PROL: um PROcessador Lisp para consultas recursivas

Rosana de Saldanha da Gama Lanzelotte *
Alexandre Ribenboim **
PUC-RIO - Departamento de Informática
Email: usersana@LNCC.bitnet

SUMARIO

Os novos sistemas de bancos de dados dedutivos ou orientados a objetos necessitam de recursos para a escrita de visões e consultas recursivas. Os construtores de objetos requerem, além disso, a capacidade de expressar consultas que envolvam funções de agregação. O presente trabalho introduz um formalismo genérico, RIO/ROO, e um processador, PROL, para consultas que envolvam recursividade linear e agregação. RIO/ROO são usadas como representações intermediárias de um otimizador de consultas extensível [LANZELOTTE90b], e possibilitam a formulação de consultas requeridas em grande parte dos sistemas de bancos de dados dedutivos ou orientados a objetos.

ABSTRACT

The new deductive and object oriented database systems require the possibility of writing recursive views and queries. Besides, object oriented systems also require aggregation. Here are introduced a generic formalism, RIO/ROO, and a processor, PROL, for queries involving recursion and aggregation. RIO/ROO are used as intermediate representations for an extensible query optimizer [LANZELOTTE90b] and enable the writing of the queries needed in most deductive and object oriented systems.

* Mestre em Ciência da Computação (PUC-Rio, 1977), cursando Doutorado na mesma universidade. Áreas de interesse: bancos de dados dedutivos, implementação de SGBD. Departamento de Informática, PUC-Rio, R. Marquês de S. Vicente, 225 - Rio de Janeiro - CEP 22453.

** Cursando Engenharia de Computação (PUC-Rio). Áreas de interesse: bancos de dados, computação gráfica. Departamento de Informática, PUC-Rio, R. Marquês de S. Vicente, 225 - Rio de Janeiro - CEP 22453.

1. Introdução

A representação de visões e consultas recursivas é um novo requisito em sistemas de bancos dedutivos ou orientados a objetos. Um problema frequentemente associado a consultas desse tipo consiste daquelas que calculam o fecho transitivo de uma relação, de aplicação prática evidente. Outro exemplo que exige a recursividade ocorre na construção de objetos com alguma parte recursiva.

Embora o tratamento genérico da recursividade seja um problema de difícil abordagem, alguns casos simples e bem conhecidos, que são os de maior aplicação prática, podem ser eficientemente resolvidos. Nesse sentido, existem várias propostas para a avaliação de consultas recursivas [BANCILHON86] [LANZELOTTE88a]. Em geral, tais propostas se aplicam à linguagem DATALOG sem extensões, cuja sintaxe é semelhante à de PROLOG, mas cuja semântica é sobretudo declarativa, e não procedimental, como é o caso em PROLOG [ULLMANN85].

Algumas extensões a DATALOG tem sido propostas no sentido de permitir a construção de objetos estruturados, o que efetivamente contribui para aumentar a sua expressividade semântica [BEERI87] [KUPER87]. Exemplos dessas extensões são construtores de tuplas, conjuntos e listas. Também foram propostas extensões no sentido de permitir o uso de símbolos funcionais e de negação [GARDARIN89]. Algumas dessas extensões (negação e funções de agregação) requerem uma semântica de estratificação, de forma que a execução da consulta proceda por etapas. Além disso, no caso das funções de agregação, é necessária a introdução de novos operadores na linguagem de consultas.

Um processador de DATALOG com extensões deve, então, estar baseado na escolha de uma representação intermediária adequada e na especificação do modelo de execução associado para consultas e conjuntos de regras. Os diversos formalismos propostos por trabalhos da área são, em geral, restritos ao caso de DATALOG sem extensões.

O presente trabalho tem como objetivo apresentar um formalismo que permita expressar consultas recursivas com agregação, bem como um modelo de execução associado e um processador, PROL, que foi

implementado em COMMON LISP [WINSTON89] para ambientes SUN e DOS. O formalismo possibilita a escrita de consultas onde intervêm novos requisitos exigidos pelos sistemas de bancos de dados dedutivos ou orientados a objetos.

O trabalho está organizado da seguinte maneira: na seção 2 é apresentado, através de exemplos, o formalismo RIO, cuja descrição detalhada encontra-se em [LANZECASA89]. Na seção 3, são discutidas as tranformações que o otimizador de consultas realiza sobre um grafo RIO para gerar um grafo ROO. Na seção 4, discute-se o modelo de execução associado a um grafo ROO. A seção 5 apresenta a implementação do processador de ROO, denominado PROL. Na seção 6 são apresentadas as conclusões.

2. RIO: uma representação genérica para consultas recursivas

RIO é um formalismo usado para representar consultas recursivas, que podem conter também agregações. Foi especificado com o objetivo de expressar consultas que são a entrada de um Otimizador de consultas extensível [LANZELOTTE90b].

O ponto de partida para a especificação de RIO são os Grafos Regra/Objetivo ("Rule-Goal Graphs") de Ullmann [ULLMANN85]. Porém, aquele formalismo é acrescido de recursos para expressar estratificação (execução em várias etapas) e funções de agregação.

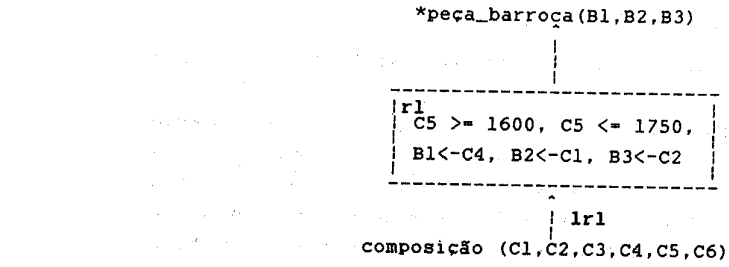
RIO será apresentado através de exemplos. Para uma descrição mais detalhada, ver [LANZECASA89]. Os exemplos pressupõem uma base de dados, cuja parte explícita [LANZELOTTE88b] é constituída pelas seguintes relações base:

compositor (Nome_comp, Data_nasc, Nacionalidade, Professor)
composição (Numero, Titulo, Tonalidade, Nome_comp, Ano, Local)
comp_instr (Nome_comp, Numero, Nome_instr, Parte)

A seguir são apresentadas as relações derivadas (definidas por regras), que constituem a parte implícita da base de dados, com os respectivos grafos RIO.

```
r1  peça_barroca (Nome_comp, Numero, Titulo) :-
                        composição (Numero,Titulo,T,Nome_comp,Ano,L),
                        Ano >= 1600, Ano <= 1750.
```

O grafo RIO correspondente à regra r1 é apresentado a seguir:



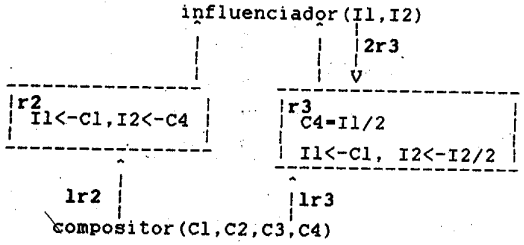
No grafo RIO, os nodos envolvidos por um retângulo, chamados de nodos **regra**, contêm a parte correspondente à seleção e junção da regra (primeira linha), bem como a parte de projeção (segunda linha). São admitidas expressões aritméticas e critérios de seleção de qualquer complexidade.

Em RIO, todos os atributos de relações são submetidos a uma renomeação, que permitirá o tratamento adequado da recursão.

A seguir é mostrada uma visão recursiva:

```
r2  influenciador(Discipulo,Mestre):-compositor(Discipulo,A,N,Mestre).
r3  influenciador(Discipulo,Mestre):-compositor(Discipulo,A,N,Prof),
                                     influenciador(Prof, Mestre).
```

O grafo RIO para a visão definida pelas regras r2 e r3 é apresentado a seguir:

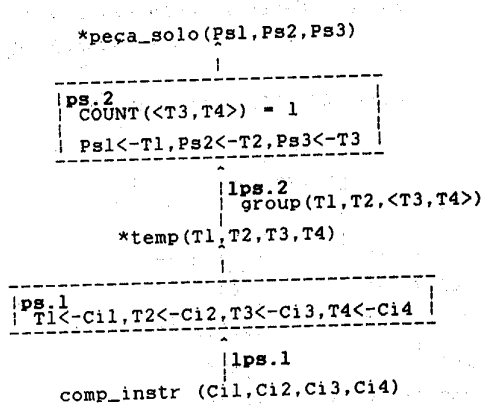


A seguir, uma visão onde é utilizada a agregação:

```
ps  peça_solo (Nome_comp, Numero, Nome_instr):-
        nest(comp_instr(Nome_comp, Numero, Nome_instr, Parte),
             Nome_comp, Numero, <Nome_instr, Parte>),
        count(<Nome_instr, Parte>) = 1.
```

A regra ps define um grupamento sobre a relação comp_instr, de modo a permitir a contagem do número de partes de uma composição e a determinação de composições para um instrumento apenas. A notação usada para indicar o grupamento é inspirada na utilizada por Beeri, na sua proposta de extensão à DATALOG com conjuntos [BEERI87]. Nessa notação, o primeiro argumento de nest é a relação sobre a qual se define o grupamento, os argumentos que seguem, exceto o último, especificam os atributos da relação sobre os quais se faz o grupamento, e o último argumento especifica um conjunto das tuplas constituídas pelos demais argumentos da relação base. O esquema da relação resultante após o nest corresponde aos argumentos de nest a partir do segundo.

A operação de nest, que é equivalente ao GROUP BY da linguagem SQL, requer uma semântica de estratificação, em que a execução de uma consulta se faz em duas etapas: uma primeira etapa recolhe as tuplas da relação base e a segunda etapa produz o grupamento. Essa estratificação é representada em RIO através da geração de dois nodos regra, com o uso de uma relação temporária.

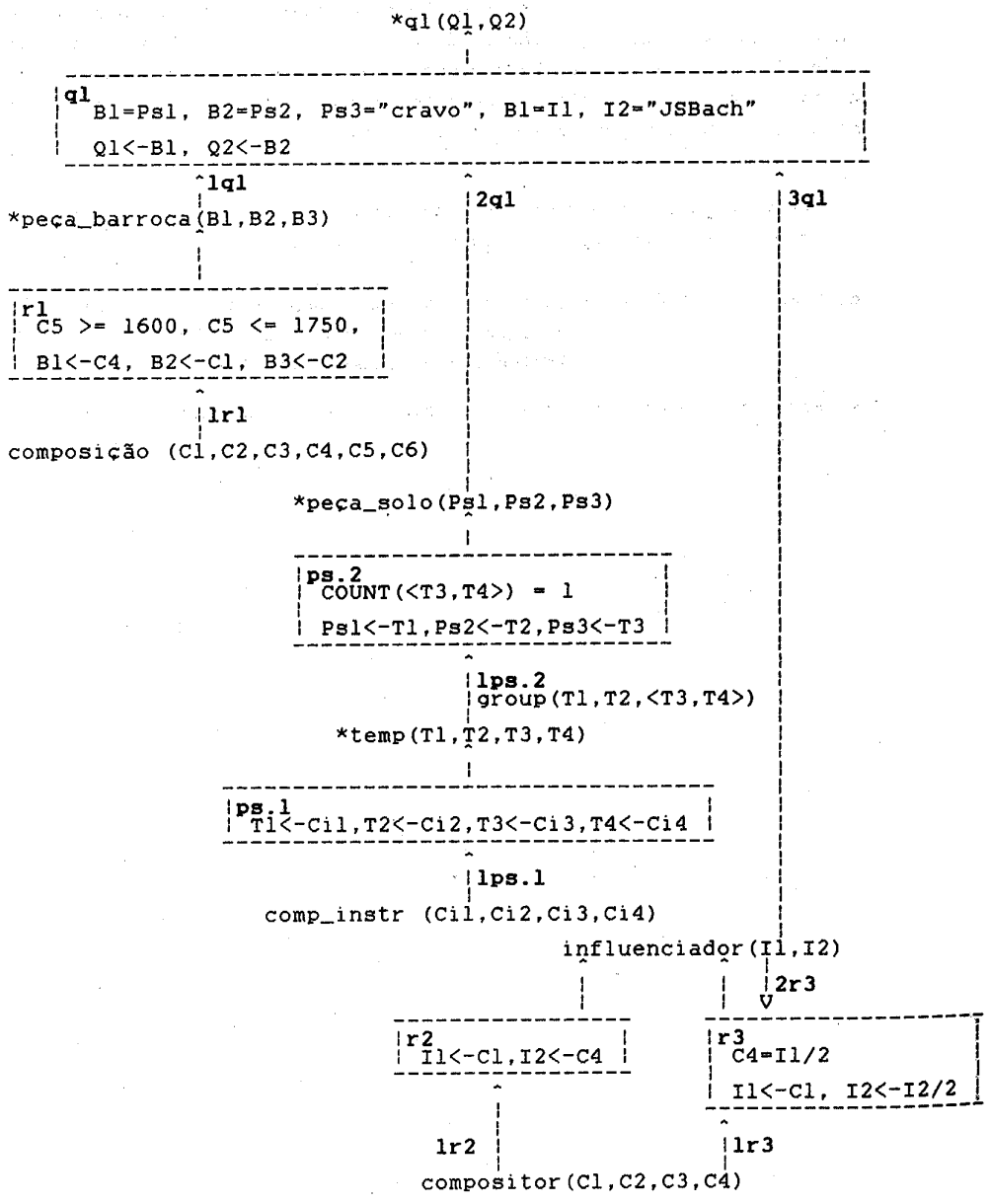


A anotação de group no arco de saida do nodo ps.1 indica que as tuplas devem ser emitidas de maneira ordenada, sendo a sequência de ordenação estabelecida pelos primeiros dois atributos, de modo a permitir o grupamento. A presença da relação *temp no grafo RIO não significa que haverá obrigatoriamente uma ordenação e materialização do resultado ordenado. O otimizador, que converte um grafo RIO em um grafo ROO, decidirá qual a estratégia para obter o resultado ordenado, o que só será explicitado no grafo ROO gerado.

A consulta q1 retorna todas as peças barrocas para cravo, compostas por compositores que foram influenciados por J.S.Bach:

```
q1(Nome_comp, Numero):- peça_barroca (Nome_comp, Numero, Titulo),
                        peça_solo (Nome_comp, Numero, "cravo"),
                        influenciador (Nome_comp, "JSBach").
```

A seguir é apresentado o grafo RIO correspondente à consulta q1:


*q1(Q1,Q2)
q1
B1=Ps1, B2=Ps2, Ps3="cravo", B1=I1, I2="JSBach"
Q1<-B1, Q2<-B2
1q1
2q1
3q1
*peça_barroca(B1,B2,B3)
r1
C5 >= 1600, C5 <= 1750,
B1<-C4, B2<-C1, B3<-C2
1r1
composição (C1,C2,C3,C4,C5,C6)
*peça_solo(Ps1,Ps2,Ps3)
ps.2
COUNT(<T3,T4>) = 1
Ps1<-T1,Ps2<-T2,Ps3<-T3
1ps.2
group(T1,T2,<T3,T4>)
*temp(T1,T2,T3,T4)
ps.1
T1<-Ci1,T2<-Ci2,T3<-Ci3,T4<-Ci4
1ps.1
comp_instr (Ci1,Ci2,Ci3,Ci4)
influenciador(I1,I2)
2r3
r2
I1<-C1,I2<-C4
r3
C4=I1/2
I1<-C1, I2<-I2/2
1r2
1r3
compositor(C1,C2,C3,C4)

3. ROO: uma representação para planos de execução

Um grafo RIO correspondente a uma consulta, mais as informações relativas ao esquema das relações envolvidas, são submetidos como entrada ao otimizador extensível. Este produz, como saída, um plano de execução para a consulta, expresso em ROO. O projeto do otimizador extensível para o qual os formalismos RIO e ROO foram especificados encontra-se em [LANZELOTTE90b]. Aqui serão apresentadas resumidamente as transformações pelas qual um grafo RIO passa para ser transformado em um grafo ROO.

O processo de otimização envolve duas etapas: a reescrita de consultas e a geração de planos de execução [FREITAG89].

A fase de **reescrita** tem como objetivo aplicar transformacões á consulta de entrada para gerar uma nova consulta, equivalente á primeira em termos dos resultados obtidos, mas que propicie mais opções em termos de geração de planos. Um exemplo desse tipo de transformação consiste da substituição de relações derivadas pelas regras que as definem. Por exemplo, uma consulta **a(X,Y)**, formulada sobre um conjunto de regras como

```
a(X,Y):-b(X,Z),c(Z,W),d(W,Y).
b(X,Y):-e(X,Z),f(Z,Y).
e(X,Y):-g(X,Z),h(Z,Y).
```

deve ser transformada em

```
a(X,Y):-g(X,S),h(S,T),f(T,Z),c(Z,W),d(W,Y).
```

Essa transformação é o equivalente à incorporação de visões no modelo relacional. A consequência disso é que a ordem de avaliação dos predicados imposta por uma sequência de regras, que não necessariamente é a melhor do ponto de vista de tempo de execução, pode ser decidida posteriormente durante a fase de geração de planos. A segunda forma apresenta maiores opções de permutação de junções a serem investigadas, portanto, mais planos poderão ser gerados e comparados.

A substituição de relações derivadas só pode ser feita de modo exaustivo quando o conjunto de regras não envolve relações recursivas,

evidentemente. Além disso, deve ocorrer uma renomeação de argumentos.

Após a fase de reescrita, o grafo RIO modificado é submetido à fase de geração de planos, que decidirá para cada nodo **regra**,

(i) qual a melhor ordem das junções;
(ii) como distribuir os filtros de seleção pelos arcos de entrada do nodo, segundo a ordem decidida em (i);
(iii) para cada relação, qual o melhor caminho de acesso;
(iv) para cada junção, qual o melhor método a ser utilizado;

Essas decisões são a consequência da escolha de um **plano ótimo** para a execução da consulta. Um plano ótimo pode ser definido como sendo o melhor dentre todos os planos investigados. Quais e quantos planos serão investigados são fatos dependentes da **estratégia de pesquisa** utilizada durante a fase de geração de planos. A escolha de uma estratégia de pesquisa é um dos pontos centrais em um otimizador de consultas [LANZELOTTE90a].

Ao fim da fase de geração de planos, o grafo RIO é transformado em grafo ROO, que reflete o plano de execução ótimo para a consulta. As principais transformações pelas quais o grafo RIO passa para se tornar um grafo ROO são:

(i) Os arcos de entrada de um nodo são anotados com números de ordem que indicam a sequência em que as relações devem ser consideradas para efeito de junção;

(ii) Todos os predicados de seleção e de junção, que constituiam a primeira linha do nodo RIO, são também distribuidos como anotações pelos arcos de entrada, em função da ordem decidida em (i).

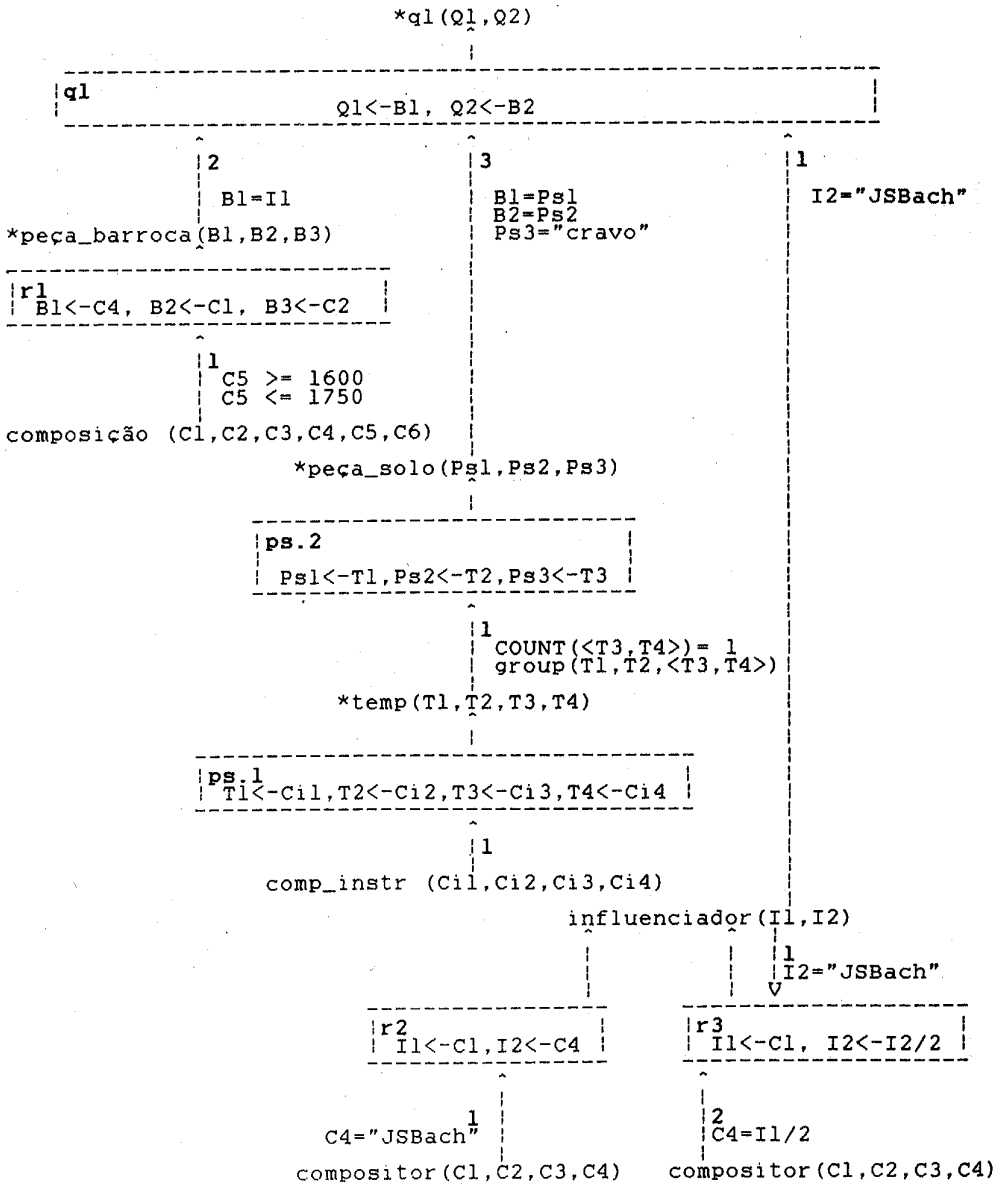
(iii) Para cada relação, é anotado o caminho de acesso escolhido.

(iv) Para cada junção, o nodo ROO correspondente é anotado com o método escolhido. Quando vários métodos de junção são escolhidos para um mesmo nodo RIO, este será quebrado em vários nodos ROO, cada qual correspondendo a um método e ao subconjunto de relações que devem ser

juntadas por esse método. Em princípio, dentro de um mesmo nodo R00 as junções são feitas em "pipelining", mas de um nodo R00 para outro pode ocorrer a materialização de resultados intermediários.

Para a implementação de PROL, algumas simplificações foram adotadas. Para cada relação, apenas o acesso sequencial é assumido como disponível. Para as junções, apenas o método de laços aninhados com "pipelining" é adotado [SELINGER79].

O grafo R00 correspondente à consulta q1 é apresentado a seguir:


*q1(Q1,Q2)
q1
Q1<-B1, Q2<-B2
2
B1=I1
*peça_barroca(B1,B2,B3)
r1
B1<-C4, B2<-C1, B3<-C2
1
C5 >= 1600
C5 <= 1750
composição (C1,C2,C3,C4,C5,C6)
3
B1=Ps1
B2=Ps2
Ps3="cravo"
*peça_solo(Ps1,Ps2,Ps3)
ps.2
Ps1<-T1,Ps2<-T2,Ps3<-T3
1
COUNT(<T3,T4>)= 1
group(T1,T2,<T3,T4>)
*temp(T1,T2,T3,T4)
ps.1
T1<-Ci1,T2<-Ci2,T3<-Ci3,T4<-Ci4
1
comp_instr (Ci1,Ci2,Ci3,Ci4)
1
I2="JSBach"
influenciador(I1,I2)
r2
I1<-C1,I2<-C4
1
C4="JSBach"
compositor(C1,C2,C3,C4)
r3
I1<-C1, I2<-I2/2
2
C4=I1/2
compositor(C1,C2,C3,C4)

4. Modelo de execução associado aos grafos ROO

O modelo de execução associado ao grafo ROO, que será adotado na implementação de PROL, é descrito a seguir.

A primeira importante decisão a ser tomada corresponde ao sentido da avaliação: ascendente ou descendente.

A avaliacão ascendente, que corresponde ao encadeamento para a frente ("forward chaining"), parte das relações base e avalia as regras seguindo a pós-ordem até chegar á raiz, que corresponde à consulta.

A avaliação descendente, ou encadeamento para trás ("backward chaining") parte da raiz, ou seja da consulta, e avalia as regras em prè-ordem até chegar às folhas, ou seja, às relações de base.

As vantagens e desvantagens de cada uma das formas de avaliação foram discutidas em [BANCILHON86]. Uma das diferenças fundamentais consiste no fato de que a abordagem descendente implica automaticamente na consideração apenas dos fatos relevantes para a consulta. Porém o paradigma da avaliação ascendente corresponde melhor ao adotado no processamento de consultas nos SGBD's, pois é dirigido pelo acesso às relações de base.

Com a utilização de filtros, pode-se eliminar a maior desvantagem da abordagem ascendente, que consiste na interveniência de fatos não relevantes para a consulta. Notar que a distribuição dos predicados de seleção e junção pelos arcos de entrada dos nodos do grafo ROO corresponde a uma filtragem, que ocorrerá durante a execução, das tuplas de relação submetidas ao filtro. Isso também é válido para as regras recursivas, como se pode observar no caso das regras r2 e r3, que avaliam a visão recursiva influenciador.

A distribuição de filtros no caso da recursividade é um problema complicado. É difícil projetar um algoritmo genérico para tal, mas pode-se resolver o problema com segurança nos casos de recursão mais utilizados. No exemplo aqui utilizado, trata-se de uma recursividade

linear, ou seja, em que o predicado recursivo só aparece uma vez no corpo da regra recursiva. A filtragem, nesse caso, é discutida em [LANZECASA89], e consiste de uma simplificação do algoritmo apresentado em [KIFER86].

Notar que a escolha de diferentes caminhos de acesso, bem como de diferentes filtros, implica que, no grafo ROO, uma mesma relação aparece tantas vezes quantas for utilizada como entrada em regras diferentes. Por exemplo, a relação compositor aparece duas vezes, e não uma apenas, como no grafo RIO. A cada uma das ocorrências corresponde um filtro diferente.

5. A Implementação de PROL

Com base no modelo de execução proposto, foi desenvolvido um PROcessador, PROL, escrito em COMMON LISP, para consultas recursivas expressas em ROO.

PROL foi desenvolvido com o objetivo de prototipar o ambiente de execução para consultas expressas em RIO. Portanto, no seu projeto foram feitas algumas simplificações compatíveis com a finalidade primeira de prototipação. Por exemplo:

- as relações base estão descritas e amazenadas em ambiente LISP;
- não é assumida a existência de outros caminhos de acesso que não o acesso sequencial;
- não são assumidos outros métodos de junção além do método de laços aninhados com "pipelining".

Apesar das relações residirem em ambiente LISP, foram projetadas funções auxiliares para as operações de entrada/saída, o que garante a modularidade e consequente facilidade de estender o protótipo para tratar relações armazenadas em disco.

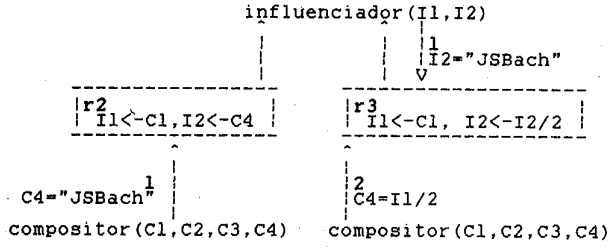
A representação dos nodos ROO em LISP é feita da seguinte forma:

(i) A cada nodo regra ROO corresponde uma lista com quatro campos:

project, **select**, source e destin. O campo **project** decide a projeção a **ser feita nas tuplas de saída do nodo.** O campo **select** está presente **por questões de compatibilidade** com RIO, mas é sempre nulo nos nodos **ROO.** O **campo source** indica quais os nodos arco que entram nesse nodo **regra;** a ordem em que os arcos são representados corresponde à ordem **de junção decidida** pelo otimizador. O campo destin indica o nome de **uma relação derivada** que recebe os resultados da regra.

(ii) a cada arco ROO corresponde uma lista com três campos: source, **parms** e select. O campo source indica qual a relação entrada para o **arco,** que tanto pode ser uma relação base como uma relação derivada. O **campo parms** contém a lista de atributos da relação de entrada **representados** através de seus identificadores obtidos a partir da **renomeação** que ocorre em RIO. O campo **select** contém uma lista de **predicados** a serem aplicados nas tuplas que fluem através desse arco, **ou seja,** corresponde aos filtros anotados no arco.

A seguir é mostrado o trecho em LISP usado por PROL que corresponde ao **subgrafo** ROO da consulta q1:




```
(setf (get 'r2 'project) '((i1 c1/1) (i2 c4/1)))
(setf (get 'r2 'select) nil)
(setf (get 'r2 'source) '(r2_1))
(setf (get 'r2 'destin) 'influenciador)

(setf (get 'r2_1 'source) 'compositor)
(setf (get 'r2_1 'parms) '(c1/1 c2/1 c3/1 c4/1))
(setf (get 'r2_1 'select) '((= c4/1 'jsbach)))

(setf (get 'r3 'project) '((i1 c1/2) (i2 i2/1)))
```

```
(setf (get 'r3 'select) nil)
(setf (get 'r3 'source) '(r3_1 r3_2))
(setf (get 'r3 'destin) 'influenciador)

(setf (get 'r3_1 'source) 'influenciador)
(setf (get 'r3_1 'parms) '(i1/1 i2/1))
(setf (get 'r3_1 'select) '((= i2/1 'jsbach)))

(setf (get 'r3_2 'source) 'compositor)
(setf (get 'r3_2 'parms) '(c1/2 c2/2 c3/2 c4/2))
(setf (get 'r3_2 'select) '((= c4/2 i1/1)))
```

As conexões entre nodos regra ROO são estabelecidas através de uma estrutura de árvore a parte. Nessa estrutura, para cada nodo regra ROO são indicados quem é o seu filho à esquerda e quem é o seu irmão, ambos sendo também nodos ROO. A seguir encontra-se o trecho PROL que estabelece as conexões entre os nodos ROO do grafo correspondente à consulta q1.

```
(setf (get 'q1 'son) 'r1)
(setf (get 'r1 'son) 'r2)
(setf (get 'r2 'brother) 'r3)
(setf (get 'r2 'son) 'ps.2)
(setf (get 'sp.2 'son) 'ps.1)
```

O Apêndice mostra o código de PROL, que procede da seguinte forma:

(i) Partindo do nodo raiz da árvore, que é o nodo regra correspondente à consulta, uma função de avaliação recursiva percorre a árvore em pós-ordem, avaliando primeiro os filhos e por último a raiz de cada sub-árvore. Tal procedimento está contido nas funções **exec_query** e **exec_query_aux**.

(ii) Para cada nodo regra, uma função exec_rule lê a primeira tupla da primeira relação de entrada, segundo a ordem de entrada dos arcos, e aplica os filtros de seleção anotados no arco. Para cada tupla, são lidas sucessivamente as tuplas das relações correspondentes aos outros arcos e aplicadas as respectivas seleções, em um procedimento que simula o método de junção por laços aninhados. Ao final da

consideração de todos os arcos de entrada para o nodo regra, a tupla resultante é submetida á projeção e armazenada na relação de saída **destin**.

Um nodo ROO que corresponda a uma regra recursiva será avaliado várias vezes, até que uma avaliação não produza mais novos resultados. A cada vez, novas tuplas geradas são escritas na relação recursiva e serão lidas na próxima avaliação do nodo. O critério de término corresponde ao fato de que uma avaliação não produza mais nenhuma tupla nova. Esse modelo corresponde a uma avaliação semi-ingênua [BANCILHON86], mas aqui a eficiência é aumentada pela existência dos filtros, que deixam passar apenas os fatos relevantes para cada regra.

6. Conclusão

O formalismo RIO/ROO pode ser usado como representação intermediária para consultas envolvendo recursão e agregação, requisitos necessários para expressar consultas a bancos de dados dedutivos e orientados a objetos [GARDARIN89] [VALDURIEZ86].

Usando um grafo RIO como entrada, o otimizador extensível o transforma em um grafo ROO, que representa o plano de execução escolhido para a consulta RIO.

Com base no modelo de execução associado a ROO, foi desenvolvido um processador PROL, escrito em COMMON LISP para ambientes SUN e DOS, com o objetivo de prototipar o ambiente de execução dos grafos ROO. O protótipo encontra-se em uso no laboratório da PUC-RIO.

Os próximos passos desse trabalho consistirão na extensão de PROL para tratar diferentes caminhos de acesso e métodos de junção. Também está sendo projetada uma ferramenta gráfica para permitir a fácil visualização dos grafos RIO e ROO, o que será de grande auxílio na percepção dos resultados da otimização.

7. BIBLIOGRAFIA

[BANCILHON86] Bancilhon F. e Ramakrishnan R. "An Amateur's Introduction to Recursive Query Processing Strategies", Proc. ACM SIGMOD Conference 1986.

[BATORY88] Batory D.S., "Concepts for a Database System Compiler" Proc. ACM PODS, 1988.

[BEERI87] Beeri C., Naqvi S., Shmueli O. e Tsur S., "Sets and Negation in a Logic Database Language (LDL1)", Proc. ACM PODS, 1987.

[FREYTAG87] Freytag J.C., "A Rule-Based View of Query Optimization", Proc. ACM SIGMOD Conference, 1987.

[FREYTAG89] Freytag J.C., "The Basic Principles of Query Optimization in Relational Database Management Systems", Information Processing IFIP 89, North Holland, 1989.

[GARDARIN89] Gardarin G. e Valduriez P., "Relational Databases and Knowledge Bases", Springer Verlag, 1989.

[KIFER86] Kifer M. e Lozinskii E.L., "A Framework for an Efficient Implementation of Deductive Databases", Proc. 6th. Advanced Database Symp., Tokyo, Japan, 1986.

[KIFER88] Kifer M. e Lozinskii E.L., "SYGRAF: Implementing Logic Programs in a Database Style", IEEE Trans. on Software Engineering, vol. 14, no. 7, 1988.

[KUPER87] Kuper G., "Logic Programming with sets", Proc. ACM Pods, 1987.

[LANZECASA89] Lanzelotte R.S.G. e Casanova M.A., "A Common Framework for Static Query Optimization in Deductive and Relational Database Systems", Relatório Técnico CCR 096, Centro Cientifico RIO, IBM Brasil, Rio de Janeiro, 1989.

[LANZELOTTE88a] Lanzelotte R.S.G., "Processamento de Consultas Recursivas: uma discussão", Procs. III Simpósio de Bancos de Dados, Recife, 1988.

[LANZELOTTE88b] Lanzelotte R.S.G., "Sistemas de Base de Conhecimento" em Melo R.N., "Bancos de Dados não Convencionais", Cap. 7, Livro da VI Escola de Computação, Campinas, 1988.

[LANZELOTTE90a] Lanzelotte R.S.G., "Modelling Search Strategies in an Extensible Query Optimizer", Procs. VI Journées Bases de Donées Avancées, Montpellier, 1990.

[LANZELOTTE90b] Lanzelotte R.S.G., "EXPLORE: um Otimizador de Consultas EXtensivel", Tese de Doutorado, Pontificia Universidade Católica do Rio de Janeiro, em preparação.

[LEE88] Lee M., Freytag J.C. e Lohman G., "Implementing an Interpreter for Functional Rules in a Query Optimizer", Proc. 14th VLDB Conference, California, 1988.

[LOHMAN88] Lohman G., "Grammar-like Functional Rules for Representing Query Optimization Alternatives", Proc. ACM SIGMOD Conference, 1988.

[SELLIS85] Sellis T. e Shapiro L., "Optimization of Extended Database Query Languages", Proc. ACM SIGMOD Conference, 1985.

[SELLIS87] Sellis T., "Efficiently supporting procedures in relational database systems", Proc. ACM SIGMOD Conference, 1987.

[SELINGER79] Selinger P.G. et al, "Access path selection in a relational database management system", Proc. ACM SIGMOD Conference, Boston, 1979.

[ULLMANN85] Ullmann J., "Implementation of Logical Query Languages for Databases", Proc. of the Islamorada Workshop on Large Scale Knowledge Bases and Reasoning Systems, 1985.

[VALDURIEZ86] Valduriez P., Khoshafian S. e Copeland G., "Implementation Techniques of Complex Objects", Proc. 12th VLDB Conference, Kyoto, 1986.

[WINSTON89] Winston P.H. e Horn B.K.P., "LISP", Addison Wesley, 3. edição, 1989.

8. APENDICE

A seguir é apresentado o código COMMON LISP de PROL.

```
;=================================================================
; Definição das funções e procedimentos que executam uma consulta
; representada por um grafo ROO.
;=================================================================


;-----------------------------------------------------------------
; Essa função retorna verdadeiro (T), caso a avaliação de todos os
; predicados na lista de seleção (SELECT), retorne verdadeiro.
;-----------------------------------------------------------------
(defun eval_select (select)
   (if (or (eq select t) (endp select))
      t
      (let ((aux (first select)))
         (case (first aux)
            ('= (when (eql (eval (second aux)) (eval (third aux)))
                  (eval_select (rest select)) ) )
            ('> (when (> (eval (second aux)) (eval (third aux)))
                  (eval_select (rest select)) ) )
            ('< (when (< (eval (second aux)) (eval (third aux)))
                  (eval_select (rest select)) ) )
            ('>= (when (>= (eval (second aux)) (eval (third aux)))
                  (eval_select (rest select)) ) )
            ('<= (when (<= (eval (second aux)) (eval (third aux)))
                  (eval_select (rest select)) ) )
            ('<> (when (<> (eval (second aux)) (eval (third aux)))
                  (eval_select (rest select)) ) )
            (otherwise nil) ) ) ) )


;-----------------------------------------------------------------
; Esse procedimento executa as projeções especificadas num dado nodo
;        ROO.
;-----------------------------------------------------------------
(defun exec_project (project)
   (mapcar #'(lambda (l)
               (set (first l) (eval (second l))) )
           project ) )


;-----------------------------------------------------------------
; Esse procedimento aciona a avaliação dos filtros de uma determinada
; regra (nodo de ROO).
;-----------------------------------------------------------------
(defun exec_rule (rule)
   (let ((filters (get rule 'source)))
      (exec_filters rule filters) ) )
```

```
;-------------------------------------------------------------------
; Avalia cada tupla das relações base sobre as condições impostas na
; parte SELECT dos filtros correspondentes.
; O procedimento utiliza a recursão para avaliar todos os filtros da
; regra. Quando não há mais filtros para testar e tem-se tuplas
; válidas os dispositivos de projeção e escrita nas relações derivadas
; são acionados.
;-------------------------------------------------------------------
(defun exec_filters (rule filters)
   (if (endp filters)                          ; base da recursão
      ;
      ; escreve na relação DESTIN a tupla retornada por EXEC_PROJECT
      ;
      (write_db (get rule 'destin) (exec_project (get rule 'project)))
      ;
      (let* ((filter (first filters))
             (relation (get filter 'source))
             (select (get filter 'select))
             (parms (get filter 'parms))
             (pointer (gensym))
             (opers (get filter 'opers)) )
        (when opers (exec_opers opers))
        (reset pointer)
         ;
         ; corpo principal: executa até o fim da relação base
         ;
         (loop
            (if (read_db relation pointer parms)
               ;
               ; avalia a parte SELECT de um filtro sobre uma tupla.
               ; Para cada tupla selecionada, executa o resto da lista
               ; FILTERS.
               ;
               (when (eval_select select)
                  (exec_filters rule (rest filters)) )
               (return t) ) ) ) ) )


;===================================================================
; Esse procedimento executa uma consulta utilizando a árvore de
; conexões.
;===================================================================
(defun exec_query (query)
   (gensym 0)
   (exec_query_aux query)                  ; applica a consulta
   (eval (get query 'destin)) )

(defun exec_query_aux (rule)
   (unless (null rule)                     ; base da recursão
      (let ((son (get rule 'son))
            (brother (get rule 'brother)) )
         (exec_query_aux son)              ; chamada recursiva com SON
         (exec_rule rule)                  ; executa a regra
         (exec_query_aux brother) ) ) )    ; chamada recursiva c/ BROTHER
```


459