Um modelo Confluente de Execução de Regras em Bancos de Dados Relacionais Ativos

Emerson dos Santos Paduan

Dissertação submetida em cumprimento parcial
dos requisitos para obtenção do grau de

Mestre em Matemática Aplicada.

Área de Concentração: **Ciência da Computação**

Orientador: **Prof. Dr. Marcelo Finger**

(O autor recebeu apoio financeiro do CNPq durante o primeiro ano da elaboração deste trabalho)

Instituto de Matemática e Estatística da USP

- São Paulo, Abril de 2000 -

Um Modelo Confluente de Execução de Regras em Bancos de Dados Relacionais Ativos

Este exemplar corresponde à versão final
da dissertação apresentada por Emerson
dos Santos Paduan e aprovada pela
comissão julgadora.

São Paulo, Abril de 2000.

Banca Examinadora:

Prof. Dr. Marcelo Finger (orientador)	IME-USP
Prof.a Dra. Dilma Menezes da Silva	IME-USP
Prof.a Dra. Claudia Maria Bauzer de Medeiros	IC-UNICAMP

Aos meus pais.

AGRADECIMENTOS

Ao Prof. Dr. Marcelo Finger, pela orientação deste trabalho, pela paciência com seu orientando e pelo apoio transmitido durante a realização deste trabalho.

À minha família, por todo incentivo, apoio e carinho.

Aos amigos que apoiaram direta ou indiretamente o desenvolvimento deste trabalho.

Ao CNPq, pelo apoio financeiro concedido.

ÍNDICE

INTRODUÇÂO **2**

1.1 INTRODUÇÃO AOS BANCOS DE DADOS ATIVOS 3

1.2.UTILIZAÇÃO DOS BANCOS DE DADOS ATIVOS 10

1.2.1 APLICAÇÕES INTERNAS *11*
1.2.2 APLICAÇÕES ESTENDIDAS *11*
1.2.3 APLICAÇÕES EXTERNAS *12*

1.3.ARQUITETURA DOS SISTEMAS DE BANCOS DE DADOS ATIVOS 12

1.3.1 ARQUITETURA EM CAMADA *12*
1.3.2 ARQUITETURA EMBUTIDA *13*
1.3.3 ARQUITETURA COMPILADA *14*

SEMANTICA DA EXECUCAO DAS REGRAS **17**

2.1 GRANULARIDADE DO PROCESSAMENTO DE REGRAS 18

2.2 EXECUÇÃO ORIENTADA A INSTANCIA X ORIENTADO A CONJUNTO 19

2.3 ALGORITMO RECURSIVO X ITERATIVO 20

2.4 EXECUÇÃO SEQUÊNCIAL X CONCORRENTE 23

2.5 MODOS DE ACOPLAMENTO 24

2.6 RESOLUÇÃO DE CONFLITOS 25

IMPLEMENTACAO DE REGRAS EM BANCOS DE DADOS COMERCIAIS **26**

3.1 VALORES DE TRANSIÇÃO 27

3.2. ORACLE 28

3.3 SYBASE 31

3.4 SQL SERVER 32

3.5 QUADRO COMPARATIVO 34

ANALISE DE REGRAS **36**

4.1.TERMINAÇÃO 37

4.2. CONFLUÊNCIA 39

4.2.1. EXEMPLO DO PROBLEMA DE CONFLUÊNCIA *41*
4.2.2 UM MÉTODO PARA ANALISAR A CONFLUÊNCIA *42*
4.2.2.1 Grafo de Execução de Regras Intencionais *43*
4.2.2.2 Comutatividade entre Regras *43*
4.2.2.3 Método para Análise da Confluência *45*
4.2.2.4 Considerações *48*
4.2.3 CONJUNTO DE AÇÕES CONSISTENTES *48*

SEMANTICA CONFLUENTE PARA REGRAS....51

5.1 ALGORITMO PARA REGRAS CONFLUENTES....54

5.2 EXEMPLO 1....54

5.3 EXEMPLO 2....55

ARQUITETURA PARA IMPLEMENTAÇÃO DA SEMÂNTICA CONFLUENTE....58

6.1 O PROCESSADOR DE DISPAROS....62

6.1.1 PASSOS 1 E 2....63
6.1.2 PASSO 3 E 4....64
6.1.3 CALCULO DE NÚMERO DE TRIGGERS DISPARADAS....64
6.1.4 CALCULO DE NÚMERO DE AÇÕES A SEREM EXECUTADAS....65
6.1.5 DESCRIÇÃO DAS TABELAS E TRIGGERS....66

6.2 CUSTO....68

6.2.1 DESEMPENHO....68
6.2.2 ESPAÇO....69

6.3 EXEMPLO....69

CONCLUSÃO....69

APÊNDICE : CÓDIGO FONTE....71

BIBLIOGRAFIA....76

ÍNDICE DE FIGURAS

Fig. 1.1 - Diferença entre BD passivo e BD ativo 4
Fig. 1.2 – Arquitetura em Camadas para bancos de dados ativos 13
Fig. 1.3 – Arquitetura Embutida para bancos de dados ativos, adaptado de [10] 14
Fig. 4.1 – Ação de Ri disparando Rj 39
Fig. 4.2 – Um ciclo que pode provocar a não-terminação 39
Fig. 4.3 – Regras comutáveis 40
Fig. 4.4 – Exemplo de um grafo de execução de regras 42
Fig. 4.5 – Caminho num grafo de execução de Regras 42
Fig. 4.6 – Regras comutativas 43
Fig. 4.7 – Confluência de regras 45
Fig. 4.8 – Grafo de execução representando parte dos caminhos p_1 *e* p_2 45
Fig. 4.9 – Grafo de execução cujos caminhos p_1 *e* p_2 *convergem para o mesmo estado final* 46
Fig. 5.1 – Relação de regras com transações 52
Fig. 6.1 – Modelo base 61
Fig. 6.2 – Nosso modelo 61
Fig. 6.3 – Diagrama do Processador de Disparos 62

Um Modelo Confluente de Execução de Regras em Bancos de Dados Relacionais Ativos

por Emerson dos Santos Paduan

Resumo

Nos sistemas de bancos de dados ativos difundidos no mercado e na literatura, regras são projetadas sem a preocupação com seu inter-relacionamento e, na maioria das vezes, com a confluência. No primeiro caso, o desenvolvedor de aplicação é, com freqüência, incapaz de ter uma visão global das regras já projetadas e, consequentemente, de saber se algumas propriedades vitais aos bancos de dados ativos são satisfeitas ou não. Já no segundo caso, o sistema de gerenciamento de banco de dados ativo não garante a consistência das informações por ele gerenciadas.

Mesmo quando se procura garantir a confluência utilizando métodos de análise estática das regras, verificamos que a complexidade para tal análise é bastante grande. Neste trabalho propomos um novo modelo de execução das regras em bancos de dados ativos com o objetivo de garantir a propriedade de confluência.

A Confluent Model for Rule Execution in Active Relational Databases

by Emerson dos Santos Paduan

Abstract

In active databases systems avaliable in the market and literature, rules are projected without thinking about their interaction and generally, their confluence. In the first case, the application developer is frequently unable to have a global vision of the rules which were previously designed and consequently, he doesn't know if somevital active database properties are satisfield or not. In the second case, the active database management system does not guarantee the consistency of information managed.

Even when trying to guarantee the confluence through static rule analysis, we verify the complexity of such na analysis is great. In this paper we propose a new model of execution for active databases rules with the objective of guaranteeing confluence.

Capítulo 1

INTRODUÇÃO

Assistimos nas últimas décadas a uma evolução considerável nos sistemas de armazenamento de dados. Trocamos o acesso seqüencial a arquivos pelos sistemas gerenciadores de bancos de dados (SGBD). Esses sistemas trouxeram grandes benefícios em relação aos antigos meios de armazenamento e acesso a dados. Porém, com o avanço da tecnologia e o aumento da complexidade das aplicações, novas necessidades surgiram e começaram a ficar evidentes as limitações que esses sistemas gerenciadores de banco de dados traziam. Duas destas limitações são:

- Implementação de restrições de integridade: os SGBDs tradicionais[1] suportam somente a construção de restrições de integridade simples, como restringir tipos de dados ou valores possíveis em um certo campo de uma tupla. Uma solução para esta limitação seria codificar as restrições desejadas em cada programa de aplicação que usa a base de dados. Fica claro que o correto funcionamento do sistema fica dependente não só do projeto do SGBD em si, mas de cada programa de aplicação. Também quando é necessária a manutenção destas regras, ela deve ser feita não só no SGBD, mas em cada programa de aplicação. Além disso vale a pena lembrar que a comunicação extra que se faz necessária entre o SGBD e os programas de aplicação impõe uma sobrecarga que torna o sistema mais lento.

- "roll back" desnecessário de transações: em um SGBD tradicional as restrições de integridade são testadas imediatamente após cada atualização feita na base de dados ou ao final de uma transação, e a resposta a uma violação de uma destas restrições é um "roll back". Em alguns casos isso poderia ser evitado se tivéssemos a possibilidade de interferir, quando uma restrição é detectada, e aplicar operações de correção.

Estas limitações dificultam muito a construção de certas aplicações que envolvem um grande número de restrições de integridade ou que implementam políticas específicas de uma empresa ou organização.

---

[1] O termo tradicional usado aqui se refere a sistemas gerenciadores de bancos de dados não ativos.

Nos anos 70 começaram as surgir os primeiros projetos de linguagens e sistemas que tinham como objetivo incluir nos SGBDs tradicionais mecanismos para realização de processos automáticos sem a interferência do usuário. Mas estes projetos só foram reconhecidos e começaram a tomar corpo nos anos 80 com o projeto HiPAC. O nome adotado para os SGBDs que eram acrescidos desta capacidade de automaticamente realizar tarefas foi **Sistemas de bancos de dados ativos**[8].

Nas seções seguintes introduziremos os conceitos necessários para o desenvolvimento desta dissertação. Não é objetivo aqui descrever todos os pontos que envolvem um SGBD Ativo, nem descrever sistemas específicos, mas apenas dar material suficiente para a discussão que será apresentada.

1.1 INTRODUÇÃO AOS BANCOS DE DADOS ATIVOS

Em poucas palavras, um sistema gerenciador de banco de dados ativo (SGBDA) é um sistema que estende as características de um SGBD ("tradicional" ou "passivo") com a possibilidade de especificar um comportamento reativo, isto é, o próprio sistema pode realizar alterações automáticas nos dados em resposta a certos eventos que ocorrem, sem a necessidade de intervenção do usuário ou programas de aplicação.

Em um SGBD toda alteração na base de dados, seja uma inserção, uma exclusão, ou uma modificação de um registro, é realizada mediante uma solicitação direta de um usuário ou através de um programa aplicativo que tenha acesso à base de dados. Já um SGBDA, como mencionamos acima, tem a capacidade de alterar os dados da base de dados sem que seja necessária uma solicitação externa como no SGBD.

O diagrama abaixo ilustra a diferença de comportamento descrita acima entre um SGBD e um SGBDA.

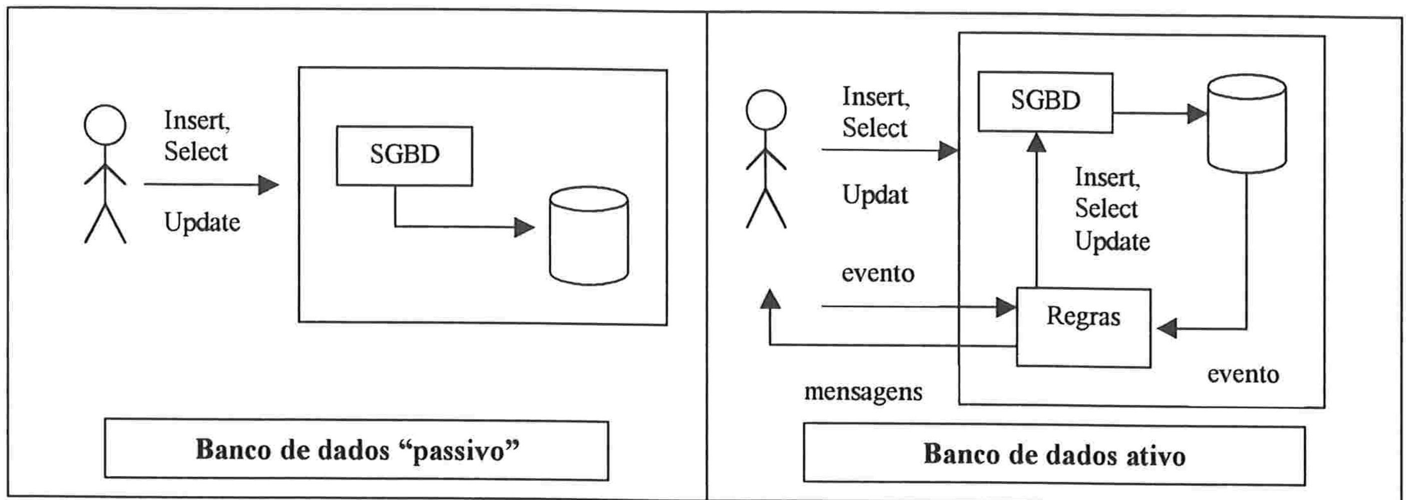


Fig. 1.1 – Comparação entre BD passivo e BD ativo

O que imprime o comportamento reativo nos bancos de dados ativos são as regras. Uma regra em um banco de dados ativo especifica que certas ações devem ser executadas, em resposta a determinados eventos, quando as condições da regra forem satisfeitas[8]; daí o nome regras ECA (Evento-Condição-Ação).

Assim, uma regra é composta por três partes básicas:

- evento: define o que provocará o disparo da regra.
- condição: expressa uma consulta realizada na base de dados. Deve ser verdadeira para que a regra seja executada.
- ação: é a tarefa que a regra deve executar.

Considere uma aplicação que contem as duas tabelas seguintes:

funcionários (cod_func, nome, cod_cargo, salario)

cargos (cod_cargo, sal_max)

A tabela funcionários mantêm os dados dos funcionários de uma empresa e a tabela cargos mantêm informações sobre o salário máximo permitido para cada cargo dentro da empresa. Toda vez que um novo funcionário é cadastrado ou seu salário é reajustado, é necessário realizar uma verificação na tabela cargos para saber se o novo salário não ultrapassa o valor máximo permitido

para seu cargo. Esse trabalho pode ser realizado automaticamente em um SGBDA criando-se uma regra que será ativada quando ocorrer uma inserção ou modificação na tabela funcionário e a ação da regra seria corrigir o salário para o valor máximo do cargo do funcionário caso esse salário tenha ultrapassado o valor máximo.

Informalmente poderíamos escrever a seguinte regra:

QUANDO um novo registro é inserido na tabela funcionários, SE o salário do funcionário for maior que o permitido para o seu cargo na empresa, REDUZA o valor do salário para o máximo permitido para o seu cargo.

Observe que destacamos três palavras na sentença acima para indicar as partes da regra. Analisando a regra acima, a parte referente ao evento que dispara a regra é a inserção de um novo registro na tabela funcionários. A condição da regra é verificar se o salário é maior que o permitido para o seu cargo e a ação a ser executada pela regra é alterar o valor do salário para o máximo permitido para o seu cargo.

Para mostrar algumas possibilidades de cada uma das partes de uma regra, vamos dar alguns exemplos.

Primeiro veremos alguns exemplos de eventos capazes de disparar regras. Cada vez que um destes eventos ocorre no banco de dados, as regras que especificam este evento na sua declaração de eventos, são disparadas.

- modificações feitas nos dados através de uma operação Insert, Update ou Delete. Por exemplo, o comando *Insert Into funcionarios values (1, "João", 30, 1000)* poderia disparar uma regra criada para analisar alguma restrição de integridade quando um novo registro é inserido na tabela de funcionários.
- consultas à base de dados através de um *Select*. Por exemplo, o comando *Select * from funcionarios* poderia disparar uma regra criada para limitar o acesso aos dados da tabela de funcionários de acordo com o direito de acesso do usuário.
- um evento temporal. Por exemplo quando o relógio marcar 10:00 uma regra poderia ser disparada para realizar o backup do sistema.

- eventos definidos pela aplicação. Por exemplo, quando ocorre um login, uma regra poderia ser disparada para ajustar as visões permitidas para o usuário.

Os eventos podem ser compostos utilizando-se conectivos lógicos (*and, or, not*). Desta forma poderíamos ter, por exemplo, uma regra que é disparada quando é feito um login fora do horário de expediente de uma empresa.

Vamos agora ver alguns exemplos de condições que podem fazer parte de uma regra e, portanto, devem ser verificadas para determinar se a ação desta regra deve ou não ser executada.

- verificação de restrições a valores permitidos em um atributo do banco de dados. Por exemplo, quando é feita uma inserção de um registro na tabela de funcionários, verifica se um campo 'idade' está recebendo dados numéricos.
- análise de um valor retornado por uma consulta ao banco de dados. Por exemplo, para que uma regra seja executada pode ser necessário que um determinado número de registros satisfaça uma certa condição. Uma consulta poderia verificar então o número de registros que atendem esta condição.
- Calculo de um valor por um procedimento. Uma regra poderia ser disparada quando o valor de um campo salário na tabela funcionários tivesse sido atualizado aumentando mais que uma certa porcentagem permitida. Assim um procedimento poderia ser chamado na condição de um regra para calcular o valor do aumento quando um registro da tabela funcionários fosse atualizado.

Alguns exemplos de possíveis ações que podem ser realizadas por uma regra são:

- operações de modificações nos dados. Por exemplo, uma regra pode alterar os valores de registros que estão sendo incluídos para que estes satisfaçam as restrições de integridade da tabela no qual o registro está sendo incluído.
- operações de recuperação de dados. Nesse caso, poderíamos ter regras que especificam operações SQL select no banco de dados de acordo com o usuário que está operando o sistema.

- chamada de procedimentos de aplicação. Podemos ter, por exemplo, um procedimento que é chamado por uma regra e realiza cálculos de ajuste nos valores do campo 'salário' dos registros de todos os funcionários que trabalham em um certo cargo.

A seguir descrevemos, de maneira geral, a sintaxe adotada nas implementações de bancos de dados para uma regra.

CREATE TRIGGER < *nome da regra* >

ON < *nome da tabela* >

FOR < *eventos* >

WHEN < *condições* >

AS < *ações* >

No parágrafo anterior dissemos "de maneira geral" porque essa sintaxe pode variar bastante de um SGBDA para outro, de acordo com a implementação (discutiremos melhor esses detalhes mais adiante).Vamos então analisar cada uma das cláusulas acima.

Na cláusula CREATE TRIGGER colocamos o nome da regra (em bancos de dados comerciais as regras são também chamadas triggers[2]. Neste trabalho utilizaremos os dois termos indistintamente para nos referir a regras).

Em um banco de dados relacional, uma regra está associada a uma tabela (em um banco de dados orientado a objetos uma regra está associada a uma classe ou a um objeto), e isso é especificado na clausula ON. Isso quer dizer que o evento que irá provocar o disparo da regra acontecerá sobre essa tabela ou objeto. Por exemplo, quando uma inserção feita na tabela de funcionários deve disparar uma regra, o nome desta tabela deve ser especificado aqui.

O(s) evento(s) que ativa(m) a regra e age(m) sobre a tabela ou objeto descrito na clausula ON são descritos na clausula FOR. No exemplo acima, este evento seria o Insert.

---

[2] Entraremos em maiores detalhes sobre triggers quando tratarmos sobre implementações de regras em bancos de dados comerciais.

As condições que devem ser testadas para verificar se a ação da regra deve ou não ser executada são descritas na clausula WHEN.

E, finalmente, as ações a serem executadas pela regra são descritas na clausula AS.

No inicio desta seção apresentamos informalmente um exemplo da aplicação de regras. Aplicando a sintaxe da regra como acabamos de descrever para esse exemplo teríamos[3]:

CREATE TRIGGER sal_max

ON funcionaios

FOR Insert, Update

WHEN / novo funcionário ultrapassa salário máximo para o cargo */*

AS / altere o salário para o máximo permitido */*

Considere que a tabela cargos possui entre outros o seguinte registro:

5	500,00

Considere ainda que queremos fazer a inserção do seguinte registro na tabela funcionários:

12	Roberto	5	600,00

Este último registro quando inserido provoca o disparo da regra sal_max que altera o valor do salário no registro e o registro inserido será:

12	Roberto	5	500,00

Em uma aplicação convencional as rotinas para suportar esta política de manipulação de dados deveriam ser codificadas em cada programa de aplicação que pudesse inserir ou modificar dados na tabela funcionários. Em um SGBDA toda a tarefa fica centralizada no banco de dados utilizando regras. A conseqüência direta é que a manipulação do código das aplicações se torna

manutenção de regras. Uma vantagem significativa desta utilização de regras ao invés da programação tradicional, é que as regras impõem um comportamento único e consistente no banco de dados, independente da aplicação ou evento que provocou o disparo da regra. Em outras palavras, as regras estabelecem políticas de gerenciamento que nenhuma aplicação pode violar e o risco de existir uma aplicação que manipule os dados de maneira incorreta e gere inconsistências na base de dados se torna muito pequeno.

Uma vez introduzida a sintaxe e o objetivo de uma regra, vamos agora entender um pouco sobre o modelo de execução das regras, ou seja, a forma como as regras são executadas dentro do SGBDA.

A execução de regras ECA se faz em 3 fases: detecção do evento, teste da condição e execução da ação. Um SGBDA monitora a ocorrência de eventos pré-especificados por regras ECA. Uma vez que o evento tenha ocorrido, a parte da condição é testada. Se o teste é satisfeito, a parte da ação é executada. Quando um evento que uma regra está esperando ocorre, dizemos que a regra foi disparada. Uma regra é eleita para execução quando sua condição é avaliada como verdadeira ou a regra não possui a parte da condição, o que também é possível.

Embora esse modelo possa parecer uniforme para todas as implementações de modelos de execução de regras, isso não é uma realidade. Os modelos existentes são bastante dependentes da aplicação que os implementam. Em algumas aplicações, com o objetivo de reduzir o tempo de resposta aos eventos, é importante avaliar as condições imediatamente após um evento ter ocorrido e executar a ação logo a seguir. Neste modelo de execução, o processamento do restante da transação que provocou o disparo da regra é suspendido até que a regra disparada seja completamente processado. A execução completa da regra pode acarretar tempos de espera longos, especialmente se a ação da regra causa o disparo de outras regras. Tempos de resposta e concorrência podem ser melhorados se a avaliação da condição e/ou a execução da ação são separadas da transação que disparou a regra e executadas em uma transação separada [6]. Dessa maneira, diferentes implementações surgem para atingir objetivos e propósitos de aplicações diferentes.

---

[3] Utilizamos comentários para ocultar os cálculos que não são relevantes ao propósito do exemplo que é exibir a sintaxe geral das regras.

O modelo de execução de regras ECA (de maneira geral) tem que considerar vários outros aspectos também. Primeiro, várias regras podem ser disparadas e eleitas para execução ao mesmo tempo. Por exemplo, suponha que duas regras r_i e r_j são disparadas quando o evento E_1 ocorre. Se o evento ocorre, isso irá disparar r_i e r_j simultaneamente. Nesse caso, a ordem de execução de r_i e r_j poderia ser determinada pelo sistema e uma ordem de execução diferente poderia levar o banco de dados a estados diferentes (não-determinístico), o que é inadmissível. Por exemplo, suponha que a condição da regra r_i é avaliada como falsa. Então sua parte da ação não deveria ser executada. Mas a execução da ação da regra r_j poderia alterar o estado do banco de dados de tal maneira que a condição da regra r_i passe a ser verdadeira. Portanto, se a ação de r_j é executada antes da condição de r_i ser avaliada, a ação de r_i será executada também, mas se a condição de r_i for avaliada antes, então a ação de r_i não será executada. Assim, temos a possibilidade de chegar a dois estados finais diferentes no banco de dados. Esse é o problema que estamos interessados em tratar neste trabalho, e que vamos, portanto, detalhar mais tarde.

Além disso, embora os bancos de dados ativos tenham muitas vantagens sobre os bancos de dados passivos, existem alguns problemas que têm impedido que eles sejam largamente utilizados, dentre os quais os principais são os seguintes:

- falta de uma metodologia de projeto de bancos de dados ativos que auxilie na determinação de qual parte do comportamento da aplicação será de responsabilidade das regras dos bancos de dados ativos e como estas regras devem ser especificadas para fornecer, eficientemente, o comportamento desejado [8].
- Dificuldade para predizer o comportamento de um conjunto de regras; isto se deve principalmente à ausência do conceito de modularização ou estruturação na maioria das linguagens de regras existentes atualmente [8].

1.2. UTILIZAÇÃO DOS BANCOS DE DADOS ATIVOS

Os sistemas de bancos de dados ativos são utilizados na implementação de várias aplicações. Segundo [8], tais aplicações podem ser divididas em três categorias:

- aplicações internas;
- aplicações estendidas; e
- aplicações externas.

Vejamos cada uma delas.

1.2.1 Aplicações Internas

Estas aplicações implementam algumas funcionalidades suportadas pelos sistemas de bancos de dados passivos utilizando regras. Os principais exemplos desta categoria são:

- gerenciamento de restrições de integridade: sabemos que as restrições de integridade fornecem meios para assegurar que mudanças feitas nos banco de dados por usuários autorizados não resultem na perda de dados [10]; portanto, as regras que as implementam simplesmente verificam se tais restrições foram ou não violadas e, em caso afirmativo, executam ações corretivas que tratam a violação em questão; e
- computação de visões: as visões são relações lógicas baseadas em uma ou mais relações e/ou outras visões que podem ser gerenciadas de duas formas. A primeira corresponde às visões virtuais que não são armazenadas, mas todas as vezes que uma referência a uma destas visões é feita, seus dados são extraídos do banco de dados por uma consulta. Também utilizamos as visões materializadas, cujos dados devem manter-se atualizados com suas relações base; para tanto, qualquer atualização nestas tabelas devem ser propagadas para as visões que a utilizam. Ambos os casos podem ser facilmente implementados utilizando regras.

1.2.2 Aplicações Estendidas

Estas aplicações utilizam as regras na implementação de funções que suportam tarefas de bancos de dados não padronizadas. Exemplos típicos são os sistemas que trabalham com grande quantidade de dados e cujas atividades possam ser divididas e processadas em servidores diferentes; neste caso, as regras podem coordenar os fluxos de dados e controle entre estes computadores. Outros exemplos desta categoria são os gerenciadores de versões e replicação.

1.2.3 Aplicações Externas

Nesta categoria, o comportamento reativo das regras é usado na implementação de sistemas pertencentes a um domínio específico. Alguns exemplos são controle de trafego aéreo, controle de empréstimos de uma biblioteca, e outros.

1.3.ARQUITETURA DOS SISTEMAS DE BANCOS DE DADOS ATIVOS

Nesta seção, apresentaremos as três possíveis arquitetura de bancos de dados ativos: em camada, embutida e compilada.

1.3.1 Arquitetura em Camada

Quando os componentes do banco de dados ativo estão localizados em uma camada acima do sistema de banco de dados passivo, conforme a figura 1.2, dizemos que a arquitetura deste banco de dados é **em camada** [8].

Nesta arquitetura, o gerenciamento e processamento de regras é separado do SGBD. Portanto, para monitorarmos os eventos que disparam as regras, devemos interceptar os comandos submetidos ao SGBD ou os dados que são retornados ao usuário ou aplicação que os solicitou. Uma vez que um evento que dispara uma regra ocorre, o processador de regras avalia as condições da regra e, se as condições são verdadeiras, envia as ações para serem executadas ao SGBD ou faz chamadas a procedimentos de aplicação [8].

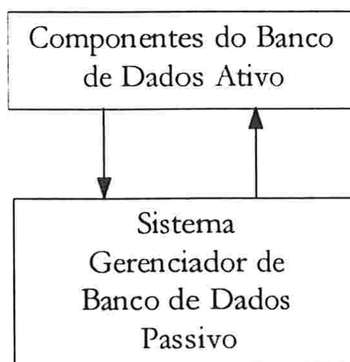


Fig. 1.2 - Arquitetura em Camadas bancos de dados ativos.

Segundo [8], as vantagens de usarmos esta arquitetura são:

- os sistemas e bancos de dados passivos podem ser facilmente convertidos em sistemas ativos sem a modificação do núcleo do sistema de banco de dados;
- sistemas de bancos de dados passivos diferentes podem ser convertidos em sistemas de bancos de dados ativos que fornecem a mesma "interface".

As desvantagens do uso desta arquitetura são:

- baixo desempenho devido à intensa comunicação entre a camada ativa e o banco de dados passivo; e
- como a camada ativa não pode interagir com subsistemas do banco de dados passivo, algumas características que requerem acesso a estes subsistemas não podem ser implementadas. Por exemplo, a camada ativa não pode interagir diretamente com o controle de concorrência do sistema e gerenciar diretamente uma transação que esteja ocorrendo.

1.3.2 Arquitetura Embutida

Na **arquitetura embutida**, todos os componentes do banco de dados ativo fazem parte do sistema de banco de dados, isto é, todo o gerenciamento e processamento de regras está integrado ao SGBD [8]. O gerenciador de dados de baixo nível monitora todas as operações que acontecem sobre os dados e avisa o componente ativo quando um evento de interesse ocorre. Uma vez que uma regra é disparada, o processador de regras avalia as condições da regra e, caso estas sejam verdadeiras, executa as ações da regra chamando procedimentos ou realizando operações diretamente sobre a base de dados. A figura 1.3 mostra um sistema de banco de dados ativo com esta arquitetura.

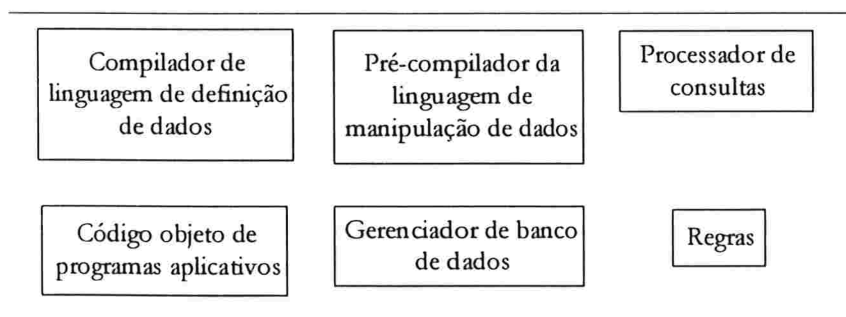


Fig. 1.3 - Arquitetura Embutida para banco de dados ativos, adaptado de [10].

As vantagens e desvantagens desta arquitetura são o oposto da arquitetura em camada. As vantagens são:

- bom desempenho no processamento de regras, visto que este é feito dentro do SGBD;
- acesso aos subsistemas do banco de dados que permitem a implementação de características mais avançadas.

As desvantagens são:

- complexidade para implementação e manutenção do sistema de banco de dados; e
- heterogeneidade dos sistemas e bancos de dados ativos.

1.3.3 Arquitetura Compilada

A arquitetura compilada não requer nenhuma atividade em tempo de execução. Quando os procedimentos de aplicação ou as operações sobre o banco de dados são compiladas, eles são modificados para incluir os efeitos das regras. Nessa arquitetura, o monitoramento dos eventos e avaliação das condições das regras intencionais não são necessários [8]. Daí obtém-se as seguintes vantagens:

- redução da complexidade da tarefa de implementação do banco de dados;
- aumento do desempenho do sistema.

Por outro lado, as desvantagens são as seguintes:

- uso de linguagens de aplicação e de regras restritas. Um exemplo desta limitação é a necessidade de detectar todos os eventos em tempo de compilação, assim, não se pode implementar os eventos temporais, eventos compostos complexos nem eventos de aplicações;
- não possibilidade de regras recursivas. Se as regras forem recursivas, isto é, uma regra pode disparar a si mesma ou disparar regras em cascata formando um ciclo, a fase de compilação poderia entrar em um ciclo infinito.

1.4 Objetivos desta dissertação

Esta dissertação tem como objetivo apresentar um novo modelo de execução para o processamento de regras de bancos de dados ativos para assegurar a propriedade de confluência, e apresentar uma implementação do novo modelo utilizando um banco de dados comercial.

Para atingir tais objetivos, este trabalho foi dividido em sete capítulos.

Os primeiro e o último capítulos são a introdução e a conclusão, respectivamente.

No capítulo 2 são apresentados os principais aspectos envolvidos no processamento de regras em de bancos de dados ativos.

Em seguida, são discutidos três implementações de regras em bancos de dados comerciais (capítulo 3).

Uma vez concluídos estes capítulos, são apresentados os principais problemas encontrados no processamento de regras (capítulo 4). Aqui apresentaremos em detalhe o problema que estamos interessados em tratar neste trabalho.

Finalmente, depois de apresentados os conceitos necessários, o capítulo 5 descreve nosso modelo para o processamento de regras, e no capítulo 6 descrevemos uma implementação deste modelo.

Capítulo 2

SEMÂNTICA DA EXECUÇÃO DAS REGRAS

O primeiro passo que temos que dar para entender o problema que estamos interessados é saber como as regas são processadas internamente por um SGBDA.

A semântica de execução das regras descreve como um SGBDA se comporta sobre um dado conjunto de regras descrevendo o processamento de regras, como ele interage com a operação normal do banco de dados e com o processamento de transações. Além das diferentes alternativas das linguagens de regras, existe um grande número de alternativas para a semântica de execução de regras. Dessa forma, mesmo para um pequeno número de regras simples, o comportamento da execução de regras pode ser muito complexo.

Como um ponto inicial para nosso estudo, vamos apresentar primeiro um modelo simples de execução de regras descrito por [8]. A seguir discutiremos algumas alternativas para este modelo.

As regras que estamos tratando, como já dissemos, são conhecidas como regras ECA (Evento - Condição - Ação). Um algoritmo para o processamento destas regras é dado a seguir.

enquanto (existem regras disparadas) faça

1. encontre uma regra R *disparada*

2. analise a condição de R

3. se a condição de R *é verdadeira, execute a ação de* R

algoritmo 2.1

Existem vários aspectos do algoritmo acima que precisamos considerar. Nas seções seguintes vamos discutir alguns deles.

2.1 Granularidade do processamento de regras

Uma das primeiras propriedades que poderíamos considerar no algoritmo 2.1 é o quão freqüentemente ele é executado. A granularidade do processamento de regras especifica essa propriedade, isto é, quão freqüentemente as regras são processadas. Uma granularidade fina significa "sempre" – regras podem ser processadas em qualquer ponto durante a execução do sistema, assim que qualquer regra é disparada. É claro que isso faz sentido apenas quando estamos considerando que as regras podem ser disparadas por um evento temporal ou um estado que o banco de dados atingiu. Caso contrário é necessária a ocorrência de pelo menos uma operação que gerasse o disparo da regra. Nesse outro nível de granularidade, considerado ainda fino, as regras podem ser processadas a cada ocorrência da "menor" operação do banco de dados. Por exemplo, em um SGBDA relacional, essa granularidade poderia corresponder a uma inserção, deleção ou atualização de um registro simples.

Para exemplificar considere um evento E que insere três registros na tabela funcionários (um comando SQL). Suponha R uma regra que é disparada quando um registro é inserido na tabela funcionários. Se a granularidade é fina, então a regra R será disparada uma vez para cada um dos registros inseridos pelo evento E, e em cada uma das vezes, a regra R será processada assim que cada registro for inserido.

Em alguns sistemas de banco de dados, os comandos de manipulação de dados fornecem uma granularidade para o processamento de regras que é "mais grosso" que as operações do banco de dados. Por exemplo, em um banco de dados relacional as regras poderiam ser processadas no final de cada comando SQL, onde um comando Insert, Delete ou Update afeta diversos registros. Durante o processamento da operação, os comandos são agrupados em transações; assim, uma outra granularidade poderia ser determinada por transação, isto é, o algoritmo é executado ao final de cada transação.

Considerando o mesmo exemplo que demos acima, mas agora para uma granularidade grossa e que o evento E está inserido em uma transação Tr, a regra R só será disparada depois que os três registros tiverem sido inseridos na tabela funcionários.

2.2 Execução orientada a instância x orientado a conjunto

A execução de regras em bancos de dados ativos é orientado a instância se a regra é executada uma vez para cada instância do banco de dados que dispara a regra ou satisfaz a condição da regra. Por exemplo, considere uma regra R_1 que é disparada pela remoção de um registro em uma tabela específica. Se a execução da regra é orientada a instância, então a condição da regra é avaliada e sua ação é executada uma vez para cada registro excluído.

Segundo [8], quando utilizamos o algoritmo 2.1 com uma execução orientada a instância devemos escolher entre dois métodos de execução:

- quando uma regra é selecionada no passo (1), os passos (2) e (3) são executados para cada instância relevante à regra em questão; ou
- uma vez que uma regra é selecionada no passo (1), os passos (2) e (3) são executados para uma única instância relevante à regra; quando o laço do algoritmo continuar, outra regra será selecionada podendo, inclusive, ser a mesma.

A execução é orientada a conjunto se uma regra é executada uma vez para todas as instâncias que dispararam a regra ou satisfizeram a condição da regra. Por exemplo, na regra R_1 descrita acima, se a execução da regra é orientada a conjunto então a condição da regra é avaliada e sua ação é executada uma vez para o conjunto inteiro de registros apagados.

A diferença no efeito real entre a execução orientada a instância e orientada a conjunto pode ser sutil. Por exemplo, considere a regra R em um banco de dados relacional ativo que é disparada pela inserção de registros de funcionários e suponha que a ação de R atualiza o valor no atributo S dos registros inseridos para ser a média dos valores de S, considerando todos os registros dos funcionários. Na execução orientada a conjunto, já que R executa uma vez para todos os funcionários inseridos, todos os funcionários inseridos recebem o mesmo valor de S. Entretanto, na execução orientada a instância, como R executa uma vez para cada funcionário inserido, cada funcionários inserido pode ter um valor diferente em S. Para ilustrar este fato considere a seguinte tabela de funcionários com 2 funcionários já inseridos:

Id	...	Salário
1	...	10
2	...	6

Considere então uma inserção de dois registros de funcionários nesta tabela:

Id	**...**	**Salário**
3	...	4
4	...	8

Na execução orientada a conjunto teríamos a seguinte tabela final:

Id	**...**	**Salário**
1	...	10
2	...	6
3	...	7
4	...	7

Já se a execução fosse orientada a evento teríamos a seguinte tabela final:

Id	**...**	**Salário**
1	...	10
2	...	6
3	...	6,7
4	...	7,7

O comportamento do algoritmo 2.1 para a execução orientada a conjunto é clara: os passos 2 e 3 são executados para o conjunto inteiro de registros afetados pela regra selecionada no passo 1. Entretanto, para o modelo orientado a instância, existe mais uma escolha a ser feita. Ou para uma regra que é selecionada no passo 1, os passos 2 e 3 são executados uma vez para cada instância relevante da regra, ou para uma regra selecionada no passo 1, os passos 2 e 3 são executados apenas para uma instância relevante da regra; quando o laço continua, a mesma regra, ou outra regra diferente pode ser selecionada.

2.3 Algoritmo recursivo x iterativo

O algoritmo 2.1 é iterativo: ele seleciona e processa uma regra, então seleciona e processa outra, e assim sucessivamente. Entretanto considere um cenário no qual a execução da ação de uma regra pode produzir alterações na base de dados que são maiores que a granularidade do processamento da regra. Por exemplo, suponha que em um SGBDA relacional o processamento de

regras ocorre depois da alteração em um registro simples, mas a ação de uma regra pode executar comandos SQL; ou suponha que o processamento de regras ocorre após comandos SQL, mas a ação da regra pode executar vários comandos. Em tais casos o processamento de regras pode ser iniciado durante a execução da ação de uma regra, i.é., o processamento de regras pode ser iniciado recursivamente. O comportamento do processamento recursivo pode ser consideravelmente diferente do comportamento do processamento iterativo.

Para ilustrar estas possíveis diferenças de comportamento considere a seguinte tabela de funcionários com 2 funcionários já inseridos:

Id	Cargo	...	Salário
1	1	...	10
2	1	...	6

Considere também a tabela seguinte que descreve os salários máximos permitidos para cada cargo.

Cargo	Salário
1	10
2	6
3	8

Suponha agora um evento que irá inserir na tabela funcionários os seguintes registros:

Id	Cargo	...	Salário
3	2	...	5
4	3	...	1

Para a tabela funcionários, estamos considerando as seguintes regras:

R1 – disparada quando um registro é inserido na tabela de funcionários e sua ação é atualizar o valor do atributo S para a média dos valores de S de todos os funcionários já inseridos.

R2 – disparada quando um registro é atualizado na tabela de funcionários e sua ação é atualizar o valor de S para o máximo permitido para o cargo do funcionário.

Para concluir o ambiente do exemplo, vamos tomar um sistema com execução orientado a instância (veja seção 2.2).

Na execução iterativa teremos a seguinte seqüência de execução:

(1) inserção do funcionário de id = 3 (vamos chamá-lo de f3)

(2) disparo da regra R1

(3) atualização do valor atributo S de f3 para 7

(4) inserção do funcionário de id = 4 (vamos chamá-lo de f4)

(5) disparo da regra R1

(6) atualização do valor do atributo S de f4 para 6

(7) disparo da regra R2

(8) atualização do valor do atributo S de f3 para 6

Dessa execução resultaria a seguinte tabela:

Id	Cargo	...	Salário
1	1	...	10
2	1	...	6
3	2	...	6
4	3	...	6

Na execução recursiva teremos a seguinte seqüência de execução:

(1) inserção do funcionário de id = 3 (vamos chamá-lo de f3)

(2) disparo da regra R1

(3) atualização do valor atributo S de f3 para 7

(4) disparo da regra R2

(5) atualização do valor do atributo S de f3 para 6

(6) inserção do funcionário de id = 4 (vamos chamá-lo de f4)

(7) disparo da regra R1

(8) atualização do valor do atributo S de f4 para 5,5

Dessa execução resultaria a seguinte tabela:

Id	Cargo	...	Salário
1	1	...	10
2	1	...	6
3	2	...	6
4	3	...	5,5

Nessa dissertação estaremos considerando o algoritmo iterativo.

2.4 Execução seqüencial x concorrente

O algoritmo 2.1 realiza o processamento de regras de maneira seqüencial: uma regra é executada por vez. Mesmo no caso recursivo uma regra é executada por vez. Para esse tipo de processamento seqüencial, um mecanismo de resolução de conflitos pode ser usado para escolher qual regra será executada quando múltiplas regras são disparadas(veja seção 2.5 a seguir). Uma alternativa para a execução seqüencial é a execução concorrente: se múltiplas regras são disparadas, as condições das regras são avaliadas e suas ações são executadas concorrentemente. A execução concorrente evita ter que usar um método de resolução de conflitos (entretanto, algum tipo de controle de concorrência é necessário para que a execução de uma regra não interfira com a outra). Podem existir diferenças significantes entre os dois modos de execução. Não vamos discutir esse ponto em detalhes aqui, mas voltaremos a ele mais tarde quando estivermos tratando do ponto principal do trabalho.

2.5 Modos de acoplamento

Vamos agora analisar o relacionamento entre o processamento de regras e transações do banco de dados.

O modo mais simples de relacionamento ocorre quando a condição de uma regra é avaliada e sua ação executada dentro da mesma transação que o evento que disparou a regra, no ponto mais próximo do processamento da regra. Entretanto, para algumas aplicações, pode ser útil esperar a avaliação da condição da regra ou a execução da sua ação até o fim da transação; ou, pode ser útil avaliar a condição da regra ou executar sua ação em uma transação separada. Estas possibilidades produzem a noção de modos de acoplamento[8], que especificam o relacionamento transacional entre o evento que disparou a regra e a avaliação de sua condição e entre a avaliação da condição e a execução de sua ação. Possíveis modos de acoplamento são:

- *Imediato*: o processamento da regra acontece imediatamente após o evento, dentro da mesma transação. Por exemplo, um modo Imediato para a ação com respeito a condição significa que a ação é executada tão logo a condição seja avaliada (considerando, é claro, que a condição tenha sido avaliada como verdadeira).

- *Adiado*: o processamento da regra ocorre no ponto de commit da transação atual. Por exemplo, um modo adiado para a condição com relação ao evento significa que se o evento ocorre durante a transação Tr, então a condição é avaliada no final da transação Tr. O modo Adiado pode ser útil, por exemplo, para regras que reforçam restrições de integridade, já que a transação pode executar várias operações que violam a restrição, mas a transação pode restaurar a restrição antes de chegar ao ponto de commit.

- *Desacoplado*: nesse caso o processamento da regra é feito em uma transação separada. Esse modo pode ainda ser subdividido em Desacoplado dependente, onde uma transação separada não é criada antes de a transação original ter feito o commit, e Desacoplado independente, onde uma transação separada é iniciada independente da transação original ter terminado. Além disso, uma "causalidade" entre as transações poderia ser especificada, tal como exigir que a transação "filha" termine após a transação original na ordenação serial. O modo Desacoplado pode ser útil quando uma longa série de regras é disparada, para dividir a transação grande resultante em um conjunto de transações menores.

2.6 Resolução de conflitos

O algoritmo 2.1 iterativamente seleciona uma regra e a processa, depois seleciona outra e processa e assim sucessivamente, isto é, ele executa seqüencialmente. No passo 1 do algoritmo 2.1 é possível que exista mais que uma regra disparada. Isso pode acontecer, entre outros motivos, porque:

- várias regras podem ter sido disparadas pelo mesmo evento, ou
- a granularidade do algoritmo (o quão freqüentemente ele e executado) pode ser grossa o suficiente para que vários eventos, que disparam regras aconteçam antes que todas as regras sejam processadas.

Portanto é necessário escolher no passo 1 do algoritmo 2.1 a regra que será processada. Isso é chamado de resolução de conflitos[8]. Algumas políticas para resolução de conflitos são:

- escolher uma regra arbitrariamente
- definir prioridades para as regras quando estas são criadas e fazer a escolha segundo estas prioridades
- escolher a regra que foi disparada a mais tempo

Uma vez entendido o modo como as regras são executadas é preciso analisar o comportamento das regras dentro do sistema. Vamos tratar desse assunto no capítulo quatro.

Capítulo 3

IMPLEMENTACAO DE REGRAS EM BANCOS DE DADOS COMERCIAIS

Nesta seção vamos apresentar um pouco de como as regras estão sendo implementadas em bancos de dados comerciais.

A introdução de regras em sistemas de banco de dados comerciais foi inicialmente motivada para tratamento de restrições de integridade entre tabelas (como visto no capítulo anterior) e implementação de políticas de gerenciamento das empresas. O nome Trigger foi adotado para regras.

As regras foram parcialmente implementadas nos bancos de dados comerciais, i.e., não se implementou todas as operações possíveis de serem realizadas com regras, nem todas as ferramentas de suporte a regras. O que se observa é que a complexidade das propostas de implementação tem sido muito maior que a complexidade das implementações que têm sido feitas[5]. Essa diferença de complexidade se dá desde os tipos de eventos detectados pelas regras até as ferramentas para suporte a regras (como um browser, um analisador, um debugger, e outras). Ou seja, não temos uma grande variedade de eventos que possam ser detectados por uma regra, como um timer, um login ou coisa semelhante, mas nos restringimos a eventos aplicados sobre os dados de uma tabela, como um Insert ou um Delete. Quanto à falta de ferramentas, não ter um debuger, por exemplo, é como desenvolver um programa em uma linguagem de programação, como C, e não ter o debuger para analisar a iteração do programa.

O principal motivo para esta diferença é que as implementações propostas levam a uma grande complexidade no tratamento da execução das regras e o que se deseja de imediato é obter os benefícios da utilização das regras sem ter que se preocupar com esta complexidade gerada quando se implementam as regras num SGBD.

Vamos descrever aqui as implementações de três bancos de dados disponíveis comercialmente. No ambiente de banco de dados comerciais não vamos mais utilizar o nome 'regra', mas sim vamos utilizar o nome 'trigger' pois este é o nome adotado por estas

implementações e também porque nos lembra que se trata de uma implementação "simplificada" das regras como acabamos de mencionar.

Antes, porém, de entrarmos nas implementações particulares dos três sistemas, vamos apresentar uma característica que é vital aos bancos de dados ativos: os valores de transição.

3.1 Valores de transição

Transações podem consistir em mudanças tão pequenas quanto a mudança de um único registro ou tão grande quanto uma mudança no banco de dados inteiro. As linguagens de regras normalmente incluem um mecanismo para se referir a valores antes e depois de operações de modificação do banco de dados. Valores de transição podem ser acessados através de palavras reservadas. Por exemplo, **inserted**, **deleted**, e **updated** podem ser usados para representar dados que foram inseridos, apagados ou alterados, respectivamente, na *transição.*

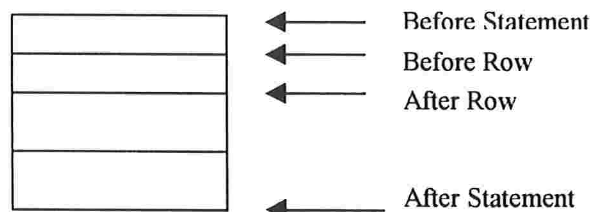
Uma transição é uma mudança no estado de um banco de dados gerado pela execução de uma seqüência de operações de manipulação de dados. Regras consideram apenas os valores "mais próximos" da transição[1]. No modelo geral isso significa que:

- se um registro é atualizado várias vezes, somente a atualização composta é considerada;
- se o registro é atualizado e a seguir ele é apagado, somente a remoção é considerada;
- se um registro é inserido e a seguir é atualizado, isso é considerado como uma inserção do valor atualizado;
- se um registro é inserido e então apagado, nada é considerado.

Mas assim como o modelo de execução das regras varia de implementação para implementação, o modelo de transição também pode possuir muitas diferenças dependendo da implementação. As diferenças estão na maneira como as regras se referem aos valores de transição (valores dos registros antes e depois de uma operação de manipulação de dados). Por exemplo, num sistema a palavra inserted pode se referir a um conjunto de registros que tenham sido inseridos, enquanto em outro ela pode se referir apenas a um único valor. Também a palavra old em um sistema pode se referir a um valor anterior de um registro atualizado, enquanto em outro ela se refere ao estado anterior do banco de dados antes do comando mais recente, ou pode também, num outro sistema, se referir ao valor inicial, antes do inicio do processamento de um registro[8].

3.2. Oracle

A primeira implementação que vamos descrever é do banco de dados Oracle7. Ele suporta triggers que são executadas antes ou depois do evento que provoca o disparo da regra, isto é, a trigger é disparada imediatamente antes do evento ser processado ou imediatamente após ele ter sido concluído. Um outro aspecto muito importante é que quando um evento afeta vários registros de uma tabela ele pode provocar o disparo de uma regra de duas maneiras: um disparo da regra para todo o evento ou um disparo da regra para cada registro que sofre a influência do evento. Isto produz quatro possíveis combinações: Before Row, Before Statement, After Row e After Statement. O diagrama a seguir mostra estas quatro possíveis combinações, sendo que cada linha do diagrama representa um registro e todo o bloco representaria então o conjunto de registros afetados pelo disparo de uma regra.



Uma trigger pode monitorar mais de uma das operações de alterações de dados (Insert, Delete ou Update) sobre a mesma tabela, mas cada operação (incluindo atualizações em qualquer coluna) pode ser monitorada por uma única trigger. Assim, combinando estas três operações com as quatro combinações discutidas acima, podemos ter no máximo 12 triggers por tabela: Before Row Insert, Before Statement Insert, After Row Insert, After Statement Insert, Before Row Delete, Before Statement Delete, After Row Delete, After Statement Delete, Before Row Update, Before Statement Update, After Row Update, After Statement Update.

A sintaxe das triggers no Oracle 7 é:

< Trigger Oracle > :: = { CREATE | REPLACE } TRIGGER < nome da trigger >

{ BEFORE | AFTER } { INSERT | DELETD | UPDATE [OF < nome das colunas >] }

ON < nome da tabela >

[[REFERENCING { OLD AS < nome antigo do registro > |

NEW AS < nome novo do registro > }

FOR EACH ROW

[WHEN < condição >]]

< bloco de comandos PL/SQL >

A condição da trigger (clausula WHEN) é suportada em conjunção com a opção FOR EACH ROW, e assim ela é testada para cada registro que a trigger afeta. Quando a condição não é verdadeira para um certo registro, a ação da trigger não é disparada para aquele registro, mas a execução da trigger não é afetada, isto é, a trigger não é abortada.

A ação disparada é um bloco procedural escrito em PL/SQL. Procedimentos PL/SQL em ações de triggers podem incluir declarações de variáveis e/ou chamadas a procedimentos externos, mas eles não podem incluir comandos de controle de transações. As referências OLD e NEW são manipuladas similarmente ao SQL3, exceto que elas são restritas ao FOR EACH ROW.

A clausula REFERENCING raramente é usada, e seu objetivo é eliminar conflitos de nomes quando existir uma tabela chamada new ou old. Isso porque as tabelas de transição são referenciadas como new e old.

Quando uma trigger pode ser disparada por mais de um evento (insert, update ou delete), os predicados especiais INSERTING, DELETING e UPDATING podem ser usados na ação para detectar que evento disparador ocorreu. Por exemplo, assuma a seguinte declaração de trigger:

CREATE TRIGGER T1 INSERT OR UPDATE ON emp

No corpo da trigger você pode inserir as seguintes condições:

IF INSERTING THEN ... ENDIF

IF UPDATING THEN ... ENDIF

A primeira condição é verdadeira, e seus comandos são executados, se o evento que disparou a trigger foi um evento de inserção na tabela emp. Para a segunda condição vale o mesmo quando o evento que disparou for uma atualização.

Num evento *Update* é possível controlar qual atributo da trigger foi afetado pela modificação. Considere o trecho de código abaixo.

```
CREATE TRIGGER T1

UPDATE OF sal, nome ON emp

BEGIN

    IF UPDATING ('SAL') THEN . . . END IF;

    ...

END;
```

A trigger T1 só será disparada se a atualização da tabela emp for realizada nas colunas sal ou nome da tabela emp. Se a atualização for realizada em alguma outra coluna, a trigger não é disparada. Além disso, no corpo da trigger, é possível especificar comandos diferentes para cada coluna que tenha sido afetada pela modificação (nesse caso a coluna sal).

Agora suponha que todos os quatro tipos de triggers são especificados para a mesma operação SQL, e a operação ocorre. O seguinte algoritmo é executado para o processamento da trigger:

1. *Execute a trigger* BEFORE STATEMENTE

2. *Para cada registro afetado pela operação SQL:*

    (a) *Execute a trigger* BEFORE ROW

    (b) *bloqueie e altere o registro, então realize a checagem de integridade referencial para o registro e asserção (O bloqueio não é liberado até que a transação sofra um commit).*

    (c) *Execute a trigger* AFTER ROW

3. *Realize a checagem da integridade referencial para o registro e asserção.*

4. *Execute a trigger* AFTER STATEMENT.

É importante notar que os passos 1, 2(a), 2(c) e 4 poderiam chamar o algoritmo inteiro recursivamente se existissem triggers definidas sobre operações SQL executadas pelas ações das triggers. Se um erro definido pelo sistema, um erro definido pelo usuário, ou ainda uma exceção

ocorre durante a execução das ações da trigger, então todas as alterações realizadas no banco de dados pela trigger e também as alterações feitas pelo evento que disparou a trigger serão abortadas.

Triggers podem ser disparadas em cascata como resultado do processamento de triggers recursivas, mas existe um número máximo de triggers disparadas em cascata que é 32. Este número pode ser modificado como o parâmetro de inicialização MAX_OPEN_CURSORS. Já que o sistema não garante qualquer ordem particular para os registros processadas por uma operação SQL, o resultado pode ser não determinístico.

3.3 Sybase

No Sybase em cada tabela podem ser definidas no máximo três triggers: uma para Insert, outra para Delete e uma última para Update. Todas as triggers do Sybase são disparadas para cada comando (statement-level) e são sempre executadas após a operação que as disparou. A sintaxe geral para uma trigger no Sybase é a seguinte:

< Trigger 1 Sybase > :: = CREATE TRIGGER < nome da trigger >

ON < nome da tabela >

FOR { INSERT | DELETE | UPDATE }

AS < comandos SQL >

No entanto, existe uma segunda sintaxe onde é criada a possibilidade de uma trigger monitorar um ou mais atributos de um registro individualmente. Nessa sintaxe, eventos Update podem ser testados para atributos específicos, e conectivos lógicos AND e OR podem ser usados para os atributos desejados. Por exemplo, suponha que em um registro da tabela funcionários estamos interessados em que uma regra seja disparada quando o campo salário for alterado. Neste caso, não definimos uma trigger que atue sobre uma atualização em qualquer atributo do registro, mas somente se o atributo que sofreu alteração for o atributo salário. Essa segunda sintaxe é mostrada abaixo:

< Trigger 2 Sybase > :: = CREATE TRIGGER < nome da trigger >

ON < nome da tabela >

FOR { INSERT | UPDATE }

AS IF UPDATE(<nome do atributo>)

[{ AND | OR } UPDATE(<nome do atributo>) ...]

< comandos SQL >

Em ambos os casos, não existe a parte da sintaxe que trate da condição. Entretanto, os comandos SQL que formam as ações da trigger podem incluir estruturas de controle e testes, o que permite que as ações das triggers sejam executadas condicionalmente. Vários tipos de comandos, incluindo comando de definição e consultas Select, não são permitidos em ações das triggers. Entretanto, comandos ROLLBACK TRANSACTION e ROLLBACK TRIGGER são suportados. ROLLBACK TRANSACTION aborta toda a transação enquanto ROLLBACK TRIGGER aborta a operação que disparou a trigger e qualquer ação subsequente disparada.

O Sybase suporta duas tabelas temporárias definidas pelo sistema chamadas INSERTED e DELETED. Essas tabelas incluem todos os registros que foram inseridas ou removidas mais recentemente. As tabelas são computadas uma vez depois da execução de um comando Insert, delete ou update, mas antes do início do processamento da trigger.

Triggers podem se encadear em cascata até um limite de 8, isto é uma primeira trigger pode, pela execução da sua ação, levar ao disparo de uma segunda trigger cuja ação dispara uma terceira e assim sucessivamente até a oitava trigger ser disparada. Se esta última provocar o disparo de uma outra, então todo processamento, desde a primeira, é abortado.

O cascateamento também pode ser desabilitado com um comando de configuração. Normalmente uma trigger não pode disparar a si mesma, mas outro comando de configuração pode ser utilizado para permitir isso.

3.4 SQL Server

Triggers são tratadas no SQL Server como um tipo especial de procedimento que é disparado automaticamente toda vez que dados são modificados em uma tabela específica.

Cada trigger pode se referir a uma das operações INSERT, UPDATE ou DELETE que atuem sobre uma tabela, ou podem se referir a uma combinação destas operações. Assim, temos no máximo três triggers por tabela e cada uma se aplica somente a uma tabela.

Uma trigger é disparada uma vez por comando SQL imediatamente após um comando de modificação dos dados ter sido completado. A trigger e o comando que a disparou são tratados como uma transação simples que pode ser abortada de dentro da trigger. Se um erro severo é detectado, a transação inteira é abortada automaticamente. Uma transação é aceita ou rejeitada como um todo pela trigger, i.é., se um registro é rejeitado, toda a transação é abortada. Entretanto, não é necessário abortar toda a transação quando apenas uma modificação não foi aceita. Usando uma subconsulta relacionada é possível forçar a trigger a testar os registros uma a uma.

Triggers podem se encadear em cascata até um máximo de dezesseis, mas o encadeamento pode ser desabilitado na configuração do sistema. A recursividade (uma trigger disparar a si mesma) não é permitida. Em outras palavras, uma trigger não dispara a si mesma em resposta a uma segunda modificação na mesma tabela em que a trigger é definida.

Assim como no Sybase, o SQL Server especifica duas tabelas especiais: INSERTED e DELETED. Elas são temporárias e usadas para testar o efeito das modificações nos dados.

- A tabela INSERTED armazena cópias dos registros afetadas durante um Insert ou Update. Os registros são inseridos na tabela onde a trigger está definida e na tabela INSERTED ao mesmo tempo.
- A tabela DELETED armazena cópias dos registros afetadas durante um Delete ou Update. Os registros são removidos da tabelas onde a trigger está definida e movidos para a tabela DELETED.

A sintaxe para as triggers em SQL Server é semelhante à sintaxe do Sybase, tendo também duas variantes, podendo-se definir se a trigger é disparada quando uma modificação é realizada sobre um atributo específico de um registro. A sintaxe geral é:

< *Trigger 1 SQL Server*> :: = CREATE TRIGGER *[proprietário.]* < *nome da trigger* >

ON *[proprietário.]* < *nome da tabela* >

FOR { INSERT | UPDATE | DELETE }

[WITH ENCRIPTION]

AS < *comandos SQL* >

A sintaxe onde se leva em consideração qual atributo foi afetado pela modificação é:

< *Trigger 2 SQL Server*> :: = CREATE TRIGGER *[proprietário.]* < *nome da trigger* >

ON *[proprietário.]* < *nome da tabela* >

FOR { INSERT | UPDATE | DELETE }

[WITH ENCRIPTION]

AS IF UPDATE (< *nome da coluna* >)

[{ AND | OR } UPDATE (< *nome da coluna* >)] <
comandos SQL >

3.5 Quadro comparativo

Vamos descrever aqui um quadro comparativo entre as três implementações vistas.

	Oracle	**Sybase**	**SQL Server**
Triggers disparadas	Antes ou depois do evento	Após o evento	Após o evento
Granularidade	Registro ou comando	Comando	Comando
Triggers por tabela	12	3	3
Ações	Não suporta controle de transações	Não suporta definições	Não suporta definição nem controle de concorrência
Cascateamento	32	8	16
Recursividade	Permitida	Pode ser permitida	Não é permitida

Critério de comparação:

- *Triggers disparadas*: uma vez que um evento tenha ocorrido, quando é possível ativar a trigger (imediatamente antes ou depois do evento)
- *Granularidade*: a trigger será disparada uma única vez por evento, ou uma vez para cada registro que o evento modifica
- *Triggers por tabela*: quantas triggers podem ser associadas a uma tabela
- *Ações*: quais ações não podem ser escritas na parte das ações de uma trigger. No nosso caso, as ações de controle de transação (como Rollback) e definição de novos objetos (como CREATE TABLE).
- *Cascateamento*: a partir do disparo de uma trigger por um evento, quantas triggers podem ser disparadas em cascata pela ação da trigger anterior.
- *Recursividade*: se é possível que uma trigger dispare a si mesma em resposta a sua própria ação.

Capítulo 4

ANALISE DE REGRAS

Um problema difícil no projeto de regras de uma aplicação é predizer como as regras se comportarão em todas as situações possíveis. Esse problema se torna muito evidente quando o projetista adiciona novas regras a uma aplicação existente, já que novas regras podem interagir com regras preexistentes de modo não esperado. Para algumas linguagens de regras de banco de dados ativos é possível realizar análises estáticas automáticas sobre conjuntos de regras para encontrar algumas de suas propriedades e predizer o comportamento de um conjunto de regras em tempo de execução. Por exemplo, é muito útil conhecer a priori que a execução da regra irá garantidamente terminar sob qualquer circunstancia (isto é, independente do que disparou a transação ou do estado do banco de dados). Quando propriedades úteis não são garantidas, as técnicas da análise de regras podem isolar as regras responsáveis pelo problema.

Existem, infelizmente, certas deficiências com a análise de regras:

- As técnicas de análise de regras são aplicadas estaticamente, não levando em conta o estado atual do banco de dados. Portanto, a análise pode determinar que uma certa propriedade não é garantida, mesmo que o estado atual do banco de dados seja tal que garanta que a propriedade seja sempre garantida.
- Técnicas de análise são altamente dependentes da construção da linguagem das regras. Eventos, condições e ações das regras devem ser analisados individualmente.
- Técnicas de análise das regras são dependentes da semântica da execução as regras, que pode variar consideravelmente de sistema para sistema. Por exemplo, em um sistema o processamento pode ser recursivo e em outro iterativo ou, um sistema pode processar as regras em transações separadas e outro dentro de uma mesma transação. Uma forte desvantagem dessas diferenças é que regras que são sintaticamente iguais podem ter comportamentos bem diferentes em sistemas diferentes.

É claro, em adição às propriedades acima, as regras devem ser "corretas", significando que elas realizam exatamente a atividade para a qual foram projetadas. Em muitos casos a corretude de

cada regra individualmente é óbvia, então a primeira providência é entender e predizer a iteração entre as regras. Analisar as propriedades formais tais como as descritas acima é, em geral, um bom primeiro passo (e normalmente suficiente) para garantir que as interações das regras sejam corretas.

Portanto, a análise de regras é realizada estaticamente em tempo de compilação sobre uma dada coleção de regras. A meta da análise de regras é estabelecer suas propriedades formais que sejam válidas para transações e estados da base de dados arbitrários. Duas destas propriedades de maior interesse são:

- *Terminação*: nós dizemos que o processamento de regras de um banco de dados ativo *termina* ou que o sistema *garante a terminação*, se para qualquer evento e estado de banco de dados, o processamento de regras sempre termina, i.é, as regras não disparam outras indefinidamente.
- *Confluência*: dizemos que a execução das regras em um banco de dados é confluente se o processamento de regras irá produzir o mesmo estado final no banco de dados, independente da ordem de execução de regras não priorizadas disparadas simultaneamente.

Vamos analisar um pouco mais detalhadamente cada uma destas prioridades nas seções seguintes.

4.1.TERMINAÇÃO

Na maioria dos algoritmos de processamento de regras, não importando se ele é iterativo ou recursivo, existe o risco da não-terminação. Por exemplo, no algoritmo 2.1 o laço continua até que não haja mais regras disparadas. Se a execução da ação de uma regra pode disparar outras regras - ou disparar a si mesma - então é possível que as regras sejam disparadas indefinidamente.

Existem vários modos de se tratar o problema de terminação:

- A não-terminação é aceita como uma possibilidade e é responsabilidade do projetista das regras garantir que ela não ocorra.
- Estipula-se um limitante superior (possivelmente como parâmetro do sistema) para o número de regras disparadas em cadeia. Quando o limite é atingido, o processamento é abortado.

- Adiciona-se restrições sintáticas ao conjunto de regras para garantir que o processamento sempre termine. Um método bem simples é garantir que uma regra não dispare outras. Um método mais sofisticado seria permitir que as regras disparem outras mas impedir que ocorram ciclos, isto é, uma regra não pode, direta ou indiretamente disparar a si mesma. Um terceiro método mais sofisticado poderia permitir ciclos, mas deveria garantir que ao longo do ciclo uma condição de uma regra deveria se tornar falsa.

O modo típico para analisar um conjunto de regras ativas quanto à terminação é construir um grafo de disparo. Os nós do grafo correspondem às regras. A figura 4.1 a seguir apresenta um grafo de disparo simples. Existe uma aresta do nó R_i para o nó R_j se e somente se a execução da ação da regra R_i pode disparar a regra R_j. Se existe um ciclo no grafo então as regras podem ser disparadas indefinidamente e a propriedade da terminação não é garantida (figura 4.2).

Fig. 4.1 - A ação de Ri dispara Rj

Fig. 4.2 - Um ciclo que pode provocar a não-terminação

A análise da terminação é geralmente *conservadora* [8]. Isto significa que se a análise determina que o processamento das regras garantidamente termina, então de fato a propriedade vale. Entretanto, a análise poderia determinar que as regras "podem não terminar", mesmo quando a propriedade é válida. Portanto, na prática, ciclos em grafos de disparo simplesmente indicam todas as possíveis causas da não-terminação. O projetista pode então analisar cada uma das possíveis causas , e determinar que o processamento de regras garantidamente termine (devido a semânticas

adicionais de regras, ou propriedades conhecidas do banco de dados), ou o projetista pode mudar as regras e executar a análise novamente.

A exatidão da análise - quão conservadora ela é - depende em grande parte da exatidão do procedimento utilizado para colocar as arestas no grafo de disparo. Usando uma análise simples existe uma aresta de R_i para R_j quando qualquer ação da regra R_i corresponde a um evento que dispara a regra R_j. Usando uma análise mais rigorosa e precisa é possível, em alguns casos, determinar que embora a ação de R_i contenha um evento de R_j, depois da execução de R_i a condição de R_j sempre será falsa. Em tais casos, não existe aresta de R_i para R_j.

Mesmo com uma análise mais precisa, existem alguns casos onde ciclos são produzidos no grafo mas a terminação ainda é garantida. Por exemplo, a terminação é garantida se regras cíclicas realizam operações monotônicas com um limite finito, como remover mas nunca inserir em uma tabela.

Neste trabalho a terminação é assumida como válida e não é objeto de nossa investigação mais detalhada.

4.2. CONFLUÊNCIA

Estamos interessados aqui em que, independente da ordem de execução das regras, o processamento de regras produza o mesmo resultado final no banco de dados. Isso pode ser ilustrado pela figura abaixo.

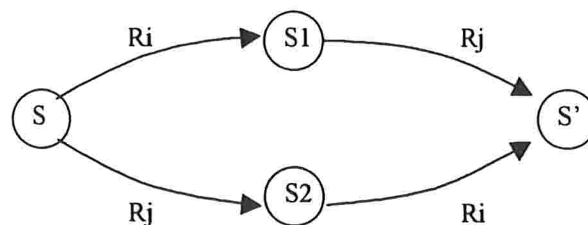


Fig 4.3 Regas Comutáveis

Na figura acima, S é o estado inicial do banco de dados. Ri e Rj são duas regras que são disparadas e eleitas para execução sobre o estado S. Embora Ri e Rj possam levar o estado inicial do

banco de dados a estados intermediários diferentes (S1 e S2 respectivamente), o estado final atingido após o processamento de Ri e Rj é o mesmo: S'.

A ausência da confluência impede que possamos prever o inter-relacionamento entre as regras, isto é, quando várias regras estão disparadas ao mesmo tempo, e a ordem de execução influi no resultado final das suas ações, não podemos determinar o estado do banco da dados após a execução destas regras. Também não conseguimos determinar a influência de uma nova regra inserida no banco de dados pode causar nas demais regras já existentes. Dessa forma, trabalhos como [11] que trazem diversas vantagens ao projeto de regras para bancos de dados ativos, como a modularização do sistema de regras, não podem ser implementados.

Vamos fazer agora algumas considerações sobre como garantir a confluência.

Em primeiro lugar, se todas as regras de um conjunto de regras de interesse são priorizadas (isto é, cada uma delas recebe um número que indica sua prioridade na ordem de execução das regras), então o conjunto de regras é completamente determinístico e a confluência é garantida. A confluência também é garantida se é certo que duas regras não serão disparadas ao mesmo tempo. Entretanto, se múltiplas regras podem ser disparadas ao mesmo tempo, e se as regras não são completamente priorizadas, então a confluência não é garantida a priori. Em geral, analisar as regras com relação à confluência é consideravelmente mais complexo que analisar a terminação.

Vários trabalhos foram desenvolvidos tratando do problema da confluência. Em [1], a confluência é analisada num contexto simples onde o impacto de eventos e os estados do banco de dados não são considerados. Já em [4] estes aspectos são considerados, porém quando é realizada a análise de complexidade do algoritmo desenvolvido neste artigo, vemos que o método é impraticável no caso geral, isto é, para casos reais onde o número de regras a serem analisadas é considerável. Da mesma forma, [9] desenvolve um novo algoritmo para análise da confluência e verifica que a complexidade é exponencial. Nosso trabalho difere essencialmente dos citados acima pois não objetiva desenvolver um novo algoritmo ou método para analisar a confluência das regras, mas sim dar uma nova semântica de execução.

4.2.1. Exemplo do problema de confluência

Vamos descrever aqui um exemplo simples de como pode ocorrer o problema da confluência. Considere uma aplicação que contenha as seguintes tabelas:

pecas (id, nome, preco)

tab1 (num)

tab2 (num)

Considere também que temos as seguintes triggers definidas para cada tabela como a seguir:

- *Trigger da tabela pecas*: ao ser inserida uma nova peça nesta tabela, insere um valor na tabela tab1 e outro na tabela tab2.

```
CREATE TRIGGER T_pecas
ON pecas
FOR INSERT
AS
        INSERT tab1 VALUES (70)
        INSERT tab2 VALUES (80)
```

- *Trigger da tabela tab1*: ao ser inserida um novo registro nesta tabela, atualiza o preço da peça com id = 1 para 10,00.

```
CREATE TRIGGER T_tab1
ON tab1
FOR INSERT
AS
        UPDATE pecas SET preco = 10,00 WHERE id = 1
```

- *Trigger da tabela tab2*: ao ser inserido um novo registro nesta tabela, apaga da tabela pecas, todas as peças com valor igual a 10,00.

```
CREATE TRIGGER T_tab2
ON tab2
FOR INSERT
AS
        DELETE pecas WHERE preco = 10,00
```

Dada a inserção do seguinte registro na tabela pecas:

(1, "agulha", 20,00)

Existem duas possíveis execuções das triggers acima.

Caso a trigger da tabela tab1 seja executada primeiro, este registro que acabou de ser inserido será removido da tabela. Caso a trigger da tabela tab2 seja executada primeiro, teremos o registro acima inserida na tabela de peças, mas com preço de 10,00.

Vamos descrever na seção seguinte o método desenvolvido em [1] por Alexander Aiken, Jennifer Widom e Joseph M. Hellerstein, para dar uma visão mais exata sobre os métodos, e a complexidade envolvida, que citamos acima. Esse método é baseado num protótipo do sistema gerenciador de bando de dados Starburst.

4.2.2 Um método para analisar a confluência

O método apresentado por [1] consiste em um algoritmo conservador que recebe um conjunto de regras e garante sua confluência ou diz que este conjunto pode não ser confluente. Quando a resposta é a segunda, o algoritmo isola as regras responsáveis pelo problema e determina critérios que garantem tal propriedade. Um problema com este método é que ele tende a repetir o processo de teste até que o conjunto seja confluente.

Antes de apresentar o método precisamos apresentar o conceito de *grafos de execução de regras* e a definição de *regras comutativas*.

4.2.2.1 Grafo de Execução de Regras

Um *grafo de execução de regras* é um grafo orientado cujos vértices representam estados do banco de dados e os arcos representam as regras escolhidas para avaliação, conforme apresentado na figura abaixo.

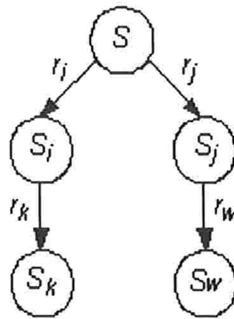


Fig. 4.4 - Exemplo de um grafo de execução de regras.

Mais formalmente, cada estado (vértice) S do grafo tem dois componentes: um estado do banco de dados (representado por D) e um conjunto TR de regras disparadas e suas tabelas de transição associadas. Representamos por S = (D, TR).

Além disso, um caminho de S1 a S2 de tamanho zero ou mais é denotado por:

$$S_1 \xrightarrow{*} S_2$$

Fig. 4.5 – Caminho num grafo de execução de regras

4.2.2.2 Comutatividade entre Regras

Seja um conjunto R de regras. Dadas duas regras r_i e r_j pertencentes a R, ambas sem prioridades e cujas ações não dispararam nenhuma outra regra, dizemos que estas regras são *comutativas* ou *comutam* se a partir de um estado *S* do grafo de execução, considerando r_i e então r_j

produzirá o mesmo estado final S' se considerássemos r_j e, em seguida, r_i; isso é mostrado na figura 4.6. Se essa propriedade não vale então dizemos que as regras são *não-comutáveis* ou *não comutam*.

Antes de prosseguir precisamos definir alguns conjuntos que utilizaremos para apresentar o lema para regras comutativas.

- R = {r1, r2, ..., rn} é o conjunto das regras que serão analisadas;
- P = {ri > rj, rk > rl, ...} estabelece as prioridades entre as regras definidas pelo usuário, onde ri > rj indica que ri tem precedência sobre rj;
- T = {t1, t2, ..., tm} é o conjunto de tabelas do banco de dados;
- O = {<I, t> | t ∈ T} ∪ {<D, t> | t ∈ T} ∪ {<U, t.c> | t.c ∈ C} é o conjunto de operações que modificam o banco de dados; onde <I, t> são as inserções na tabela t, <D, t> exclusões da tabela t e <U, t.c> é a atualização da coluna c da tabela t.

Segundo [1], o lema abaixo é utilizado para verificarmos se duas regras comutam ou não.

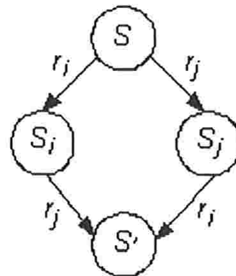


Fig. 4.6 - Regras comutáveis.

Lema 4.1: Sejam duas regras distintas ri e rj e o conjunto O de operações. Se qualquer uma das condições abaixo for satisfeita então ri e rj podem ser não comutativas; caso contrário elas comutam:

- a ação de r_i dispara r_j;
- a ação de r_i desfaz as mudanças que dispararam r_j;

- a ação de r_i altera informações nas colunas utilizadas na avaliação da condição e execução da ação de r_j;
- as inserções da ação de r_i em uma tabela t podem afetar o que a ação de r_j atualiza ou apaga na mesma tabela;
- ambas as ações de r_i e r_j atualizam a mesma coluna na tabela t.

Analisando cuidadosamente este lema, observamos que o mesmo é muito restritivo. Imaginemos que as regras r_i e r_j atualizem a mesma coluna na tabela t, mas nunca na mesma tupla. Assim, pelo lema anterior estas regras podem não comutar, mas sabemos que a atualização feita por r_i não influi na feita por r_j e, portanto, r_i e r_j são comutativas.

Vejamos, informalmente, o método para análise da confluência e como resolver esse inconveniente. Para uma abordagem formal consulte [1].

4.2.2.3 Método para Análise da Confluência

Na figura 4.6, mostramos duas regras confluentes. Entretanto esta não é a situação mais freqüente (lembre-se que ambas as regras não dispararam nenhuma outra); na prática cada uma destas regras pode disparar outras, o que torna a análise do conjunto R mais complexa.

Assim, devemos provar que a partir de um estado S de um grafo de execução GE, a escolha de qualquer uma das regras elegíveis, r_i ou r_j, produz o mesmo estado final S' neste grafo, conforme apresentado na figura abaixo.

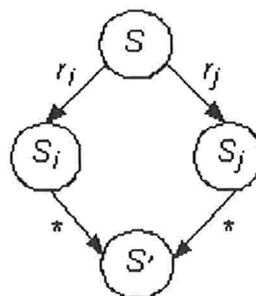


Fig. 4.7 - Confluência de regras.

Note que no caminho entre S_i e S' estamos considerando a regra r_j e todas as regras que possam ser disparadas por r_i e r_j. O mesmo ocorre no caminho entre S_j e S'.

Como r_i e r_j comutam, r_j pode ser avaliada a partir do estado S_i. Entretanto, o que acontece se r_i disparar uma regra r que possua prioridade maior que r_j? Nesse caso, r deve ser processada antes de r_j e ambas devem ser comutativas. Análise análoga é feita a partir do estado S_j. Portanto, estamos construindo um grafo com dois caminhos p_1 e p_2 a partir dos estados S_i e S_j, respectivamente. Neste grafo de execução as regras com prioridade maior que r formam o conjunto R_1 e são consideradas a partir de S_i originando o estado S_i', a partir do qual a regra r_j é processada. Similarmente, as regras com precedência maior que r_i compõe o conjunto R_2 e são consideradas a partir de S_j originando o estado S_j', a partir do qual a regra r_i é executada (figura 4.4).

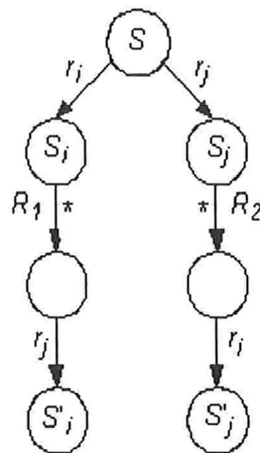


Fig. 4.8 - Grafo de execução representando parte dos caminhos p_1 e p_2.

Continuando o caminho p_1, podemos, a partir de S_i' processar as regras no conjunto R_2. Similarmente, executamos as regras em R_1 a partir do estado S_j' obtendo o estado final S' e, consequentemente, provando a confluência das regras intencionais em r_i e r_j (figura 4.5).

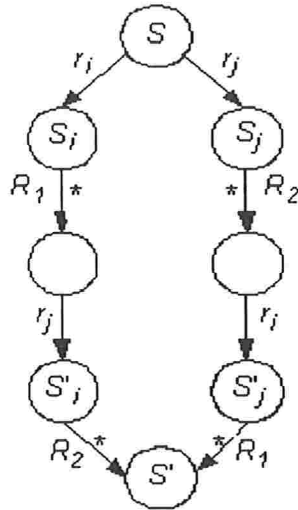


Fig. 4.9 - Grafo de execução cujos caminhos p_1 e p_2 convergem para o mesmo estado final.

Segundo [1], a situação descrita acima não é necessariamente válida, pois não sabemos se há outras regras com prioridade maior que as regras do conjunto R_2 no estado *Si'*. O mesmo problema ocorre com S_j' no conjunto R_1. Para evitar esta situação, os autores propuseram que as regras do conjunto tenham prioridade superior às regras do conjunto R_2 ou vice-versa, originando o seguinte requisito de confluência:

Definição: Considere duas regras r_i e r_j, sem prioridade entre si, pertencentes ao conjunto R. Seja $R_1 \subseteq R$ e $R_2 \subseteq R$ construídos pelo seguinte algoritmo:

$R_1 \leftarrow \{r_i\}$

$R_2 \leftarrow \{r_j\}$

Repita até que os conjuntos R_1 *ou* R_2 *não mudem*

$R_1 \leftarrow R_1 \cup \{r$ *é disparada por* r_1 *para alguma regra* $r_1 \in R_1$ *e*

$r > r_2 \in P$ *para alguma regra* $r_2 \in R_2$ *e* $r \neq r_j\}$

$R_2 \leftarrow R_2 \cup \{r$ *é disparada por* r_2 *para alguma regra* $r_2 \in R_2$ *e*

$r > r_1 \in P$ *para alguma regra* $r_1 \in R_1$ *e* $r \neq r_j\}$

Para todo par de regras $r_1 \in R_1$ *e* $r_2 \in R_2$, r_1 *e* r_2 *devem comutar.*

Assim, o método apresentado em [1] pode ser resumido da seguinte forma:

Se não há caminhos infinitos em nenhum grafo de execução para o conjunto R*, então*

Para cada par de regras r_i *e* r_j *em* R

Construa os conjuntos R_1 *e* R_2

Se as regras em R_1 *comutam com as regras de* R_2 *então*

o conjunto R *é confluente.*

4.2.2.4 Considerações

Gostaríamos de ressaltar dois pontos sobre o método apresentado aqui:

- em primeiro lugar, quando a análise do conjunto *R* determina que o mesmo não é confluente, é fácil isolar as regras que violaram esta propriedade. Freqüentemente, estas regras (r_i e r_j, por exemplo) são aquelas utilizadas para a construção dos conjuntos R_1 e R_2 e não comutam. Uma vez identificadas as regras r_i e r_j, a melhor solução é definirmos uma prioridade entre elas;
- em segundo, o método baseia-se no lema 4.1 que, como vimos anteriormente, é restritivo. A fim de resolver este inconveniente, em [1] é proposta a confluência parcial das regras intencionais pertencentes ao conjunto *R*. Basicamente, a confluência é garantida para as tabelas importantes do banco de dados. É claro que a decisão da importância de uma tabela é subjetiva e está intimamente relacionada ao domínio do sistema.

4.2.3 Conjunto de Ações Consistentes

Consideremos duas regras r_i e r_j, e a tabela *T*(X, Y, Z) pertencente a um determinado banco de dados. Sejam também as ações a_i e a_j correspondente às regras r_i e r_j, respectivamente. Dizemos que estas ações são conflitantes se uma das seguintes condições são verdadeiras:

- a_i = INSERT INTO T VALUES(x, y, z) e a_j = DELETE FROM T WHERE X=x;
- a_i = UPDATE T SET Y = y' WHERE X=x e a_j = UPDATE T SET Y = y" WHERE X=x; e

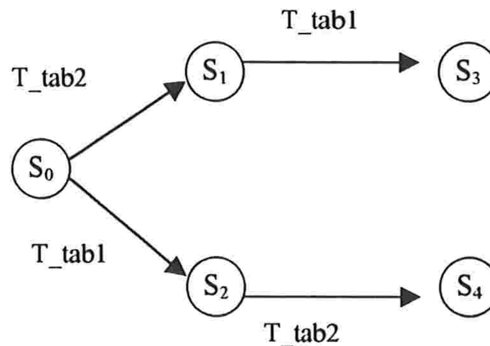
- a_i = UPDATE T SET Y = y' WHERE X=x e a_j = DELETE FROM T WHERE X=x.

Dizemos que um conjunto de ações A = {a_1, a_2, ..., a_n} é consistente se não existirem duas ações a_i e a_j ∈ A conflitantes.

Exemplo:

O conjunto de ações A' = {UPDATE funcionarios SET cargo = 01 WHERE Cod_func = 005, UPDATE funcionarios SET salario = 1000,00 WHERE Cod_func = 005, UPDATE funcionarios SET salario = 2000,00 WHERE Cod_fun = 007} é consistente. Já o conjunto A" = {UPDATE funcionarios SET cargo = 01 WHERE Cod_func = 004, UPDATE funcionarios SET cargo = 02 WHERE Cod_func = 004} não é consistente, pois duas atualizações são feitas sobre o mesmo registro e atributo da tabela funcionarios.

Podemos colocar o exemplo da seção 4.2.1 em um grafo de execução para analisar a confluência. Dessa forma teríamos:



O estado S_0 é o estado inicial do banco de dados, sem o registro inserido. Vamos chamar este registro de *rg* e analisar o que acontece com esse registro nos estados finais S_2 e S_4.

Há dois caminhos a serem analisados no grafo: o caminho gerado pelo disparo de *T_tab1* seguido por *T_tab2* e o caminho gerado por *T_tab2* seguido por *T_tab1*.

a) *T_tab1* seguido por *T_tab2*

O estado S_1 gerado pelo disparo de *T_tab1* apresenta o registro *rg* com valor do campo *preço* igual a 10,00. A seguir o disparo de *T_tab2* leva o banco de dados ao estado S_3 onde o registro *rg* não existe pois foi apagado.

b) *T_tab2* seguido por *T_tab1*

Quando *T_tab2* é disparada o banco de dados chega ao estado S_2 que apresenta o registro *rg* sem modificações. A seguir *T_Tab1* é disparada e leva o banco de dados ao estado S_4 onde o registro *rg* está presente com valor no campo *preço* igual a 10,00.

Vemos que os estados finais S_3 e S_4 são diferentes e, portanto, a confluência não é garantida.

Capítulo 5

SEMÂNTICA CONFLUENTE PARA REGRAS

Neste capítulo vamos apresentar uma nova semântica para o tratamento do problema da confluência que garante, por construção, que as regras são confluentes. Esta nossa semântica nos permitirá o projeto modular de regras em bancos de dados ativos. Os modelos de execução anteriores eliminam os problemas de confluência somente em termos da resolução de conflitos. Quando múltiplas regras são disparadas (e são eleitas para execução), o escalonador de regras seleciona uma regra de acordo com um certo conjunto de critérios tal como a regra disparada há mais tempo, ou que possui as condições mais complexas por exemplo. Embora este esquema seja usado em ambientes de IA, no ambiente de bancos de dados precisamos ter uma execução das transações de maneira mais previsível.

Como já dissemos anteriormente, a origem do problema da confluência de regras é a necessidade de escolhermos, dentre várias regras disparadas, uma que será processada. Nosso modelo propõe a avaliação das condições e execução das ações de todas as regras em "paralelo", evitando assim a escolha de uma regra para execução.

Em nosso algoritmo modificamos, então, o modelo de execução de regras ECA apresentado no capítulo 2 para que ele execute segundo nosso modelo. Vamos a seguir descrever as modificações realizadas.

Vamos repetir aqui o algoritmo 2.1 para discussão. Chamaremos este algoritmo de *algoritmo base.*

enquanto (existem regras disparadas) faça

1. encontre uma regra R *disparada*

2. analise a condição de R

3. se a condição de R *é verdadeira, execute a ação de* R

No algoritmo base a primeira etapa (passo 1) envolve a escolha de uma regra para execução, o que queremos evitar. No nosso algoritmo, o que precisamos determinar não é uma regra, mas todas as regras disparadas por um evento. Então, traduzindo este passo do algoritmo base para o passo 1 do nosso algoritmo temos:

Encontre todas as regras disparadas

Isso não significa, entretanto, todas as regras disparadas no sistema, mas sim aquelas referentes à transação que iniciou o algoritmo. Vamos explicar melhor. Considere a representação gráfica abaixo.

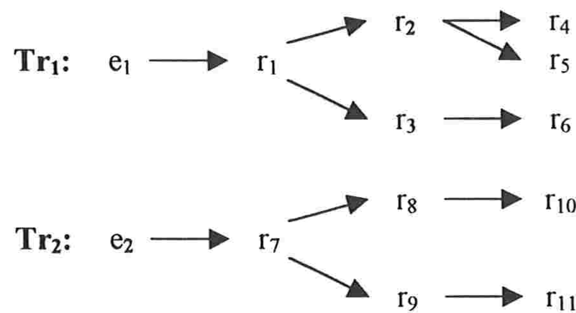


Fig. 5.1 – Relação de regras com transações

Nesta representação, e_1 e e_2 são dois eventos distintos que causaram o disparo das regras r_1 e r_7 respectivamente. Cada um destes eventos iniciou uma transação diferente (Tr_1 e Tr_2 respectivamente). Isso significa que nosso algoritmo é chamado uma vez para cada transação e a cada vez que o passo 1 é executado, ele seleciona todas as regras disparadas dentro da transação que iniciou o algoritmo. Por exemplo, considerando a transação Tr_1, a primeira vez que o passo 1 é executado ele encontra a regra r_1; na segunda vez ele encontra as regras r_2 e r_3 e, na terceira vez, ele encontra as regras r_4, r_5, e r_6.

O segundo passo do algoritmo base é analisar a condição da regra selecionada. No nosso caso, seguindo a proposta de não escolher uma regra, analisamos as condições de todas as regras encontradas no passo anterior. As regras que tiverem suas condições avaliadas como verdadeiras passam para o passo seguinte do algoritmo. Transformando o algoritmo base temos:

Avalie as condições de todas as regras encontradas no passo anterior

O terceiro e último passo do algoritmo base é a execução da ação da regra, caso essa tenha sido avaliada como verdadeira no passo anterior. Em nosso algoritmo, executamos as ações de todas as regras com condições verdadeiras.

Analisando este último passo, é possível que a ação destas regas dispare outras regras, como é o caso no nosso exemplo da transação Tr_1, onde as regras r_4 e r_5 são disparadas pela execução da ação da regra r_2. Entretanto essas novas regras disparadas não são processadas até que todas as regras disparadas atualmente tenham tido suas ações concluídas. Ainda neste exemplo da transação Tr_1, isso quer dizer que não iniciamos o processamento das regras r_4, r_5, ou r_6 até que as ações das regras r_2 e r_3 tenham sido concluídas. Traduzindo esse passo do algoritmo base para o nosso temos:

Execute as ações de todas as regras com condição verdadeira

Há entretanto um novo passo a ser inserido em nosso algoritmo, que não se encontra no algoritmo base, e é o responsável por verificar a existência de ações conflitantes no conjunto das ações eleitas para execução. O conceito de regras conflitantes foi apresentado na seção 4.2.3.

Este passo que estamos introduzindo deve ser feito após todas as regras com condições verdadeiras terem sido encontradas, isto é, logo após o passo 2. Quando uma ação conflitante é encontrada, a transação é abortada. Com isso o estado inicial do banco de dados é preservado, não ocorrendo perda de informação. Como sabemos quais são as ações conflitantes, é possível emitir um aviso ou relatório ao usuário ou programador sobre as regras que causaram o aborto.

O custo introduzido com esse passo é o custo de encontrar ações no conjunto de ações a serem executadas que satisfaçam as condições apresentadas na seção 4.2.3. Esse custo é o custo de comparar cada uma das ações duas a duas - custo polinomial O (n^2), onde n é o número de ações a serem analisadas.

Incluindo então o novo passo e codificando todo o algoritmo, temos o algoritmo descrito na seção seguinte.

5.1 Algoritmo para regras confluentes

Apresentamos aqui o algoritmo iterativo que modifica o modelo de execução de regras ECA, apresentado no capítulo 2, pelo descrito na seção anterior.

Enquanto (existem regras disparadas)

1) *Construa* E*, o conjunto de todas as regras disparadas.*

2) *Avalie as condições de todas as regras do conjunto* E *e construa o conjunto A de todas ações das regras cujas condições são verdadeiras;*

3) *se existem ações conflitantes em A, aborte a transação;*

4) *Execute todas as ações do conjunto A;*

Gostaríamos de ressaltar aqui as duas características que fazem desta semântica um modelo de execução confluente de regras.

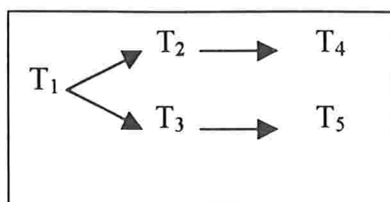
A primeira delas é a execução "paralela" de todas as regras que estavam disparadas em cada passo do algoritmo. Este tipo de execução evita a escolha de uma destas regras para avaliação e, consequentemente, a causa do problema da confluência.

A segunda característica é a verificação da consistência das ações das regras cujas condições foram avaliadas como verdadeiras. Após esta verificação, temos a certeza de que a execução destas ações não são conflitantes e, portanto, sua ordem de execução não interfere no estado final do banco de dados.

Assim, o algoritmo acima implementa uma semântica de execução de regras intencionais confluente.

5.2 Exemplo 1

Considere as triggers T_1, T_2, T_3, T_4 e T_5. Considere que a ação de T_1 dispara T_2 e T_3 e que a ação de T_2 dispara T_4. T_3, por sua vez, dispara T_5. Podemos representar isso, graficamente, da seguinte maneira.



Considere ainda que a execução de T_3 torna a condição de T_2 falsa.

Nas soluções gerais de implementação, supondo que a trigger T_1 tenha sido disparada, as ações da trigger T_1 começam a ser executadas em alguma ordem e podemos ter duas execuções possíveis para o disparo da trigger T_1:

Caso 1: $T_1 \rightarrow T_2 \rightarrow T_4 \rightarrow T_3 \rightarrow T_5$

Caso 2: $T_1 \rightarrow T_3 \rightarrow T_5$

Em nosso algoritmo teremos a seguinte possibilidade de execução:

Caso 1: $T_1 \rightarrow \begin{matrix} T_2 \\ \text{e} \\ T_3 \end{matrix} \rightarrow \begin{matrix} T_4 \\ \text{e} \\ T_5 \end{matrix}$

Isto é, mesmo que T_3 invalide o teste da condição de T_2, ela já havia sido eleita para execução antes da ação de T_3 ser executada, e portanto T_2 é executada.

5.3 Exemplo 2

Vamos agora exemplificar de maneira prática a utilização do algoritmo.

Considere que temos uma empresa que mantém um banco de dados de seu estoque de peças. Vamos utilizar três tabelas para nosso exemplo: uma tabela chamada pecas (que mantém informações sobre as peças em estoque), uma tabela estoque (que mantém informação sobre a quantidade disponível em estoque de uma certa peça), e ,finalmente, a tabela pedidos (que armazena informações sobre a quantidade de peças que deve ser comprada). Abaixo descrevemos os campos destas tabelas.

- peças (codigo, nome, valor, qtdemin)

  onde: *codigo:* código da peça

  nome: nome descritivo da peça

  valor: valor da peça

  qtdemin: quantidade mínima que deve ser mantida em estoque

- estoque (codpeca, qtde)

  onde: *codpeça:* código da peça

  qtde: quantidade disponível em estoque

- pedidos (codpeca, qtde)

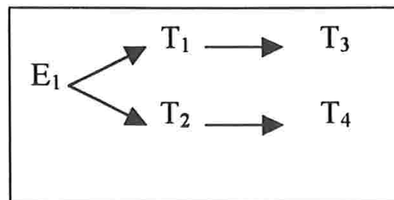
  onde: *codpeca:* código da peça

  qtde: quantidade da peça a ser pedida

Considere as seguintes regras que são aplicadas sobre as tabelas acima:

- Quando uma nova peça é inserida na tabela pecas, essa peça deve ser inserida pelo sistema na tabela estoque com quantidade em estoque igual a zero. Vamos chamar esta regra de T_1.
- Quando uma nova peça é inserida na tabela pecas, essa peça deve ser inserida pelo sistema na tabela pedidos com quantidade a ser pedida igual a quantidade mínima que deve ser mantida em estoque da peça. Vamos chamar esta regra de T_2.
- Quando uma nova peça é inserida na tabela de estoque, deve ser calculado um desconto de 10% sobre o preço da peça, por esta ser nova no conjunto de peças da empresa (essa é uma estratégia de vendas da empresa). Vamos chamar esta regra de T_3.
- Quando um novo pedido é inserido na tabela de pedidos, deve ser emitido um relatório com o nome da peça e a quantidade a ser pedida, para toda peça com valor maior ou igual a 10,00. Vamos chamar esta regra de T_4.

Considerando o evento "inserir uma nova peça" como sendo E_1, temos uma representação gráfica semelhante à que descrevemos no exemplo 1 para os disparos das regras que acabamos de propor, ou seja:



Entretanto, neste caso, a execução de T_3 pode tornar a condição de T_4 falsa (e não T_1 com T_2, como no exemplo anterior).

Utilizando o algoritmo 2.1 de processamento de regras ECA apresentado inicialmente no capítulo 2, temos uma situação onde a ação da regra T_4 é executada antes da ação da regra T_3 e, portanto, o relatório é emitido, e uma outra situação onde o relatório não é emitido pois a ação de T_4 é executada após a ação de T_3.

Por outro lado, utilizando o nosso algoritmo sempre temos o relatório gerado, pois quando as condições são testadas, nenhuma ação foi ainda executada e, portanto, a ação de T_3 não torna falsa a condição de T_4.

Para concluir vamos representar a execução do exemplo dado na seção 4.2.1 segundo nosso algoritmo.

1) inserção do registro (1, "agulha", 20.00)

2) *T_peças* é disparada e insere os valores 70 e 80 nas tabelas *tab1* e *tab2* respectivamente

3) *T_tab1* e *T_tab2* são disparadas

4) As condições de *T_tab1* e *T_tab2* são avaliadas, mas somente a condição de *T_tab1* é verdadeira.

5) Executa a ação de *T_tab1*

6) O banco de dados após a execução possui o registro inserido com valor do campo *preço* igual a 10.00.

Dessa forma, nosso modelo eliminou o problema de confluência, pois chegamos a um único estado do banco de dados após a execução.

Capítulo 6

ARQUITETURA PARA IMPLEMENTAÇÃO DA SEMÂNTICA CONFLUENTE

No capítulo 1 dissemos que os termos *regras* e *triggers* seriam utilizados indistintamente para se referir a *regras*. Neste capítulo entretanto vamos distinguir estes termos. Quando utilizamos o termo *regras* estamos tratando de regras de bancos de dados ativos e quando utilizamos *triggers* estamos tratando de triggers do SQL Server utilizadas para implementar nossa semântica. As regras devem ser executadas segundo nossa semântica, mas as triggers são executadas segundo o algoritmo base.

Para implementar nosso algoritmo escolhemos o banco de dados SQL Server da Microsoft. Em nossa escolha levamos em consideração que se fosse possível implementar nosso modelo no SQL Server, que é mais restrito em seus recursos quanto a triggers (veja quadro comparativo apresentado na seção 3.5), também seria possível implementar em um sistema mais robusto. Na implementação procuramos utilizar recursos do próprio sistema, sem interferir com ferramentas externas para não embutir sobrecarga extra. Para tanto foi necessário criar algumas tabelas auxiliares.

É preciso salientar que neste caso a implementação fica, altamente dependente da plataforma escolhida, podendo haver diferenças significativas na implementação em outra plataforma.

Foram assumidas algumas simplificações para que pudéssemos tratar o problema e também imaginamos o controle de transações e tabelas temporárias semelhante ao que ocorre no SQL Server. Abaixo descrevemos essas características.

- Cada transação que se inicia no sistema faz uma chamada, ou inicia nosso algoritmo e esse algoritmo atua somente nas regras que estão sendo disparadas dentro desta transação, a partir do evento que iniciou o disparo da primeira regra.
- Quando um evento que dispara uma regra acontece, ele inicia uma transação. No inicio desta transação todo o ambiente necessário para a execução do nosso algoritmo deve ser

preparado, isto é, todas as tabelas auxiliares que serão utilizadas dentro desta transação devem ser inicializadas. Cada uma das tabelas auxiliares somente pode ser acessada de dentro desta transação, o que quer dizer que nenhuma outra operação ou evento que não esteja na transação pode "enxergar" estas tabelas. Se um outro evento que dispara regras ocorrer no sistema, ele inicia uma outra transação e tem suas próprias tabelas auxiliares.

- Consideramos somente os eventos Insert, Update e Delete como eventos que disparam regras e também são as únicas ações que podem ser realizadas por uma regra.

Nosso algoritmo deve executar as regras de maneira "paralela" para garantir a confluência de acordo com a semântica dada no capítulo 5. Para isso é necessário intervir na maneira como o SQL Server executa o processamento das regras e imprimir este novo comportamento na avaliação. Como não temos acesso ao código fonte do SQL Server, formulamos uma nova maneira de escrever as regras e as submetemos ao nosso "processador de disparos (PD)" antes de permitir que o SQL Server tome controle da execução.

Em vez de o usuário escrever suas regras diretamente para o SQL Server, ele as escreve como um Insert para uma tabela chamada *acoes*. Esse Insert terá a seguinte sintaxe:

Insert acoes values (<comando>, <onde>, <complemento>, <condição>, <nome>)

onde:

<comando>: vamos ter 3 comandos de triggers possíveis: Insert, Delete e Update

<onde> : nome da tabela sobre a qual o comando vai ser executado

<complemento> : complemento do comando. Por exemplo "values" no caso do comando Insert

<condição> : condição para que o comando seja executado. É possível que seja NULL, isto é, que não haja nenhuma condição a ser avaliada

<nome> : nome único que identifica a regra que está sendo criada.

Exemplo:

Considere a regra descrita a seguir.

```
create trigger tins on dbo.tab1
for insert
as
Update pecas
Set preco = 10
Where id = 1
go
```

Para nosso algoritmo, ela passa a ser escrita como:

```
create trigger tins on dbo.tab1
for insert
as
insert acoes values("update", "pecas", "set preco = 10", "id = 1", "tins")
go
```

Graficamente podemos representar nossa intervenção no modo como as regras são processadas na seguinte forma:

Fig. 6.1 – modelo base

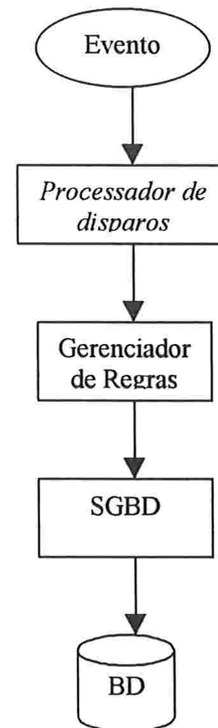


Fig. 6.2 – nosso modelo

Inserimos uma nova "camada" no modelo para interferir na maneira como as regras são executadas (isto é, para conseguirmos simular a execução "paralela"). Nosso *Processador de Disparos (PD)* "segura" as regras disparadas por um evento até que todas elas, e não só algumas, estejam prontas para serem processadas (isso inclui todas as regras disparadas pela execução das ações das regras disparadas no ciclo anterior do algoritmo). Só então são analisadas todas as condições (passo 3) e executa todas as ações correspondentes (passo 3). Vamos detalhar a seguir como funciona nosso *Processador de Disparos*.

6.1 O Processador de Disparos

Vamos inicialmente descrever os passos mais importantes do processo. Não vamos nos deter, por hora, aos detalhes de implementação nem falar sobre as inicializações que são necessárias em cada ciclo do algoritmo.

O diagrama 6.3 representa o PD. Acompanhe o diagrama à medida que descrevemos nas seções a seguir como é implementado cada passo do algoritmo apresentado no capítulo 5.

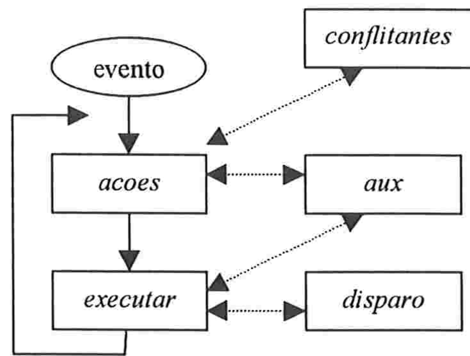


Fig. 6.3 Diagrama do Processador de Disparos

6.1.1 Passos 1 e 2

Criamos uma tabela chamada *acoes* e uma trigger associada a ela. Essa trigger é disparada cada vez que um novo Insert é feito nesta tabela e dessa forma ganhamos então controle sobre como e quando as ações das regras serão executadas.

A primeira tarefa desta trigger é verificar se todas as regras disparadas no ciclo anterior do algoritmo já foram inseridas na tabela *acoes* (discutiremos como esse número é obtido mais adiante). Essa tarefa de verificação corresponde ao controle que desejamos obter para que todas as condições das regras disparadas sejam avaliadas em "paralelo", como se fossem feitas ao mesmo tempo (passo 1 do nosso algoritmo). Se todas as regras ainda não foram inseridas então o PD deve continuar aguardando (isso significa que a trigger termina).

Quando todas as regras já tiverem sido inseridas o próximo passo é verificar as regras que possuem condição verdadeira através do procedimento *TestaCondicao*. As condições de todas as regras são avaliadas e aquelas que tiverem suas condições avaliadas como verdadeiras são copiadas da tabela *acoes* para a tabela *executar* (isso corresponde ao passo 2). E finalmente, após o procedimento *TestaCondicao* ter avaliado todas as condições (e movido todas as triggers com condição verdadeira para a tabela executar), ativa o procedimento *Proc_Exec*.

O procedimento *TestaCondicao* mencionado acima, realiza o teste das condições fazendo uma consulta ao banco de dados utilizando como condição para esta consulta as condições das regras. Se o resultado da consulta retornar algum registro, então a condição é verdadeira, mas se a consulta for vazia, a condição é falsa.

6.1.2 Passos 3 e 4

Antes de iniciar a descrição deste passo, cabe uma observação sobre o motivo pelo qual os passo 3 e 4 estão separados dos demais passos. A razão para este fato já foi mencionada anteriormente na seção 3.4 quando descrevemos o SQL Server: uma trigger não pode disparar a si mesma, ou seja, o comportamento recursivo não é permitido. Como a cada ciclo do algoritmo (que corresponde a uma execução das ações da trigger *T_acao*) novas ações podem ser colocadas na tabela de triggers disparadas (*acoes*) e, obviamente, como queremos que a trigger *T_acao* seja disparada, precisamos utilizar de um artifício que foi um procedimento responsável pela execução das ações e a inserção na tabela *acoes* das próximas triggers disparadas.

Continuando a descrição da execução do PD, o próximo passo inicia com a chamada do procedimento *Proc_Exec* sobre a tabela *executar*, que mencionamos anteriormente.

A tabela *executar* tem a estrutura semelhante à tabela *acoes*, mas não tem nenhuma trigger associada a ela. As regras que estão nessa tabela são aquelas que devem ter suas ações executadas. Essa execução será iniciada ao final da execução da trigger *On Insert* da tabela *acoes* através da chamada ao procedimento *Proc_Exec*.

O procedimento *Proc_Exec*, responsável pela execução das ações das regras, a exemplo da trigger da tabela *acoes*, só executa suas tarefas quando todas as regras que terão suas ações executadas tiverem sido movidos para a tabela *executar*. A primeira tarefa deste procedimento na realidade implementa o passo 3 do nosso algoritmo, fazendo a verificação da existência de ações conflitantes. Caso sejam encontradas ações conflitantes a transação é abortada, caso contrário o procedimento executa cada uma das ações das regras da tabela *executar*.

Uma vez que passamos pelos pontos principais do algoritmo, resta explicar como determinamos o número de triggers disparadas e o número e ações que devem ser executadas no próximo ciclo do algoritmo. Vamos detalhar essas duas necessidades.

6.1.3 Calculo de número de regras disparadas

Em cada ciclo do algoritmo é necessário conhecer o número de regras que foram disparadas para que a análise das condições seja feita em "paralelo", como propõe nosso algoritmo. Ou seja, a

trigger da tabela *acoes* só inicia os testes de condição das regras depois que todas as regras foram inseridas na tabela *acoes*.

Para avaliar as condições das regras utilizamos a tabela auxiliar *aux*. Essa tabela mantém dados sobre o número de regras que serão disparadas no próximo ciclo do algoritmo e o número de regras disparadas que tiveram suas condições avaliadas como verdadeiras (na realidade utilizamos apenas um registro nesta tabela com estes dois valores). Vamos mostrar aqui como é calculado o número de regras disparadas para o próximo ciclo. O cálculo do número de regras que tiveram suas condições avaliadas como verdadeiras será mostrado na seção seguinte.

O número de regras que serão disparadas no próximo ciclo é o número de regras que serão disparadas pela execução das ações das regras no ciclo atual. Esse número é calculado pelo procedimento *Proc_exec* a partir de uma tabela chamada *disparos* que mantém uma relação das ações que provocam o disparo de regras dentro do banco de dados. Por exemplo, suponha que uma inserção em uma tabela chamada *peças* provoque o disparo de uma regras chamada *Tins_peças*. Então deve existir um registro na tabela *disparos* que contenha a ação *inserte* na tabela *peças*. Dessa maneira, é possível, antes de executar as ações das regras, fazer uma consulta na tabela disparos verificando quantas ações das regras atuais irão disparar regras no próximo ciclo.

6.1.4 Calculo de número de ações a serem executadas

Da mesma forma que calculamos o número de regras que foram disparadas para que a análise das condições seja feita em "paralelo", também precisamos calcular o número de regras que terão suas ações executadas para que estas ações sejam feitas em "paralelo". Ou seja, o procedimento de execução das ações (*Proc_exec*) só inicia a execução das ações quando todas as regras tiverem sido colocadas na tabela *executar*.

O número de ações das regras que serão executadas é obtido por meio do procedimento *Testa_Condição*, que verifica se a condição da regra é verdadeira através de uma consulta ao banco de dados aplicando a consulta da regra. Por exemplo, suponha que a condição de uma regra seja verificar que uma certa peça tenha quantidade em estoque inferior a 10 unidades. O procedimento *Testa_Condição* executa um *select* no banco de dados com esta condição. Se a consulta retornar algum registro então a condição é verdadeira, caso contrário, ela é falsa. Isso é feito regra por regra e

aquelas que tiverem suas condições avaliadas como verdadeiras, são inseridas na tabela *executar* para terem suas ações executadas no próximo passo do algoritmo.

Antes de terminar este assunto precisamos lembrar que os valores constantes na tabela *aux* no início do primeiro ciclo deve ter sido corretamente iniciado. Como dissemos no início do capítulo, assumimos alguns fatos na implementação do algoritmo e um deles é que no início da transação, as tabelas temporárias e auxiliares são devidamente iniciadas. Então no início do algoritmo (início da transação) a tabela *aux* deve ter sido inicializada com valor 1 para o número de regras disparadas e 0 para o número de ações a executar.

6.1.5 Descrição das tabelas e triggers

A seguir estamos descrevendo mais detalhadamente as tabelas que foram necessárias para executar a implementação do nosso algoritmo.

- *acoes*: tabela das regras disparadas no ciclo atual do nosso algoritmo. Contém os seguintes campos:

  1. *acao*: ação a ser executada pela regra (Insert, Delete, Update)

  2. *onde*: tabela onde a ação será aplicada

  3. *comple*: complemento a ser utilizado na ação (por exemplo um valor a ser utilizado num update)

  4. *condicao*: condição para que a ação da regra seja executada

  5. *nome*: nome da regra.

- *executar*: tabela de todas as ações a serem executadas no passo 4 do nosso algoritmo. Estas são ações das regras disparadas no sistema e que tiveram suas condições avaliadas como verdadeiras. Contém os seguintes campos:

  1. *acao*: ação a ser executada pela regra (Insert, Delete, Update)

  2. *onde*: tabela onde a ação será aplicada

3. *comple*: complemento a ser utilizado na ação (por exemplo um valor a ser utilizado num update)

4. *condicao*: condição para que a ação da trigger seja executada

- *aux* : Tabela auxiliar para a implementação do algoritmo. Seu objetivo é controlar o número de regras que serão avaliadas/executadas no próximo ciclo do algoritmo que será iniciado. Contém os seguintes campos:

  1. *acoes*: número de regras que serão disparadas (no próximo ciclo) pela execução das ações das regras atualmente sendo avaliadas (no ciclo atual do algoritmo).

  2. *executar*: número de regras que tiveram suas condições avaliadas como verdadeiras e terão portanto suas ações executadas.

- *Disparo*: Tabela que mantém as relações das regras que serão disparadas por um evento em uma certa tabela. Contém os campos:

  1. *evento*: o evento (Insert, Update, Delete) que dispara uma regra associada à tabela descrita no campo a seguir.

  2. *sobre*: tabela sobre a qual, se o evento do item anterior ocorrer, dispara uma regra

Triggers associadas às tabelas auxiliares:

- *On Insert para a tabela acoes*: quando é inserida uma nova ação a ser executada, esta trigger verifica se já foram inseridas todas as ações das regras que foram disparadas no passo anterior (ou o número estipulado de regras na primeira execução) e em caso afirmativo verifica quantas e quais destas ações devem ser executadas. Ela atualiza na tabela *aux* a quantidade de ações a serem executadas, e, na tabela *Executar*, ela insere as ações que devem ser executadas. A seguir ativa o procedimento que é responsável pela execução das ações.

Procedimentos:

- *Proc_Exec*: este procedimento é responsável pela execução das ações das regras que tiveram suas condições avaliadas como verdadeiras no ciclo atual do algoritmo. Ele verifica quantas regras serão disparadas pela execução das ações que ele irá executar e atualiza a tabela *aux* para o próximo ciclo do algoritmo.
- *TestaCondicao*: Este procedimento é utilizado pela trigger da tabela *acoes* para verificar se a condição de uma regra disparada é verdadeira ou não.

6.2 Custo

Para analisar o impacto que nosso modelo teria quando implementado sobre um sistema de banco de dados ativos, vamos analisá-lo comparando cada passo do algoritmo proposto com relação ao algoritmo básico de regras ECA. Também vamos estimar o custo extra com o espaço de armazenamento.

6.2.1 Desempenho

Vamos iniciar nossa análise pensando nos passos 2 e 4 do nosso algoritmo. Ambos são equivalentes aos passos 2 e 3 do algoritmo base respectivamente. Não há nenhum acréscimo de gasto de tempo nestes dois passos pois a única diferença é que toda regra em nosso algoritmo é executada simultaneamente enquanto que no algoritmo base elas são executadas uma de cada vez.

No passo 1 do nosso algoritmo temos um gasto de tempo extra, já que no algoritmo base uma regra disparada entra diretamente no processamento sem nenhum teste adicional, enquanto que no nosso algoritmo é necessário verificar se todas as regras do ciclo anterior já tiveram suas execuções completadas. Ou seja, cada regra disparada exige um teste a mais para descobrir se o processamento pode ser iniciado. Esse tempo extra cresce linearmente com número de regras disparas num ciclo do algoritmo, o que, em um ambiente onde o número de regras não é demasiado grande, não causa um impacto significativo na degradação da performance.

E, finalmente, o passo 3 do nosso algoritmo que, por não fazer parte do algoritmo base, gera um gasto de tempo extra. O tempo gasto aqui é com uma consulta feita em uma tabela para descobrir se existem regras disparadas que sejam conflitantes. O preço pago por esta consulta depende do número de regras conflitantes que existem no sistema e que podem estar disparadas

dentro de uma mesma transação num mesmo ciclo do algoritmo ($O(n^2)$, onde *n* é o número de regras disparadas). No pior caso temos nosso algoritmo disparado n vezes (cada regra disparar outra) o que nos dá uma complexidade de pior caso $O(n^3)$. Na prática, porém, espera-se uma complexidade menor pois em problemas reais, raramente teremos um conjunto de regras onde todas sejam disparadas e cada uma dispare outra.

Dessa forma nosso modelo não causa grandes prejuízos com relação ao tempo extra gasto no processamento das regras, exceto para ambientes particulares onde o número de regras seja muito grande.

6.2.2 Espaço

Quanto ao espaço extra consumido pelo nosso modelo, temos três tabelas extras utilizadas em nossa implementação que são relevantes. Uma delas é a tabela que armazena os nomes das triggers que são conflitantes (chamada *disparo* em nossa implementação). A quantidade de espaço necessária aqui não é significante.

As outras duas tabelas utilizadas são as que armazenam as triggers disparadas e que estão em processamento. A primeira (chamada *acoes* em nossa implementação) armazena as triggers que terão suas condições avaliadas. A segunda tabela (chamada *executar* em nossa implementação) tem tamanho igual ou menor que a tabela *acoes* pois contem as triggers da tabela *acoes* que foram avaliadas como verdadeiras. Ambas as tabelas tem tamanho proporcional ao número de triggers disparadas no sistema. Podemos dizer que, em geral, seu impacto sobre o espaço total utilizado no sistema não é grande, à exceção de sistemas específicos onde possam existir grande número de triggers disparas simultaneamente.

6.3 Exemplo

Vamos descrever aqui como fica o exemplo visto em 3.3.2. Vamos escrever as triggers vistas da seguinte forma:

1. Trigger da tabela peças:

insert acoes values("Insert","tab1","values(70)",NULL,"TP1")

insert acoes values("Insert","tab2","values(80)",NULL,"TP2")

2. Trigger da tabela tab1:

insert acoes values("update", "pecas", "set preco = 10", "id = 1","T1")

3. Trigger da tabela tab2:

Insert acoes values("delete", "pecas", NULL, "preco = 10", "T2")

Quando o registro (1, "agulha", 20,00) é inserido na tabela de peças, a trigger descrita no item 1 acima é disparada e realiza uma inserção na tabela de ações a serem executadas. Quando as duas ações são inseridas, é chamado o procedimento para avaliar as condições das triggers. Neste caso somente a trigger da tabela tab1 é verdadeira e portanto somente esta trigger é que será executada. Sendo assim nosso banco de dados sempre terá o mesmo resultado final.

Capítulo 7

CONCLUSÃO

O principal objetivo desta dissertação foi a apresentação de um novo modelo de execução das regras ECA para garantir a propriedade de confluência. Para tal, modificamos o algoritmo básico de processamento destas regras.

Para atingir este objetivo realizamos um amplo estudo sobre os sistemas de bancos de dados ativos verificando sua importância e impacto em determinados tipos de ambientes, como gerenciamento de restrições de integridade complexos e implementação de políticas de gestão empresarial específicas a estes ambientes. Também estudamos algumas implementações de regras em bancos de dados comerciais. Constatamos ao longo do estudo que existem ainda problemas na utilização dos bancos de dados ativos devido à falta de ferramentas para desenvolvimento, análise e manutenção das regras.

Um dos maiores problemas encontrados, e que foi o principal enfoque deste trabalho, foi o problema de garantir a propriedade da confluência, cuja origem é a necessidade de escolher uma dentre várias regras disparadas simultaneamente para execução. Para resolvermos o problema propusemos a execução "paralela" de todas as regras disparadas.

Como resultado de nosso trabalho chegamos a uma implementação do nosso modelo utilizando o banco de dados comercial SQL Server. Durante a implementação encontramos diversas dificuldades entre elas a falta de uma documentação que descreva a forma de processamento das regras ou o seu relacionamento com as transações. Para descobrir como esse processamento ocorre realizamos vários testes submetendo diferentes regras para processamento em diferentes estados da base de dados.

Nosso modelo foi implementado como uma "camada acoplada" sobre o SQL Server modificando a forma de tratamento das regras para refletir o novo comportamento proposto por nosso modelo. Como não tínhamos acesso direto ao código fonte do SQL Server, utilizamos

triggers e tabelas auxiliares para implementar nosso modelo. O controle sobre como o processamento ocorre foi ganho fazendo com que as regras fossem escritas como uma inserção em uma das tabelas que criamos ao invés de escrevê-la diretamente para o processamento do banco de dados. A partir desse ponto pudemos controlar o momento em que a regra deve ser disparada e quando suas ações devem executadas.

Analisando os resultados da implementação, chegamos à conclusão de que é possível realizar a implementação em um sistema de banco de dados ativo sem que haja demasiada degradação da performance do sistema. Também como resultado de garantir a confluência, podemos conseguir a modularização das regras através de Tabelas de Decisão, como demonstrado no trabalho de [11].

Na implementação que fizemos, alguns pontos foram supostos (como geração das tabelas temporárias e a execução de apenas uma transação no sistema) e que poderiam ser implementados futuramente. Em trabalhos futuros é possível, por exemplo, planejar para que o próprio sistema determine quando uma regra seja conflitante com uma outra sem que haja necessidade de o projetista colocar esta informação em uma tabela.

CODIGO FONTE DA IMPLEMENTAÇÃO

1) Trigger para a tabela ações

```
CREATE TRIGGER T_acao ON dbo.acoes
FOR INSERT
AS
Declare @Qtde int, @quant int, @verdade int, @testar int, @temp char(20)
Declare @acao varchar (15), @onde varchar(20), @comple varchar(20), @nome char(3)
Declare @condicao varchar(40), @aux varchar(10), @fim int
Declare cursor_acao SCROLL CURSOR for Select acao, onde, comple, condicao,nome from acoes
OPEN cursor_acao

/* Le o numero de acoes a realizar */
Select @verdade = 0 /*executar From aux*/
/*as execucoes comecam sempre em zero*/
update aux set executar = 0

Select @testar = acoes From aux
Select @fim = 0

/* verifica se ja inseriu todas as acoes na tabela de acoes */
Select @Qtde = Count(*) From acoes
if @Qtde = @testar
  begin

    /* inicializa apagando as acoes ja executadas*/
    delete from executar

    /*posiciona o cursor na proxima tupla */
    FETCH NEXT FROM cursor_acao INTO @acao, @onde, @comple, @condicao, @nome
    While (@@FETCH_STATUS <> -1)
     begin
        /*verifica se existe alguma condicao */
        if @condicao <> NULL
         begin
          Exec TestaCondicao @onde, @condicao
          select @quant = verdade from a
          select @aux = str (@quant)
          if @quant > 0
           begin
                select @verdade = @verdade + 1
                update aux set executar = @verdade
                insert executar values (@acao,@onde,@comple,@condicao)
           end
         end
        /* Nao ha condicao a ser testada*/
        else
```

```
            begin
              select @verdade = @verdade + 1
              update aux set executar = @verdade
              insert executar values (@acao,@onde,@comple,@condicao)
            end
          FETCH NEXT from cursor_acao INTO @acao, @onde, @comple, @condicao, @nome
    end
    select @fim = 1
end
 close cursor_acao
 DEALLOCATE cursor_acao
 if(@verdade > 0) and (@fim = 1) Exec Proc_Exec
GO
```

2) Trigger para a tabela peças

```
CREATE TRIGGER Tinst ON dbo.pecas
FOR INSERT
AS
insert acoes values ("Insert", "tab1", "values(70)", NULL,"T1")
insert acoes values ("Insert", "tab2", "values(80)", NULL,"T2")
GO
```

3) Trigger para a tabela tab1

```
CREATE TRIGGER Tins ON dbo.tab1
FOR INSERT
AS
insert acoes values("update","pecas","set preco = 10","int=1","T3")
GO
```

4) Trigger para a tabela tab2

```
CREATE TRIGGER T2inst ON dbo.tab2
FOR INSERT
AS
insert acoes values ("delete","pecas",NULL,"preco = 10","T4")
GO
```

5) Procedimento Proc_Exec

```
CREATE PROCEDURE Proc_Exec AS
Declare @Qtde int, @cont int
Declare @acao varchar(15),@onde varchar(20), @comple varchar (20)
Declare @condicao varchar(40), @fazer varchar(100)
Declare cursor_exec SCROLL CURSOR for select acao, onde, comple, condicao from executar
open cursor_exec
select @cont = executar from aux
select @Qtde = Count(*) from executar
select @acao = str(@cont)

/*verifica se ja inseriu todas as acoes a serem executadas */
```

```
if @Qtde = @cont
 begin
    update aux set acoes = 0

/* verifica se existem triggers conflitantes */
        if(select count(nome) from acoes
          where nome in
          (select c.T1 from conflitantes c
           where c.T2 in (select nome from acoes)) )> 0
          ROLLBACK TRANSACTION

  /*apaga todas as entradas da tabela acoes*/
  delete from acoes

  /* verificar o numero de triggers que serao disparadas no prox. loop */
  FETCH FIRST from cursor_exec into @acao,@onde,@comple,@condicao
  while (@@FETCH_STATUS <> -1)
  begin
   select @Qtde = (select count(*) from disparo where evento = @acao and sobre = @onde)
   if @Qtde > 0
        update aux set acoes = acoes + @Qtde
   FETCH NEXT from cursor_exec into @acao,@onde,@comple,@condicao
  end

  /* executa acoes */
  FETCH FIRST from cursor_exec into @acao,@onde,@comple,@condicao
  while (@@FETCH_STATUS <> -1)
   begin
    select @fazer = @acao + " " + @onde
    if @comple <> NULL
        select @fazer = @fazer + " " + @comple
    if @condicao <> NULL
        select @fazer = @fazer + " where " + @condicao
    FETCH NEXT from cursor_exec into @acao,@onde,@comple,@condicao
    select @cont = @cont -1
    if @cont = 0
        begin
         CLOSE          cursor_exec
         DEALLOCATE cursor_exec
        end
    EXEC (@fazer)
   end
  end
 GO
```

6) Procedimento TestaCondição

```
CREATE PROCEDURE TestaCondicao @onde varchar(20),@condicao varchar(40) AS
Declare @t varchar(150)
Select @t = "if ((Select count(*) from " + @onde + " where " + @condicao + ") >0) update a set verdade =1 else
update a set verdade =0"
Exec (@t)
GO
```

Bibliografia

[1] A. Aiken, J. Widom, e J. M. Hellerstein. Behavior of database production rules: termination, confluence, and observable determinism. ACM SIGMOD págs. 59-68, 1992

[2] ACT-NET Consortium. The Active Database Management System manifesto: a rulebase of ADBMS features. SIGMOD Record, vol. 25, N.º 3, setembro/1996

[3] S. K. Kim, S. Chakravarthy. A Confluent Rule Execution Model for Active Databases. Tech. Report, University of Florida, Octuber 1995

[4] Xianchang Wang, Jia-Huai You, Li Yan Yuan. On Clonfluence Property of Active Databases with Meta-Rules. Departament of Computing Science, University of Alberta Edmonton, Canada.

[5] Jennifer Widom. Research Issues in Active Database Systems: Report from the Closing Panel at RIDE-ADS' 94

[6] S. Chakravarthy. A Comparative Evaluation of Active Relational Databases. Tech. Report, University of Florida, Janeiro 1993

[7] ACT-NET Consortium. The Active Databasd Management System Manifesto: A Rulebase of ADBMS Features. SIGMOD Record, vol. 25, nº3, Setember 1996

[8] J. Widow, and S. Ceri. Active Database Systems: Triggers and Rules for Advanced Database Processing. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[9] M.H. van der Voort, A. P. J. M. Siebes. Termination and confluence of rules execution. Centrum voor Wiskunde en Informatica (CWI) – ReportRapport.1993

[10] H. F. Korth, and A. Silberschatz. Sistema de Banco de Dados. Makron Books, 2ª Edição, São Paulo, 1993.

[11] M. O. Furtado. Projeto de regras confluentes para bancos de dados ativos usando tabelas de decisão. Dissertação de Mestrado, IME-USP, São Paulo, 1999.