQS vão "enxergar" um estado de fonte que já reflete U_i. Para corrigir esse possível problema o algoritmo ECA incorpora uma "consulta de compensação"para cada consulta contida em UQS. Essas consultas compensatórias corrigem os efeitos da atualização Ui nas consultas existentes em UQS. Uma importante conseqüência de usar consultas compensatórias é que os resultados de uma consulta só pode ser aplicados a visão quando o resultado da mesma e de todas as consultas compensatórias tenham sido recebidas.O ECA coleta as respostas intermediárias em um relação temporária chamada COLLECT, e apenas atualiza a visão quando todas as resposta das consultas de UQS tenham sido recebidas ou seja UQS= ϕ. Desta maneira ele previne estados inválidos de visões.

3.2 Algoritmo ECA

Na Fonte:

S_up$_i$: executa U$_i$
Envia U$_i$ para o warehouse
Trigger evento W_up$_i$ no warehouse
S_qu$_i$: recebe consulta Q$_i$
Faz A$_i$= Q$_i$[ss$_i$] (onde ss$_i$ é o estado corrente)
Envia A$_i$ para o warehouse
Trigger evento W_ans$_i$ no warehouse

No warehouse:

W_up$_i$: recebe U$_i$
Faz Q$_i$= V$<$ U$_i$ $>$ - $\Sigma_{Q_j \in UQS}$ Q$_j$$<U_i$$>$
Envia Q$_i$ para a fonte
Trigger evento S_qu$_i$ na fonte
W_ans$_i$: recebe resposta da consulta A$_i$
Faz COLLECT= COLLECT + A$_i$;
Se UQS= ϕ então
{
$MV \longleftarrow MV + COLLECT$
$COLLECT \longleftarrow \phi$
}
else não faz nada

Fim

3.2.1 Exemplo

Sejam as relações nas fontes

	W	X
r1	1	2

	X	Y
r2	-	-

	Y	Z
r3	-	-

Seja a visão da fonte definida por V= Πw(r1 $\bowtie$ r2 $\bowtie$ r3). Inicialmente a visão materializada (MV) e COLLECT estão vazios.

1. O warehouse recebe o evento U_1= *insert*(r1,[4,2]). O warehouse envia uma consulta Q_1=V<U_1>= Π w([4,2] $\bowtie$ r2 $\bowtie$ r3)

2. O warehouse recebe U_2= *insert*(r3,[5,3]) como UQS= Q_1 o warehouse envia Q_2=V< U_2> - Q_1< U_2>= Π w(r1 $\bowtie$ r2 $\bowtie$ [5,3]) - Πw([4,2] $\bowtie$ r2 $\bowtie$ [5,3])

3. O Warehouse recebe U_3= *insert*(r2,[2,5]) como UQS= { Q_1, Q_2 } o warehouse envia Q_3= V< U_3> - Q_1< U_3> - Q_2< U_3>= Πw(r1 $\bowtie$ [2,5] $\bowtie$ r3) - Πw([4,2] $\bowtie$ [2,5] $\bowtie$ r3) - Πw((r1-[4,2]) $\bowtie$ [2,5] $\bowtie$ [5,3])

4. O warehouse recebe A_1= [4]. COLLECT= COLLECT +([4])= ([4]), UQS= {Q_2,Q_3}

5. O warehouse recebe A_2= [1]. COLLECT= ([4]) + ([1])= ([1],[4]), UQS = {Q_3}

6. O warehouse recebe A_3= ϕ. COLLECT= ([1],[4]) + f= ([1],[4]), UQS= ϕ

7. O warehouse atualiza a visão MV= ϕ + COLLECT= ([1],[4]).

O algoritmo ECA garante a consistência forte porém não é completo, pois a completude requer que todo os estado da fonte seja refletido em algum estado da visão. Mas no algoritmo algum estado das fontes pode ser perdido enquanto o algoritmo recolhe as respostas das consultas.

3.3 Algoritmo ECA-Key

O algoritmo ECA^K é uma expansão do ECA onde os atributos na lista de projeção da definição da visão contém atributos chaves para cada uma das tabelas da base. O algoritmo ECA^K funciona da seguinte maneira:

1. COLLECT é inicializado com a visão materializada corrente. Ao invés de armazenar as modificações na MV, COLLECT é uma cópia de trabalho da MV

2. Quando um *delete* é recebido no warehouse, nenhuma consulta é enviada para as fontes, ao invés disso o *delete* é aplicado ao COLLECT imediatamente usando uma operação especial chamada de *key-delete*(A,r1,[x,y]) . Esta operação apaga todas as tuplas contidas na relação r1, que esta contida em A, onde atributo chave da tupla [x,y] é igual a chave de r1.

3. Quando um *insert* é recebido no warehouse uma consulta é enviada para a fonte. Porém não são adicionadas consultas compensatórias.

4. A medida que as respostas são recebidas elas são acumuladas no conjunto COLLECT como no ECA original porém sem tuplas duplicadas. A garantia de não se ter tuplas duplicadas e que a visão possui as chaves para todas as relações bases.

5. Quando o UQS esta vazio a visão materializada é atualizada sobrepondo-a com as tuplas em COLLECT.

3.3.1 Exemplo

Sejam as relações nas fontes

	W	X			X	Y
r1	1	2		r2	2	3

Seja a visão da fonte definida por V= Π w,y(r1 $\bowtie$ r2). Inicialmente a visão materializada(MV)= ([1,3]) e COLLECT=MV.

1. O warehouse recebe o evento U_1= *insert*(r2,[2,4]). O warehouse envia uma consulta Q_1=V< U_1>= Πw,y(r1 $\bowtie$ [2,4])

2. O warehouse recebe U_2= *insert*(r1,[3,2]) Q_2=V< U_2=Πw,y([3,2] $\bowtie$ r2).

3. O warehouse recebe U_3= *delete*(r1,[1,2]). A operação *key-delete* deleta as tuplas da forma [1,x] obtendo COLLECT= COLLECT - ([1,3])= ϕ, UQS= $\{Q_1,Q_2\}$

4. O warehouse recebe A_1= [3,4]. COLLECT= COLLECT +([3,4])= ([3,4]), UQS= $\{Q_2\}$

5. O warehouse recebe A_2= ([3,3],[3,4]).COLLECT= ([3,4]) + [3,3])= ([3,3],[3,4]). Como ([3,4]) é uma tupla duplicada ela não é adicionado ao COLLECT. Como UQS= ϕ, MV é ajustado pelo COLLECT então MV= ([3,3],[3,4]). E o COLLECT é setado para vazio.

3.4 Algoritmo ECA-Local (ECA)

Este algoritmo combina as consultas compensatórias do algoritmo ECA com as atualizações locais do ECA^K para produzir um algoritmo que aplica-se as visões gerais. No ECA^L cada atualização é tratada tanto local com não-local dependendo da definição particular da visão e as atualizações das relações base. Porém manter a ordem correta de execução entre o processamento local e não-local não é uma tarefa simples.

3.5 Propriedades da Família ECA

- São incrementais, ou seja cada atualização no warehouse é baseado numa atualização da fonte ao invés de recomputar toda a visão do warehouse

- Eles não adicionam carga extra a fonte como locks e rótulos de tempo.

- Quando a freqüência de atualização é baixa o algoritmo ECA se comporta da mesma maneira que o algoritmo incremental de visão materializada de [BLT86] pois as consultas compensatórias são usadas apenas quando a resposta de uma consulta ainda não foi recebida antes que a próxima atualização ocorra nas fontes.

3.6 Família de Algoritmos Strobe [ZGMW98]

3.6.1 Ambiente Transacional do warehouse

A complexidade de desenvolver algoritmos de consistência para warehouse é muito próximo ao escopo das transações nas fontes. Quanto maior o escopo de uma transação mais complexo o algoritmo se torna. A família de algoritmos strobe foi desenvolvida para abranger três tipos de cenários abaixo descritos:

- Single Update Transactions (Única atualização por Transação): É o mais simples dos ambientes onde cada atualização compreende sua própria transação e é notificada para o warehouse separadamente. Sistemas legados que não compreendem transações falham nesta categoria.

- Source-Local Transactions (Transações locais na fonte): é uma seqüência de atualizações executadas na mesma fonte que juntas formam uma única transação. O objetivo é refletir todas essas ações atomicamente no warehouse. É assumido que cada fonte tem um escalonador serial local para todas suas transações locais. Os lotes de atualizações que são passados para o warehouse juntos são considerados ambientes de transacões de fontes locais com transação única.

- Global Transactions (Transações globais): neste cenário existem transações globais que contém ações que são executadas em múltiplas fontes. É assumido que existe uma ordem de serialização global para as transações globais. O objetivo é refletir as transação global atomicamente no warehouse. Dependendo da quantidade de informação que se tem sobre a transação o objetivo pode ser alcançável.

3.6.2 Mensagens e Eventos

Existem dois tipos de mensagens da fonte para o warehouse: informando uma atualização e retornando a resposta a uma consulta. Existe um tipo de mensagem do warehouse para as fontes: o warehouse pode enviar consultas as fontes. Cada transação simples e cada transação local de fonte é informada ao warehouse como uma mensagem no ponto em que a transação *commit*[7]. Para transações globais atualizações podem ser entregues de diversas maneiras: o site que confirma a transação pode recolher todas as atualizações e enviá-las ; cada site pode enviar suas próprias atualizações uma vez que saibam que a transação global tenha sido confirmada.

Cada ação da fonte junto com a mensagem enviada para o warehouse é considerada um evento. Os eventos são atômicos e ordenados pela seqüência da ações correspondentes.

Assume-se que duas mensagens enviadas por uma fonte são entregues na ordem que foram enviadas. Porem não há restrição para a ordem de entrega de mensagens de fontes diferentes, ou seja as fontes não são coordenadas.

O segundo ponto assumido (ordenação ponto-a-ponto) ,pode não ser sempre assegurado, isto é duas mensagens enviadas por rotas diferentes; contudo, o aspecto crucial é que as mensagens serão executas na ordem que foram criadas, não necessariamente na ordem que foram entregues. Para garantir este aspecto são adicionados números seqüências para todas as mensagens de cada fonte. Desta forma o warehouse pode detectar a falta de uma mensagem e esperá-la antes de processar as outras mensagens recentes que já chegaram ao warehouse.

3.6.3 Estados de Fontes e de Warehouse

Cada estado do warehouse *ws* representa o conteúdo do warehouse em um dado momento. O estado do warehouse muda sempre que a visão é atualizada. Sejam *ws1....,wsf* estados do warehouse, assume-se que existe um estado final do warehouse *wsf* quando todas as atividades cessam.

Sejam u o número de fontes identificadas unicamente por um identificador x $(1 \leq x \leq u)$.O estado da fontes *ss* é um vetor que contém u elementos e representa o estado de cada

[7]termo amplamente utilizado para determinar quando uma transação é finalizada com sucesso

fonte em um dado momento. O estado de uma fonte x é dado por ss[x]. Assume-se que as atualizações da fonte são executadas de forma serializável através de todas as fontes, ou seja existe um escalonador serial "S" que representa a execução das atualizações. O estado final das fontes depois que S termina é denotado por ss_q. V(ss) é o resultado de computar sobre a visão V o estado de fonte ss.

Cada transação de fonte leva a fonte de um estado consistente para outro. Para algum escalonador serial R, usa-se *result(R)* para referir-se ao vetor de estado das fontes que resulta da sua execução.

3.6.4 Níveis de Consistência

Assume-se que as visão do warehouse inicialmente é sincronizada com o estado da fonte, $V(ss_0)= V(ws_0)$. Cada nível de consistência assume a validade no nível anterior. Os níveis de consistência definem o que significa o algoritmo ser consistente em um ambiente onde as atividades das fontes é desacoplada das visões do warehouse:

- Convergência: para todas as finitas execuções $V[ws_f]= V\ [ss_q]$. Que significa que após a última atualização e após todas atividades terem cessado, a visão é consistente com os dados da fonte.

- Consistência Fraca: Para todas execuções e para todo ws_i, existe um vetor de estados de fontes ss_j tal que $V[ws_i]= V[ss_j]$. Além disso para cada fonte x existe um escalonador serial de transações R= T1,.....,Tk tal que result(R)[x]= ss_j[x].O que significa que cada estado do warehouse reflete um estado válido em cada fonte, e existe um escalonamento serializável local em cada fonte que garante este estado. Porém cada fonte pode refletir um escalonamento seriável diferente e o warehouse pode refletir um conjunto diferente de transações confirmadas em cada fonte.

- Consistência Forte: Existe um escalonador serial R e um mapeamento m, dos estados do warehouse nos estados das fontes tal que: (i) O escalonador serial R é equivalente a execução atual das transações nas fontes.(ii) Para todo ws_i, $m(ws_i)= ss_j$ para algum j e $V[ws_i]= V[ss_j]$. (iii) Se $ws_i < ws_k$ então $m(ws_i) < m(ws_k)$. O que significa que cada estado do warehouse reflete um conjunto válido de estados as fontes, refletindo o mesmo escalonamento serial global e a ordem dos estados do warehouse é a mesma das ações das fontes.

- Completude (Completeness):Para todo ss_j definido por R existe um ws_i tal que $m(ws_i)= ss_j$, ou seja existe um completa preservação de ordem mapeada entre os estados da visão e os estados da fonte. Apesar desta propriedade ser importante acredita-se que é muito forte e na prática é desnecessária nos ambientes de warehouse.

3.6.5 Definições

Os algoritmos da família Strobe receberam este nome por causa das luzes estroboscópicas, pois eles periodicamente "congelam"as constantes mudanças das fontes em um visão consistente no warehouse.

- Uma visão V no warehouse sobre n relações é definido por uma expressão Project-Select-Join (PSJ) da forma V=$\Pi proj(\sigma cond($r1$\bowtie$ r2 $\bowtie$ .. rn $))$.Duas relações podem estar em uma mesma fonte ou em fontes diferentes. As listas de projeções contém os atributos chaves para cada uma das relações.

- A visão materializada MV de uma visão V é o estado corrente de V no warehouse V(ws). Quando uma visão é definida sobre múltiplas fontes, uma atualização em uma fonte pode gerar uma consulta a multiplas fontes. Como não se pode assumir que as fontes vão cooperar para responder essa consulta, o warehouse deve decidir para onde enviar primeiro a consulta.

- Dado uma consulta Q que necessita de resposta, a função *next-source(Q)* retorna o par *(x,Q^i)* onde x é á próxima fonte a ser contatada e Q^i é a porção da consulta Q que será executada em x. Quando Q estiver completa o valor de$x$ é nil. A^i será a resposta recebida no warehouse em resposta a sub-consulta Q. Para consultas do tipo *project-select-join* o procedimento *next-source* sempre irá escolher a relação com a qual pode ser efetuado a junção com a parte conhecida da consulta.

- Dado uma consulta Q o procedimento *source-evaluate* será o procedimento responsável por computar toda a consulta, ou seja ele determina a próxima porção da consulta (sub-consulta) e a computa até que a resposta final da consulta seja calculada. O procedimento *source-evaluate* pode retornar uma resposta incorreta quando existem transações concorrentes nas fontes que interferem na evolução da consulta. A idéia chave do algoritmo Strobe é armazenar as atualizações que ocorrem durante a evolução da consulta e mais tarde compensá-las.

Source−Evaluate(Q): retorna resposta a uma consulta$Q$

inicio
  i=0; WQ=Q; A^0=Q
  "WQ é a consulta corrente (a sub consulta)"

(x,Q^1)$\longleftarrow$ *next_source*(WQ);
Enquanto x não é nil
Seja i= i+1;
Envie Q^ipara a fonte x;
Quando x retornar A^i faça WQ= WQ$< A^i >$
Seja (x, Q^{i+1}) $\longleftarrow$*next_source(WQ)*;
Retorne (A^i);
Fim.

- O conjunto de consultas não respondidas (UQS) é o conjunto de todas as consultas que o warehouse enviou para alguma fonte mas que ainda não recebeu resposta.

- A operação *key-delete(R,U_i)* deleta da relação R todas as tuplas cujo atributos chave tenham os mesmo valores que U_i.

- $V< U >$ denota a expressão da visão V com a tupla U substituída pelas relações de U.

3.6.6 Descrição Algoritmo Strobe

O algoritmo processa atualizações na ordem em que elas chegam, enviando consultas para as fontes quando for necessário. Essas atualizações não são executadas imediatamente na visão materializada*(MV)*. Ao invés disso é gerada uma lista de ações AL para ser executada na visão. A MV só é atualizada quando se tem certeza que aplicar todas as ações de (AL), como uma única transação no warehouse, irá levar o warehouse para um estado consistente. Este processo ocorre quando não existe consultas pendentes de respostas e todas as atualizações recebidas foram processadas. Por motivo de simplificação este algoritmo tratará apenas de inserções e deleções sendo que as alterações podem ser tratadas de *deletes* seguidos de *inserts*. Quando o warehouse recebe um *delete* ele gera uma ação de deleção para as tuplas correspondentes (que tem o mesmo valor de chave) na MV. Quando um *insert* chega o warehouse precisa gerar e processar uma consulta, usando o procedimento *source-evaluate( )*. Enquanto a consulta Q esta sendo respondida pelas fontes, atualizações podem chegar ao warehouse e a resposta da consulta obtida pode estar incorreta. Para fazer então uma compensação, existe um conjunto de pendências *pending(Q)* que contém as atualizações que ocorrem enquanto a consulta Q é processada. Quando a resposta de Q chega ao warehouse ela é totalmente compensada. Feito este processo uma ação de inserção é gerada paraMV e é colocada na lista de ações AL.

O algoritmo Strobe garante consistência forte para ambiente Single-update-Transaction (Cada atualização em uma transação).

Algoritmo Strobe

Em cada fonte:

Após executar a atualização U_i envie U_i para o warehouse

Se receber a consulta Q_i compute a respostaA^i sobre*ss[x]* (estado corrente de x)

Envie A^i para o warehouse.

No warehouse:

Inicializa AL como conjunto vazio <>

Quando receber uma atualização U_i

Se U_i for um *delete*

$\forall$ $Q_j \in$ UQS adicione U_i para *pending(*Q_j*)*;

Insira *key-delete(MV,* U_i*)* em AL;

Se U_i for um *insert*

Faça $Q_i = V < U_i >$ e faça *pending(*Q_i*)*$= \phi$;

Faça $A^i =$ *source-evaluate(*Q_i*)*;

$\forall$ $U_j \in$ *pending(*Q_i*)*, aplique *key-delete*(A_i,U_j);

Adicione *insert*(MV,A_i) a AL;

Quando $UQS = \phi$, aplique AL a MV como uma única transação sem adicionar tuplas repetidas

Reset $AL = \phi$;

Fim

Exemplo Sejam as relações nas fontes:

	A	B		B	C		C	D
r1	1	2	r2	-	-	r3	3	4

Seja a visão da fonte definida por V= (r1 ⋈ r2 ⋈ r3). Onde r1, r2, r3 são três relações residentes nas fontes x, y, z respectivamente. Inicialmente a visão materializada(MV)=ϕ.

1. AL= <> O warehouse recebe o evento U_1= *insert(r2,[2,3])* da fonte y. O warehouse gera a consulta Q_1= *(r1 ⋈ [2,3] ⋈ r3)*. Para responder Q_1 o warehouse envia primeiro Q_1^1= *(r1 ⋈ [2,3])* para fonte$x$.

2. O warehouse recebe A_1^1= ([1,2,3]) da fonte x então a consulta Q_1^2= ([1,2,3] ⋈ r3) para a fonte z.

3. O warehouse recebe U_2= *delete(r1,[1,2])* da fonte x. Ele primeiro adiciona U_2 para a fila de pendência, *pending*(Q_1) e adiciona *key-delete*(MV,U_2) emAL. O AL resultante é$AL=<$ *key-delete*(MV,U_2)$>$.

4. O warehouse recebe A_1^2= [1,2,3,4] da fonte$z$. Como a *pending*(Q_1) o warehouse aplica *key-delete*(A_1^2, U_1) e a resposta resultante é A_2= ϕ. Então nada é adicionado a AL. Como não há consultas pendentes então o warehouse atualiza a visão materializada MV aplicando AL= <*key_delete*(MV,U_2)>. A MV resultante é igual a vazio, MV= ϕ. A visão final é correta e fortemente consistente com a relações da fonte.

3.7 Algoritmo Strobe Transacional (Transaction-Strobe Algorithm)

O algoritmo Strobe-Transacional (T-Strobe) adapta o algoritmo Strobe para garantir consistência forte para ambientes com transações locais nas fontes. O algoritmo T-Strobe coleta todas as atualizações executadas por uma transação e processa essas atualizações como uma unidade atômica. Colocar em lote as atualizações de uma transação reduz o número de mensagens de consultas que devem ser enviadas para e das fontes, além de tornar mais fácil garantir a consistência.

- *UL(T)* é a lista de atualizações de uma transação T. *UL(T)* contém os *inserts* e os *deletes* executados por T em ordem. *IL(T)* $\subseteq$ *UL(T)* é a lista de inserções de T, ou seja contém todas as inserções de T.

- *key*(U_i) denota os atributos chaves de uma tupla a ser inserida ou a ser deletada. Se *key*(U_i)= *key*(U_j) então U_i e U_j denotam a mesma tupla.

3.7.1 Descrição do algoritmo T-Strobe

Primeiro o warehouse remove os pares de *insert* e *delete* sobre uma mesma tupla que tenha sido inserida e depois removida. Este passo é uma otimização que permite "economizar "uma consulta. Depois o warehouse adiciona todas os *deletes* restantes para a lista de ações AL. Tratados os *deletes* o warehouse gera uma consulta com todos os *inserts*. Como no algoritmo Strobe os *deletes* que chegarem após a consulta ser gerada são subtraídos do resultado da consulta.

3.7.2 Algoritmo Strobe-Transacional (T-Strobe)

Em cada fonte:

Após executar a atualização U_i envie U_i para o warehouse
Se receber a consulta Q_i compute a resposta A^i sobre*ss[x]*
(estado corrente de x) e envie A_i para o warehouse.

No warehouse:

InicializaAL=<>
Desde receba de *UL(*T_i*)* para uma transação T_i
Para cadaU_j,U_k $\in$ *UL(*T_i*)* tal que U_j é um *insert* e U_k é um *delete*, $U_j < U_k$ e *key(*U_j*)*= *key(*U_k*)* remova U_j e U_k de *UL*(T_i)

Para cada *delete* $U \in$ *UL(*T_i*)*:
$\forall$ $Q_j \in UQS$ adicione U para *pending(*Q_j*)*;
Insira *key-delete*(MV,U) em AL;
Faça $Q_i = \cup_{U_j \in IL(T)}$ $V< U_j >$ e faça *pending(*Q_i*)*= ϕ ;
Faça A_i= *source-evaluate(*Q_i*)*;
$\forall U \in$ *pending(*Q_i*)*, aplique *key-delete(*A_i, U);
Adicione *insert*(MV,A_i) a AL;
QuandoUQS= ϕ, apliqueAL a MV sem adicionar tuplas repetidas a MV
Reset AL= ϕ;
Fim

3.8 Algoritmo Strobe-Global(Global Strobe)

O algoritmo Strobe-Global (G-Strobe) adapta o algoritmo T-Strobe para garantir consistência forte para ambientes com transações globais nas fontes. Seja TT o conjunto de identificadores de transações que o warehouse tenha recebido desde a ultima atualização de MV. O algoritmo G-Strobe é o mesmo do T-Strobe com exceção que ele apenas atualiza MV com as ações de AL quando três condições tenham sido satisfeitas.

1. UQS= ϕ ;

2. Para cada transação T_i em TT que depende de (no senso do controle de concorrência) outra transação T_j, T_j esta sempre em TT;

3. Todas as atualizações de transações em TT tenham sido recebidas e processadas;

Garantir a condição 3 é fácil se as fontes cooperarem, mesmo que não exista controle de concorrência global. Se todas as atualizações de uma transação global são enviadas em uma única mensagem pelo site que *commit* a transação, o warehouse vai sempre possuir toda a transação. Se as atualizações são enviadas em mensagens separadas, então os identificadores de transação são necessários a cada mensagem mais um contador com a quantidade de atualizações envolvidas na transação. Juntos o contador e o os identificadores tornam possível ao warehouse coletar todas as atualizações antes de processá-las.

Garantir a condição 2 na prática é mais complexo. Fontes geralmente não informam as transações do qual uma transação a ser confirmada depende. Esta condição tem de ser garantida indiretamente. Não é possível receber uma transação sem receber primeiro pelo menos uma das atualizações de cada transação de qual ela depende, levando em consideração o senso de dependências das transações. Se as atualizações são enviadas separadas por cada fonte deve-se adicionar um número seqüencial a cada atualização para fazer com que o warehouse espere por todas as atualizações anteriores geradas por uma fonte antes de executar uma atualização. Apesar desta condição ser conservativa ela não garante a segunda condição. O mecanismo para enviar transações para o warehouse irá determinar se o algoritmo G-Strobe pode garantir razoavelmente a consistência forte nas visões do warehouse. Se o G-Strobe não é viável então pode-se reverter para os algoritmos T-Strobe e Strobe e garantir uma nível mais fraco de consistência. O algoritmo Strobe garante convergência para transações globais.O algoritmo T-Strobe garante consistência fraca para transações globais.

3.9 Complete Strobe (C-Strobe)

Um problema com os algoritmos da família Strobe é que se existem atualizações constantes, os algoritmos podem não encontrar uma estado onde UQS (conjunto consultas pendentes) é vazio e a visão materializada pode ser atualizada. Para resolver esta questão existe um quarto algoritmo da família Strobe chamado de Strobe Completo (C-Strobe) que pode atualizar o warehouse após alguma atualização da fonte ou um lote de atualizações. Por motivo de simplificação será descrito a seguir o algoritmo para forçar a atualização da MV após cada atualização.

Para forçar uma atualização da MV após uma atualização Ui chegar ao warehouse, é necessário computar a visão resultante. Porém outras atualizações concorrentes nas fontes podem complicar o problema. Por exemplo se a atualização Ui for um *insert* para computar a próxima MV o warehouse tem que enviar uma consulta Qi para as fontes. Enquanto a resposta Ai da consulta não chega outras atualizações podem chegar ao warehouse e deverão ser compensadas para manter a resposta Ai correta. Se estas atualizações forem *inserts* deve-se apenas remover as tuplas correspondentes de Ai. Se a atualização for um *delete* o warehouse pode necessitar adicionar tuplas a respostaAi, mas para computar essas tuplas uma nova consulta será gerada para as fontes. Quando a resposta para as consultas intermediárias chegarem ao warehouse então a visão pode ser materializada.

3.9.1 Definições

- O conjunto de consultas enviadas pelo warehouse para computar a visão após a atualização de inserção U_i e denotado por $Q_{i,\_,\_}$, $Q_{i,j,\_}$ são as consultas enviadas em respostas a atualização U_j que ocorre enquanto se computa a resposta para uma consulta em $Q_{i,\_,\_}$. Um único inteiro k é usado para distinguir qual consulta $Q_{i,j}$, como $Q_{i,j,k}$. Quando se processa cada atualização U_i separadamente, uma lista de ações não é necessária. No algoritmo C-Strobe cada atualização é compensada por uma subseqüente atualizações então ele pode ser aplicado diretamente na visão.

- $Q < U_i >$ é a consulta resultante após a tupla atualizada em U_i sobrepõem sua relação base em Q. Se a relação base de U_i não aparece em Q então $Q< U_i >= \phi$.

- *Delta* é o conjunto de mudanças que precisa ser aplicada a MV por uma atualização de inserção. Quando *Delta* for computado corresponderá a um simples ação de *insert(MV,Delta)* na$A$L se mantivermos uma lista de ação. As deleções podem ser aplicadas diretamente aMV.

- Define-se $\Pi proj(U_i)$ para ser a tupla que casa em todos os valores com a tupla de atualização U_i, mas apenas os atributos de U_i que aparecem na lista de projeções *proj*.

- Usa-se uma versão diferente de *key-delete*: *key-delete*(Delta,*U_k*)* onde esta apenas deleta de *Delta* aquelas tuplas que casam com U_k nos atributos chaves e não-chaves.

Quando se adiciona tuplas a *Delta* esta se assumindo que irão ser adicionadas tuplas com chaves iguais e os demais valores diferentes. Estas tuplas violam a condição de chaves, porém são apenas temporárias em *Delta* e são necessárias para o correto funcionamento do algoritmo C-Strobe.

- C-Strobe é completo pois a MV é atualizada uma vez após cada atualização e o estado resultante do warehouse corresponde ao estado da fontes após a mesma atualização. O processo de compensação sempre termina porque uma expressão em $Q_{i,j,k} < Up >$ sempre tem uma relação base a menos que $Q_{i,j,k}$.

3.10 Algoritmo C-Strobe

Em cada fonte:

Após executar a atualização U_i envie U_i para o warehouse.
Se receber a consulta Q_i compute a resposta A^i sobre ss[x] (estado corrente de x)
  Envie A^i para o warehouse.

No warehouse:

Inicializa $Delta = \phi$
Com a chegada das atualizações elas são colocadas numa fila de espera
Processa-se cada atualização U_i em ordem de chegada:
  Se U_i é um *delete*
    Aplique *key-delete(MV, U_i)*
  Se U_i é um *insert*
    Faça $Q_{i,i,0} = V < U_i >$
    Faça $A_{i,i,0} =$ *source-evaluate($Q_{i,i,0}$)*;
    Repita para cada $A_{i,j,k}$ até UQS$= \phi$
      Adicione $A_{i,j,k}$ a *Delta* (sem tuplas duplicadas)
      Para todos *deletes* U_p recebidos entre U_j e $A_{i,j,k}$:
        Faça $Q_{i,p,newp} = Q_{i,j,k} < U_p >$
        Faça $A_{i,p,newp} =$*source-evaluate($Q_{i,p,newp}$)*;
        Quando as respostas chegarem voltar ao Repita)
  Para todas as inserções U_k recebidas entre U_i e a ultima resposta,
  Se $\neg\exists U_j < U_k$ tal que U_j é um *delete* e $\Pi proj$ (U_j)$= \Pi proj$ (U_k) então
    Aplique *key-delete*(Delta, U_k)*
  Faça $MV = MV + Delta$;
  $Delta = \phi$;
Fim

4 Conclusão

Está cada vez mais evidente a necessidade de evolução dos algoritmos que proporcionam as atualizações de dados entre os sistemas operacionais e o data warehouse com um tempo menor para sincronismo de dados entre esses dois ambientes. As organizações das empresas e as novas condições de trabalho impulsionam o desenvolvimento destas novas características de sincronismo de dados. O levantamento bibliográfico realizado por este trabalho teve por objetivo analisar e entender algumas famílias de algoritmos utilizadas para tal processo de sincronismo de dados. Esta análise serve como base inicial para uma dissertação de mestrado que tem por motivação o estudo e possível criação de um *framework*(ambiente) onde esses algoritmos possam ser comparados e simulados. Através destas comparações e simulações sera possível caracterizar paramêtros de eficiência dos algoritmos e de acordo com as condições do meio encontrado escolher o melhor algoritmo para a situação. Desta maneira captar os paramêtros de eficiência e criar uma simulação que mostre a necessidade de um ambiente adaptativo seriam tarefas a serem desenvolvidas no mestrado.

Referências

[AV98] Christopher Adamson and Michael Venerable. *Data Warehouse Design Solutions*. John Wiley & Sons, 1998.

[BGMS92] Yuri Breitbart, Hector Garcia-Molina, and Abraham Silberschatz. Overview of multidatabase transaction management. *VLDB Journal: Very Large Data Bases*, 1(2):181–293, 1992.

[BLT86] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In C. Zaniolo, editor, *sigmod*, pages 61–71, Washington, DC, May 1986. acm.

[GM95] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques and Applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, 1995.

[GMS] Ashish Gupta, Inderpal Singh Mumick, and V.S Subrahmanian. Maintaining views incrementally (extended abstract). http://www.researchindex.com/58429.html.

[MSRK98] Mukesh K. Mohania, Sunil Samtani, John Roddick, and Yahiko Kambayashi. Advances and research directions in data warehousing technology, 1998. http://www.researchindex.com/355289.html.

[QGMW96] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making views self-maintainable for data warehousing. In *Fourth International Conference on Parallel and Distributed Information Systems (PDIS '96)*, pages 158–169, Los Alamitos, Ca., USA, December 1996. IEEE Computer Society Press.

[QW97] Dallan Quass and Jennifer Widom. On-line warehouse view maintenance for batch updates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 26,2 of *SIGMOD Record*, pages 393–404, New York, May 13–15 1997. ACM Press.

[SMKK99] Sunil Samtani, Mukesh K. Mohania, Vijay Kumar, and Yahiko Kambayashi. Recent advances and research problems in data warehousing. *Lecture Notes in Computer Science*, 1552:81–92, 1999.

[ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 316–327, San Jose, California, 22–25 May 1995.

[ZGMW96] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. The strobe algorithms for multi-source warehouse consistency. In *Fourth International Conference on Parallel and Distributed Information Systems (PDIS '96)*, pages 146–157, Los Alamitos, Ca., USA, December 1996. IEEE Computer Society Press.

[ZGMW98] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. Consistency Algorithms for Multi-Source Warehouse View Maintenance. *Distributed and Parallel Databases*, 6(1):7–40, 1998.