**Implementação de um Banco de Dados Relacional Dotado de uma Interface Cooperativa**

*Nascif Abrão Abousalh Neto*
*Ariadne Maria Brito Rizzoni Carvalho*

**Relatório Técnico DCC–06/93**

# Implementação de um Banco de Dados Relacional Dotado de uma Interface Cooperativa

Nascif Abrão Abousalh Neto*
Ariadne Maria Brito Rizzoni Carvalho

12 de abril de 1993

**Sumário**

Este documento apresenta um sistema de gerenciamento de bancos de dados que foi desenvolvido com o objetivo de permitir o estudo de interfaces cooperativas. O sistema adota o modelo relacional, utilizando os recursos da linguagem PROLOG para implementar as relações que compõem o banco de dados e realizar a avaliação de consultas, que podem ser construídas de forma interativa, tornando desnecessário o conhecimento de uma linguagem formal de acesso ao banco de dados. A transformação da consulta do usuário num conjunto de metas PROLOG equivalentes envolve um processo de otimização, que leva em conta o estado do banco de dados e isola a avaliação de partes independentes da consulta. A característica cooperativa do sistema está em sua capacidade de identificar falsas pressuposições do usuário e elaborar respostas que as corrijam.

*Projeto financiado pela FAPESP - Fundação de Amparo à Pesquisa do Estado de São Paulo – processo 91/0846-6

# 1 Introdução

Uma das características que tornam o diálogo em linguagem natural uma ferramenta tão poderosa para troca de informações é a existência de um *contexto do diálogo*, que pode ser definido como sendo o conjunto de crenças, fatos e regras de comportamento comuns aos participantes do diálogo, e que portanto são assumidas implicitamente na avaliação de suas afirmações. Com isso, muitas informações adicionais àquelas obtidas por meio de uma interpretação puramente literal podem ser extraídas da afirmação pelo outro participante.

Utilizando estas informações adicionais o ouvinte pode encontrar inconsistências entre a afirmação e o contexto, alertando o emissor da afirmação sobre este fato, e/ou identificar uma possível intenção do emissor e fornecer informações relevantes para sua concretização. É importante observar que estas atividades caracterizam o diálogo em linguagem natural como um processo *cooperativo*, no sentido de que o ouvinte pode tomar a iniciativa e fornecer ao emissor informações úteis sem que um pedido explícito tenha sido efetuado.

Esta situação não ocorre nas interfaces tradicionais dos sistemas gerenciadores de bancos de dados; como a única interpretação que estas interfaces possuem das consultas de seus usuários é o seu sentido imediato, literal, não há como derivar as informações adicionais necessárias para a exibição de um comportamento cooperativo conforme o descrito acima.

A solução para este problema não implica necessariamente na construção de uma interface com capacidade de processamento de linguagem natural. A capacidade de aceitar a língua nativa do usuário permite que "pistas" lingüísticas sejam utilizadas na transmissão de informações periféricas ao sentido literal da consulta; no entanto, esta capacidade não é indispensável para a manutenção de um contexto do diálogo.

Neste trabalho buscamos explorar a idéia de que as interfaces existentes podem incorporar mecanismos que facilitem sua interação com o usuário mantendo a simplicidade associada ao processamento de linguagens formais. Com este intuito foi implementado um sistema de

gerenciamento de banco de dados, que utiliza os recursos da linguagem PROLOG[Cloc 84, Arit 87, Arit 88] para representar o esquema das relações e realizar a avaliação de consultas. As principais características da interface deste sistema são:

- A utilização de um modelo simplificado do contexto do diálogo (que inclui o conteúdo atual do banco de dados) na análise das consultas do usuário. Esta análise é realizada nas consultas que não puderam ser satisfeitas pelo banco de dados; no lugar de informar simplesmente que nenhuma resposta pode ser obtida, a interface elabora uma resposta corretiva identificando as pressuposições do usuário que por serem falsas causaram a falha na avaliação da consulta.

- A utilização de janelas do tipo caixa de diálogo[1] e controles como botões e listas para suportar a construção interativa das consultas ao banco de dados. Internamente a consulta é traduzida para uma forma intermediária, expressa num subconjunto da linguagem QUEL[Ston 75]; esta forma intermediária é por sua vez convertida num conjunto de metas PROLOG para realização da avaliação da consulta.

Este artigo está organizado da seguinte forma: na seção 2 o processo de transformação de uma consulta expressa na linguagem QUEL em um conjunto equivalente de metas PROLOG é analisado, sendo o processo ilustrado pelos exemplos da seção A; na seção 3 propõe-se o PROLOG como formalismo de representação de bancos de dados e discute-se um método de otimização da avaliação de consultas nesta forma; na seção 4 o processo de elaboração de respostas cooperativas é introduzido e, finalmente, na seção 5 a interface gráfica para geração de consultas é descrita, juntamente com exemplos de sua utilização.

[1] em inglês: *dialog box*

# 2 Processamento de Consultas expressas em QUEL

A linguagem QUEL foi escolhida como formalismo para o acesso ao banco de dados por vários fatores, sendo os principais:

- Sua proximidade com a lógica de primeira ordem, podendo ser vista como uma variante sintática de um subconjunto da mesma. Este fato simplifica a transformação de consultas expressas na linguagem QUEL em metas PROLOG.
- A sua ampla utilização comprovada pela literatura, o que facilita a comparação com outros sistemas. Ao usar o mesmo formalismo, tivemos a chance de aproveitar os mesmos casos de teste.

O projeto do processador de consultas foi baseado originalmente na implementação do sistema PIQUE [Maie 88]. Foi adotada uma arquitetura modular, refletindo as várias fases do processo de análise e processamento da consulta. Estas fases serão descritas nas seções seguintes.

## 2.1 Leitura da Consulta

Em seu modo mais simples, a interface oferece ao usuário o *prompt* "QEL>" e aguarda que ele digite um comando na linguagem QUEL. São oferecidos recursos simples de edição de linha, além da possibilidade de continuação de um comando na linha seguinte (o que é indicado terminando-se a linha com o caracter "\"). O comando é convertido numa sequência de caracteres e submetido à análise léxica.

## 2.2 Análise Léxica

Nesta fase uma sequência de caracteres representando a consulta do usuário é transformada em uma sequência de *tokens*. Estes tokens correspondem aos símbolos terminais da gramática da linguagem de acesso e/ou a valores que podem ser assumidos pelos campos das relações (por

exemplo, constantes inteiras ou cadeias de caracteres delimitadas por aspas simples).

## 2.3 Análise Sintática

A finalidade deste fase é verificar se a sequência de tokens gerada na fase anterior é válida segundo as regras sintáticas da linguagem adotada. Esta validação é realizada por uma gramática implementada segundo o formalismo DCG[Pere 80]. A estrutura resultante da análise sintática deve permitir a geração de uma lista de metas PROLOG por meio de um processo de análise semântica.

### 2.3.1 O formato de uma consulta em QUEL

Para facilitar o entendimento da fase de análise sintática, vamos fazer uma breve descrição da sintaxe de uma consulta na linguagem QUEL. Maiores detalhes podem ser encontrados em [Delo 85].

Uma consulta é composta de três partes:

- Declaração de variáveis de tupla:

  RANGE OF *variável de tupla* IS *nome de relação*

  Podem existir tantas declarações de variáveis de tupla quantas forem necessárias, introduzindo relações distintas ou não. Uma variável de tupla serve para identificar explicitamente a relação que está sendo referenciada. Este conceito é especialmente útil na junção[2] de relações e na operação de auto-junção. Uma vez declarada, a variável de tupla permanece associada à relação em questão até que uma nova declaração da mesma variável seja efetuada, podendo ser utilizada em outras consultas. Em outras palavras, o escopo de uma declaração de variável de tupla abrange toda a sessão, e não somente a consulta em que foi realizada.

[2] Em inglês: *join*

- Lista de objetivos:

  RETRIEVE ( *lista de objetivos* ):

  A lista de objetivos é composta de termos de projeção, relacionando os campos das relações que devem ser mostrados para cada conjunto de tuplas que satisfaça a qualificação.

  Um *termo de projeção* representa o valor do campo de uma relação e é identificado pela seguinte notação:

  <variável de tupla>.<nome de campo>

- Qualificação:

  WHERE *qualificação*

  A qualificação representa a combinação de condições que devem ser satisfeitas para que um conjunto de tuplas seja aceito como resposta. A linguagem QUEL possui várias classes, de acordo com a complexidade dos operadores válidos numa qualificação. Na classe $QUEL_0$ - que foi tomada como modelo neste sistema - uma qualificação pode ser:

  - Uma fórmula atômica no formato $f\Theta g$, onde $f$ e $g$ são termos de projeção ou constantes[3] e $\Theta$ é um dos operadores de comparação $>, \geq, <, \leq, =$ ou $\neq$.
  - Qualquer expressão elaborada a partir de fórmulas atômicas e dos operadores lógicos NOT, AND e OR.

[3] Na especificação original são aceitas também operações aritméticas envolvendo estes termos, mas esta funcionalidade ainda não foi implementada

Uma qualificação também pode ser vista como uma *árvore*, na qual os nós internos são operadores lógicos e as folhas são fórmulas atômicas no formato acima descrito (é desta forma que o usuário vê uma qualificação na interface gráfica). Esta abstração é importante para o entendimento dos próximos módulos, e será referenciada posteriormente.

Assim, para considerar válida uma consulta o analisador sintático deve verificar se:

- As declarações de variáveis de tupla se referem a relações conhecidas;
- Os termos de projeção são consistentes (em sua associação de nomes de campo e relações) com os esquemas do banco de dados;
- Os argumentos das comparações (constantes e/ou termos de projeção) pertencem a domínios compatíveis;

Se estas condições forem satisfeitas, é gerada uma estrutura contendo a lista de predicados introduzidos pelas declarações de variáveis de tupla, a lista de termos de projeção correspondente a lista de objetivos e uma árvore formada por operadores lógicos e de comparação equivalente a qualificação. Caso alguma destas condições seja violada, mensagens de erro apropriadas serão emitidas.

## 2.4 Análise Semântica

Como dissemos anteriormente, a linguagem QUEL pode ser vista como uma variação sintática de um subconjunto da lógica de primeira ordem, que por sua vez pode ser expresso diretamente em PROLOG. Uma maneira de fazer a conversão do primeiro formalismo para o segundo é a seguinte:

- Cada declaração de variável de tupla introduz uma instância do predicado utilizado para representar a relação. Todos os argumentos desta instância são variáveis livres, que serão referenciadas pelos termos de projeção;

- Cada termo de projeção é substituído pela variável livre correspondente; as variáveis resultantes da substituição dos elementos da lista de objetivos são utilizadas para obtenção das respostas. Estas variáveis são chamadas de *variáveis de retorno*.

- A árvore de qualificação é mapeada diretamente nos operadores lógicos e de comparação apropriados.

Aplicando estas regras, teríamos a seguinte transformação:

```
range of E is emprestimo
range of U is usuario
retrieve (U.nome) where E.ra = U.ra and E.tombo = 1234

answer(Q) :- usuario(P,Q,R), emprestimo(S,T), P = S, T = 1234
```

Apesar da vantagem de ser extremamente simples (poderia até ser realizada diretamente como subproduto da análise sintática), esta transformação gera construções ineficientes do ponto de vista de sua execução pelo PROLOG. No exemplo acima poderíamos tornar implícitas as unificações, o que geraria a seguinte meta:

```
answer(Q) :- usuario(P,Q,R), emprestimo(P,1234).
```

É altamente desejável que o conjunto de metas gerado possua unificações implícitas sempre que possível, pois isto permite ao PROLOG considerar um número bem menor de alternativas na avaliação da consulta.

Uma questão mais séria a ser considerada é a da equivalência entre a construção gerada e a consulta original: estas regras de transformação podem gerar construções incorretas quando aplicadas a consultas que envolvam operadores lógicos que modifiquem o *escopo* das variáveis, como a negação e a disjunção (este conceito será analisado em detalhes na próxima seção).

Diante destes argumentos, fica claro que somente a análise sintática não é suficiente para gerar metas PROLOG cuja avaliação irá gerar por

sua vez respostas à consulta. O processo de análise semântica foi introduzido para garantir que as construções geradas sejam corretas e para tornar implícitas, sempre que possível, as unificações. Veremos mais adiante que isto ainda não é o bastante para gerar metas cuja avaliação seja eficiente.

### 2.4.1 O Conceito de Escopo

Um *escopo* é um dos possíveis "caminhos" tentados pelo mecanismo de dedução do PROLOG durante a avaliação de um conjunto de metas.

As metas pertencentes a este caminho formam uma região na qual a instanciação de suas variáveis é visível. Este conceito nos permite afirmar que só é correto tornar implícita uma unificação dentro do escopo que contém a instância que define a variável unificada.

Uma outra consequência desta idéia é que qualquer duplicação de uma determinada instância de predicado dentro de um escopo será redundante. Assim, antes de realizar uma inserção devemos garantir que ela ainda não ocorreu no escopo local (ou no escopo ancestral, como veremos adiante).

A estrutura dos escopos é determinada pela estrutura dos operadores lógicos, da seguinte forma:

- Os ramos de um AND pertencem ao mesmo escopo porque compartilham os efeitos da instanciação de variáveis;
- Os ramos de um OR criam escopos distintos, porque um ramo só será executado se o anterior falhar. Isso implica que qualquer instanciação realizada pelo ramo que falhou será automaticamente desfeita antes do início da execução do ramo seguinte;
- Uma negação cria um novo escopo porque instanciações realizadas em seu interior são desfeitas após a sua execução.

Os operadores OR e NOT introduzem novos escopos locais, que não afetam com suas unificações as metas anteriores. No entanto, estas metas afetam o estado das variáveis dentro dos novos escopos, e devem ser

levadas em conta no seu processamento. Dizemos que as metas anteriores constituem um escopo *ancestral*, que é herdado pelos novos escopos. O escopo ancestral deve ser examinado na verificação da ocorrência de uma instância, mas não pode ser alterado pela realização de uma unificação implícita envolvendo uma das variáveis por ele definidas.

Podemos concluir, portanto, que a transformação da árvore de qualificação num conjunto de metas não deve ser orientada para os nós, mas para os escopos. A primeira ocorrência de uma das variáveis de uma instância dentro de um escopo causa a introdução desta instância numa lista a ele associada; além de ser utilizada para garantir que não serão introduzidas duas cópias da instância neste escopo, esta lista servirá posteriormente para gerar o conjunto de metas.

### 2.4.2 Pré-processamento da Árvore de Qualificações

O processo de transformação começa com a associação de variáveis livres aos campos das instâncias de predicado (introduzidas pelas variáveis de tupla na análise sintática). Duas instâncias do mesmo predicado recebem variáveis distintas para seus argumentos.

A seguir, a árvore de qualificação é compactada de modo a facilitar a determinação dos escopos. Esta compactação consiste na transformação dos operadores lógicos binários e unários em operadores n-ários. Um exemplo deste processo pode ser visto a seguir:

$$\mathsf{and}(\mathsf{A} > \mathsf{B},\ \mathsf{and}(\mathsf{C} > \mathsf{D},\ \mathsf{and}(\mathsf{E} > \mathsf{F})))$$

torna-se

$$\mathsf{and}([\mathsf{A} > \mathsf{B},\ \mathsf{C} > \mathsf{D},\ \mathsf{E} > \mathsf{F}])$$

Durante a compactação os nós internos dos operadores lógicos são ordenados numa forma conveniente ao procedimento de inserção de instâncias, ou seja, os operadores que geram novos escopos são deixados para o final. Por meio deste procedimento conseguimos minimizar o número de

duplicatas de instâncias (que podem surgir quando uma variável de tupla é referenciada simultaneamente em escopos separados, sem ter aparecido anteriormente num escopo ancestral).

A ordenação faz com que o operador OR seja sempre o último ramo (de um AND ou de um NOT) a ser analisado durante a fase de análise de escopos; isto impede a existência de caminhos alternativos levando a um mesmo nó (o que faria com que ele pertencesse a dois ou mais escopos ao mesmo tempo). Embora este caso possa ocorrer na execução da consulta, ele não deve surgir durante a fase de inserção de predicados. Além disso, garantimos através da aplicação de transformações aos operadores lógicos que nunca haverá mais que um OR como ramo de um operador lógico (mais uma vez, para impedir a existência de caminhos alternativos levando a um mesmo nó).

Em paralelo às operações de compactação e ordenação, os termos de projeção contidos nas folhas e na lista de objetivos são associados às variáveis livres correspondentes aos campos das instâncias.

### 2.4.3 Inserção de Predicados sob Demanda e Tratamento da Unificação

Após o pré-processamento da árvore, executamos um percurso em largura cujo resultado é a construção de uma nova árvore com a mesma estrutura de operadores lógicos da anterior, mas que contém instâncias de predicados e unificações implícitas refletindo a organização da estrutura de escopos.

Antes de iniciar o percurso, inicializamos o escopo da raiz com as instâncias que definem as variáveis de retorno. Esta foi a maneira mais simples de garantir que teremos acesso aos seus valores durante a avaliação da consulta. Um exemplo de consulta que torna necessário este procedimento é apresentado a seguir:

```
range of U is usuario
retrieve (U.nome) where not U.ra > 870840
```

Se não inicializássemos o escopo da raiz com uma instância do pre-

dicado **usuario**, teríamos como resultado a seguinte meta[4]:

```
answer(Q) :- \+ (usuario(P,Q,R), Q > 870840).
```

A avaliação desta meta não gera o resultado esperado, pois a instanciação da variável **Q** só é visível dentro do escopo introduzido pela negação. Se, no entanto, fizéssemos a inicialização do escopo da raiz conforme descrito anteriormente, teríamos:

```
answer(Q) :- usuario(P,Q,R), \+ Q > 870840.
```

Agora a instanciação da variável está num escopo acessível, e a avaliação da meta pode ser realizada sem problemas.

Após a inicialização do escopo da raiz passamos a analisar cada um dos nós da árvore, mantendo um registro das instâncias presentes no escopo local e no escopo ancestral e das modificações na estrutura dos escopos. O processamento é realizado conforme o tipo do nó:

**Conjunção** O processamento de uma conjunção faz com que as instâncias eventualmente definidas em seus ramos sejam incluídas no escopo local.

**Disjunção** O encontro de uma disjunção provoca a concatenação do conteúdo do escopo local com o do escopo ancestral e a associação de novos escopos locais, inicialmente vazios, a cada um de seus ramos. Os novos escopos são então analisados de maneira independente.

**Negação** Uma negação também introduz um novo escopo inicialmente vazio; no entanto ela não adiciona ao escopo ancestral as instâncias que venham a ser definidas em seu escopo. Com isso garantimos que, se uma variável definida dentro do escopo da negação for utilizada novamente fora deste escopo, uma cópia da instância será providenciada para definir a variável.

[4] O operador \+ denota negação em PROLOG.

**Comparação** Uma comparação nunca instancia variáveis; portanto, para processá-la temos apenas que garantir que as variáveis envolvidas estejam instanciadas. Buscamos uma instância dos predicados (possivelmente uma só) que defina as variáveis no escopo local e no escopo ancestral (nesta ordem). Se a encontrarmos, nada mais precisamos fazer; caso contrário, inserimos esta instância no escopo local.

**Unificação** Inicialmente, procedemos a uma busca pelas instâncias que definem as variáveis envolvidas da mesma forma que no tratamento de uma comparação. A seguir, verificamos se ela pode ser realizada implicitamente; isto só pode ocorrer se as seguintes condições forem satisfeitas:

1. Um dos termos envolvidos na unificação for uma variável definida no escopo local;
2. Esta variável ainda não tiver sido instanciada (unificada implicitamente com uma constante).

A estrutura produzida pela análise semântica, embora essencialmente equivalente a um conjunto de metas, é mantida numa forma não executável que é mais adequada ao processamento pelo módulo de otimização.

Os conceitos de escopo, compactação e ordenação da estrutura de operadores lógicos e de inserção de instâncias sob demanda foram desenvolvidos como parte deste trabalho.

# 3 Otimização

O resultado da análise semântica é uma construção equivalente a um conjunto de metas PROLOG [Cloc 84]. Lembrando que as relações que compõem o banco de dados são representadas no mesmo formalismo (um registro ou tupla corresponde a uma cláusula unitária), vemos que o mecanismo de dedução do PROLOG pode ser utilizado na avaliação

da consulta. Esta é uma avaliação exaustiva, gerando todas as possíveis instanciações das variáveis de retorno. O conjunto destas instanciações é o resultado da consulta.

Para avaliar um conjunto de metas, o mecanismo de dedução do PROLOG tenta encontrar uma unificação para cada uma das metas do conjunto seguindo a ordem em que se apresentam (da esquerda para a direita). A avaliação (não exaustiva) de uma meta busca uma unificação com as cláusulas que a definem na ordem em que estas se apresentam no Banco de Dados. Se houver sucesso, as variáveis desta meta são instanciadas. Se nenhuma unificação for possível ou se não houver mais metas a analisar, é realizado o retrocesso. Em outras palavras, o PROLOG retorna à última meta unificada, desfaz os efeitos da unificação e tenta uma unificação alternativa.

Uma das deficiências deste mecanismo é que ele assume que todas as metas em uma conjunção são interdependentes. Isto está correto desde que estas metas possuam variáveis livres em comum; as partes de uma conjunção que não compartilham variáveis entre si podem ser avaliadas de forma independente, reduzindo o número de alternativas que precisam ser examinadas.

Podemos concluir que para diminuir o custo da avaliação de uma consulta devemos minimizar o número de alternativas consideradas pelo PROLOG durante sua execução. Isto pode ser feito de duas formas: buscando uma reordenação ótima das metas e isolando as partes independentes.

Estas idéias são apresentadas em [Warr 81], que também descreve em linhas gerais um algoritmo para otimização de conjuntos de metas envolvendo conjunções, negações e predicados de segunda ordem (que manipulam conjuntos), mas não disjunções. Este algoritmo foi implementado, formando a base do módulo de otimização. Uma reformulação posterior acrescentou à sua funcionalidade o tratamento de disjunções.

## 3.1 Detalhamento do Algoritmo de Otimização

### 3.1.1 Ordenação das metas

Em cada passo, o processo de otimização reordena as metas de modo que a próxima a ser executada seja aquela que provoque o menor aumento (ou o maior decréscimo) no número de alternativas consideradas.

O algoritmo de ordenação seleciona a meta de menor custo e determina que variáveis serão instanciadas pela sua execução (assumimos que a execução com sucesso de um predicado instancia todas as suas variáveis livres, o que é válido no contexto de uma aplicação de Banco de Dados mas não num programa PROLOG genérico). O processo é então repetido para o restante das metas, após a atualização de seus custos.

Neste contexto uma meta pode ser, além de um predicado, uma comparação, um operador lógico relacionando outras metas ou uma partição (um subconjunto do conjunto de metas cujos elementos compartilham variáveis livres entre si mas não com o restante do conjunto de metas). Estas entidades são chamadas de *meta-predicados*, e o cálculo do custo a elas associado varia conforme a sua estrutura particular.

A função de custo de um predicado associa a cada estado de instanciação de seus argumentos um valor numérico, representando o fator pelo qual a execução do predicado irá multiplicar o número de possibilidades que estão sendo consideradas. Esta função se baseia em informações estatísticas sobre o banco de dados.

O custo de um predicado é tomado como o número de tuplas da relação correspondente ao predicado, dividido pela *variedade* de cada argumento instanciado. O conceito de variedade foi introduzido para facilitar o cálculo da função de custo, e significa o número de valores distintos que um atributo pode assumir nas tuplas que compõem a relação em questão. Note que a variedade de uma relação depende do estado do banco de dados.

O custo de uma comparação é infinito, se algum de seus argumentos não estiver instanciado, ou um número muito pequeno, em caso contrário. Com isso garantimos que não serão realizadas comparações envolvendo variáveis livres, o que geraria um erro de execução. Se os dois

argumentos estiverem instanciados, a comparação se transforma num simples teste.

A negação é tratada de forma semelhante; seu custo é infinito até que os predicados em seu interior não possuam mais variáveis livres como argumentos. A associação deste custo à negação resulta no adiamento de sua execução. Este adiamento é necessário devido a forma pela qual a negação foi implementada na linguagem PROLOG; ela só funciona corretamente se a meta negada não contiver variáveis livres no momento de sua execução. Uma vez que suas variáveis tenham sido instanciadas, o custo da negação é o de sua meta interna de menor custo (da mesma forma que o de uma conjunção ou de uma partição). Já o custo associado a uma disjunção é tomado como o de sua meta interna de maior custo.

O custo de uma meta só precisa ser atualizado quando a última meta selecionada instanciar alguma de suas variáveis livres. O cálculo do custo de um predicado é realizado de forma incremental (se um de seus argumentos for instanciado, basta dividir o seu custo atual pelo tamanho do domínio do argumento). Para tornar a análise mais eficiente, os meta-predicados mantêm uma lista contendo todas as variáveis ainda não instanciadas em seu escopo; com isso podemos determinar rapidamente para todo um conjunto de metas se o seu custo deve ser atualizado ou não.

### 3.1.2 Isolamento de Partes Independentes

Podemos ver um conjunto de metas como um grafo onde os nós representam predicados e meta-predicados e as arestas representam variáveis livres. Duas ou mais arestas podem corresponder a mesma variável.

O ato de selecionar um predicado para execução corresponde a retirada do grafo do nó a ele associado e de todas as arestas associadas as variáveis que ele instancia (alguns meta-predicados, como negações e comparações, nunca instanciam variáveis). A retirada de arestas pode desconectar partes do grafo; os componentes conexos deste grafo representam as partes independentes, ou *partições*, do conjunto de metas.

Chamamos de particionamento ao procedimento de identificar e isolar

as partes independentes de um conjunto de metas, utilizando a abstração de componentes conexos acima descrita. O isolamento de partes independentes da consulta que não compartilhem variáveis livres com a lista de retorno é feito através de sua substituição no conjunto de metas por uma chamada ao predicado **snips**/1, tendo a partição como parâmetro. O predicado **snips**/1 é definido da seguinte forma:

```
snips(Goal):-
        call(Goal),
        !.
```

O ponto de exclamação denota o predicado de controle corte[5]. Sua utilização garante que a partição **Goal** não será avaliada novamente pelo mecanismo de retrocesso durante a avaliação do conjunto de metas. Note que para garantir a obtenção do conjunto completo de respostas, uma partição só será isolada se não for a responsável pela instanciação de uma das variáveis de retorno.

Além de diminuir significativamente o número de retrocessos durante a execução da consulta, o particionamento também diminui a complexidade do próprio algoritmo de otimização porque, ao identificar as partições, permite que elas sejam otimizadas independentemente.

O isolamento de partes independentes é feito dentro do procedimento de ordenação das metas. Após a seleção da meta de menor custo, o restante é imediatamente submetido ao processo de particionamento e agrupado em partições. Uma partição é daí em diante tratada como um único predicado pelo procedimento de ordenação, ficando transparente o fato de que na realidade trata-se de um conjunto de metas.

### 3.1.3 Tratamento de Disjunções

Um dos problemas encontrados na implementação deste módulo foi a generalização do algoritmo original para permitir a manipulação correta de disjunções.

[5] Em inglês: *cut*

A complexidade desta tarefa está relacionada com o fato de que o conjunto de variáveis que serão instanciadas pela disjunção depende do ramo que será executado; em outras palavras, este conjunto só será conhecido *após* a execução da disjunção. Assim, se durante o processo de otimização uma disjunção for selecionada como a meta de mais baixo custo, teremos que atualizar e particionar as metas restantes com base em um conjunto de variáveis instanciadas indeterminado, que só será conhecido em tempo de execução.

Para exemplificar esta situação, vamos imaginar que a disjunção selecionada envolva dois predicados, e que o primeiro instancia a variável A enquanto que o segundo instancia a variável B. Como a disjunção representa caminhos mutuamente exclusivos durante a execução, não podemos tomar nem {A, B} nem {} como o conjunto de variáveis instanciadas.

Tínhamos um problema semelhante relacionado com escopos na fase de análise semântica. A solução adotada foi a ordenação dos operadores de tal forma que o OR fosse sempre o último a ser processado. Não podemos utilizar esta mesma solução neste caso, porque estamos utilizando a ordem como uma informação de controle para o PROLOG; por outro lado, podemos conseguir o mesmo efeito de uma maneira diversa.

O que queremos é que a seleção de uma disjunção deixe o conjunto de metas restante vazio; assim, não teremos mais necessidade de conhecer o conjunto de variáveis por ela instanciadas. Como não é possível mover a disjunção arbitrariamente para o final da lista interna de metas do meta-predicado que a contém, resolvemos então incorporar o conjunto restante à disjunção. Cada ramo da disjunção é combinado com uma cópia do conjunto de metas restante, que é eliminado. Os ramos são então otimizados de forma independente. Note que esta cópia pode ser executada sem problemas, pois cada ramo está num escopo diferente e não afeta os demais.

Um exemplo de otimização envolvendo disjunções pode ser visto na seguinte consulta:

> Mostre o nome dos países que possuam uma fronteira comum e tenham relações comerciais, sendo o exportador europeu ou

o importador sul-americano.

Esta consulta, quando expressa em QUEL, se torna:

```
range of F is fronteira
range of Ex is exporta
range of Eu is europeu
range of Sa is sulAmericano
retrieve (F.pais1, F.pais2) where
F.pais1 = Ex.origem and F.pais2 = Ex.destino and
(Eu.nome = F.pais1 or Sa.nome = F.pais2)
```

Vamos supor um estado do banco de dados no qual as relações **fronteira** e **exporta** tenham um custo de execução muito maior do que o das relações **europeu** e **sulAmericano** (possivelmente devido a um grande número de tuplas). Neste caso a disjunção seria selecionada para execução antes dos demais predicados.

Tornando implícitas as unificações mas não fazendo o tratamento de disjunções acima descrito, teríamos duas formas alternativas de transformar esta consulta num conjunto de metas:

```
answer([P1, P2]) :- (europeu(P1) ; sulAmericano(P2)),
                    fronteira(P1,P2), exporta(P1,P2,_).

answer([P1, P2]) :- (europeu(P1) ; sulAmericano(P2)),
                    exporta(P1,P2,_), fronteira(P1,P2).
```

A diferença entre estas duas formas está na escolha, realizada pelo algoritmo de otimização, de qual predicado deve ser executado imediatamente após a disjunção. Esta escolha depende do cálculo dos custos de execução destes predicados, o que por sua vez depende do estado de suas variáveis (se estão livres ou instanciadas). Mas este estado não está disponível durante o processo de otimização; não temos como saber qual das duas variáveis, **P1** ou **P2**, será instanciada pela execução da disjunção.

Aplicando o tratamento proposto nesta seção, os dois últimos predicados seriam movidos para cada um dos escopos da disjunção e então otimizados separadamente. Supondo que a instanciação de **P1** tenha maior impacto sobre o custo de **exporta** e analogamente para **P2** e **fronteira**, teríamos a seguinte meta:

```
answer([P1, P2]) :- (europeu(P1), exporta(P1,P2,_), fronteira(P1,P2));
                    (sulAmericano(P2)), fronteira(P1,P2), exporta(P1,P2,_)).
```

# 4 Elaboração de Respostas Cooperativas

## 4.1 Uma Teoria Limitada de Cooperatividade

Perguntas em linguagem natural sempre oferecem pistas significativas sobre as crenças e objetivos de seu emissor, além de expressarem um pedido literal de informação. Estas pistas são comunicadas através de uma convenção de cooperação conversacional[Kapl 82] que diz:

*Uma pergunta deve permitir a escolha de uma resposta direta.*

Ou seja, do ponto de vista do questionador, deve haver sempre a possibilidade de mais de uma resposta direta à pergunta formulada. Esta convenção permite ao *respondedor*[6] concluir que o questionador não só não conhece a resposta para a questão como também não conhece nenhuma proposição que forneça uma resposta à questão (ou que implica que nenhuma resposta direta para a questão seja correta). Esta duas últimas inferências podem revelar uma grande quantidade de informação sobre o conhecimento efetivo do questionador.

Cada uma das questões abaixo, quando utilizadas numa conversa cooperativa e interpretadas literalmente, permitem ao ouvinte tirar as seguintes conclusões sobre as crenças do questionador:

**(1a)** Que professor lecionou Cálculo nas férias de verão ?

[6] Em inglês: *respondent*

**(1b)** *Houve um curso de Cálculo nas férias de verão.*

**(2a)** Quantos livros foram emprestados pela biblioteca municipal no último sábado ?

**(2b)** *Algum livro foi emprestado pela biblioteca municipal no último sábado.*

(1b) deve ser verdadeira para que uma resposta direta à questão (1a) seja correta - se nenhum curso de Cálculo foi oferecido nas férias de verão, então nenhum nome fornecido em resposta seria adequado. Portanto, para que o questionador possa perguntar (1a) de forma apropriada ele deve acreditar que (1b) não é falsa, já que não acreditar neste fato violaria a convenção conversacional (nenhuma resposta direta poderia ser correta). Como (1b) é uma pré-condição para a validade de qualquer resposta para (1a), dizemos que ela é uma *pressuposição* de (1a).

Uma generalização do conceito de pressuposição, chamada de *suposição* mantém a propriedade de ter que ser assumida pelo questionador como uma pré-condição para que uma pergunta seja apropriada. Enquanto que a falha de uma pressuposição bloqueia a possibilidade de qualquer resposta correta, a falha de uma suposição implica em que no máximo uma resposta direta à pergunta é potencialmente correta. No exemplo anterior, se (2a) for falsa (isto é, nenhum livro foi emprestado pela biblioteca municipal no sábado) ainda existe uma única resposta correta, "zero".

De acordo com a convenção de cooperatividade, um questionador deve acreditar na validade de todas as pressuposições e suposições subentendidas em sua pergunta; caso contrário, o respondedor não teria opção de *escolher* uma resposta direta.

Do ponto de vista do respondedor, existe uma convenção de cooperação correspondente:

> *O respondedor deve sempre corrigir quaisquer pressuposições que ele acredite serem falsas* (respondendo indiretamente a

questão) *no lugar de dar uma resposta correta e direta, mesmo que ela exista.*

Assim, um respondedor cooperativo pode responder indiretamente à questão (2a) com "Nenhum livro foi emprestado pela biblioteca municipal no sábado", em vez de simplesmente dar uma resposta correta direta "zero". (1a) deveria receber a resposta indireta "Não foram oferecidos cursos de Cálculo nas férias de verão". Note que tais respostas permitem a dedução da resposta direta correta, se ela existir. Estas respostas são chamadas de *respostas corretivas indiretas*, já que elas corrigem as pressuposições falsas do questionador com relação ao domínio e respondem indiretamente à pergunta.

## 4.2 Implementação de uma Interface Cooperativa

As convenções de cooperação acima descritas permanecem válidas quando aplicadas ao diálogo numa linguagem formal, no contexto de uma interface de banco de dados. Neste caso o papel do questionador e do respondedor são fixos, sendo desempenhados respectivamente pelo usuário e pela interface. Dentro das idéias expostas no começo do artigo, estas convenções fazem parte do contexto do diálogo juntamente com o conteúdo do banco de dados (ou, mais precisamente, com a parcela deste conteúdo que é do conhecimento do usuário). Como um todo, este contexto será utilizado para analisar uma consulta que tenha falhado (retornado um conjunto resposta vazio) e identificar uma resposta corretiva indireta apropriada.

A elaboração de uma resposta corretiva indireta é baseada na identificação do *conjunto minimal de falha*, isto é, do conjunto dos menores subconjuntos do resultado do processamento da consulta que fornecem um conjunto vazio de respostas quando avaliado. Falamos aqui de **conjunto** minimal porque podemos ter vários subconjuntos minimais com a mesma cardinalidade. Os elementos deste conjunto minimal representam pressuposições falsas do usuário; a apresentação destes elementos (usando o formalismo QUEL) fornece ao usuário as respostas corretivas indiretas.

A identificação do conjunto minimal de falha é realizada em duas fases. Na primeira, os predicados introduzidos por variáveis de tupla são avaliados individualmente; com isso verificamos se algum deles não representa um domínio vazio, o que é imediatamente informado ao usuário terminando o processo de elaboração de respostas cooperativas.

Na segunda fase, a árvore de condições da consulta - no formato em que é fornecida pela análise sintática - é tratada como um grafo onde as arestas representam os operadores lógicos e os nós são predicados e condições sobre os valores dos campos das relações. Neste contexto um grafo é considerado *conexo* se os seus nós compartilham variáveis livres. Este grafo é percorrido numa variação da busca em largura. Em cada nível desta busca são gerados os subconjuntos conexos do grafo que possuem a cardinalidade do nível atual. Note que somente um conjunto conexo pode ser um conjunto minimal de falha.

Os subconjuntos gerados passam pelos processos de análise semântica e de otimização, sendo então avaliados pelo mecanismo de dedução do PROLOG. Aqueles subconjuntos que tem como resposta um conjunto vazio são convertidos de volta ao formalismo QUEL e apresentados ao usuário, finalizando a segunda fase. Se nenhum subconjunto falhar, passamos ao próximo nível gerando conjuntos de maior cardinalidade. Este processo terminará quando encontrarmos um nível que contenha algum conjunto minimal de falha, ou quando chegarmos ao nível de cardinalidade mais alta (correspondendo a consulta original).

Ao construir os elementos de um nível, partimos sempre dos elementos do nível anterior e tentamos acrescentar um nó. Este processo é feito com o cuidado de que os novos elementos gerados sejam conexos e não redundantes (isto é, dois elementos distintos formados pelos mesmos nós). Os nós introduzidos são tratados diferentemente, conforme o tipo de operador lógico utilizado para conectá-lo ao elemento do nível anterior:

- Um nó conectado por uma *conjunção* faz parte do mesmo escopo dos nós restantes, e portanto pode ser avaliado juntamente com os mesmos;

- Os ramos de uma *disjunção* estão contidos em escopos distintos; portanto, cada um deles é adicionado separadamente ao conjunto do nível anterior e avaliado como uma conjunção. Os resultados desta operação (para cada ramo) são coletados; se algum ramo devolver como resultado um conjunto não vazio, os outros ramos não são analisados e consideramos que o conjunto como um todo não falhou;

- Uma negação inverte o resultado da análise; em outras palavras, nós no escopo de uma negação cuja avaliação resulta num conjunto vazio são descartados, enquanto que aqueles que apresentam ao menos uma resposta são considerados como possíveis pressuposições falsas.

# 5 Construção Interativa de Consultas

Visando facilitar a interação do usuário com o sistema e liberá-lo da necessidade de conhecer o formalismo QUEL, foi construída uma camada sobre a interface anteriormente descrita que permite a construção interativa de uma consulta, abrangendo todas as etapas desta tarefa: carga das relações, definição de variáveis de tupla, seleção das variáveis de retorno, edição da árvore de condições e visualização do resultado da avaliação da consulta.

## 5.1 Visão Geral da Interface

A Figura 1 mostra a interface em seu estado inicial, ou seja, antes da carga dos bancos de dados e definição de variáveis de tupla.

Podemos ver quatro controles do tipo *list box*:

**Relations** apresenta os bancos de dados presentes no diretório corrente que ainda não foram carregados;

**Variables** apresenta as variáveis de tupla definidas e os domínios ao qual estão associadas;

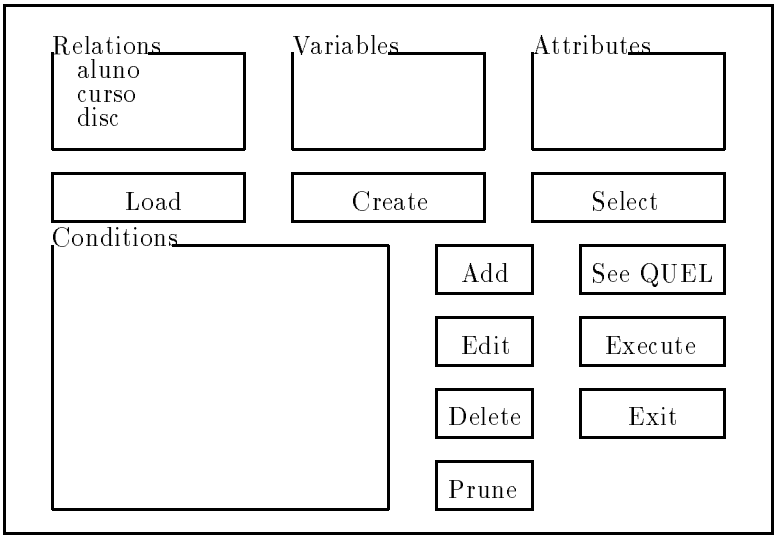


Figura 1: Visão inicial da interface

**Attributes** apresenta as variáveis de retorno (aquelas cujo valor deve ser apresentado ao usuário);

**Conditions** representa graficamente a árvore de condições.

O conteúdo dos três primeiros controles pode ser alterado por meio dos botões a eles associados (respectivamente: **Load**, **Create** e **Select**). Estes botões abrem janelas de edição adaptadas para a funcionalidade de cada controle.

A *list box* **Conditions** tem quatro botões associados: **Add**, **Edit**, **Delete** e **Prune**. Estes botões estão relacionados com a edição da árvore de condições, como veremos adiante.

Os últimos três botões não estão associados diretamente a nenhum controle. O botão **See QUEL** mostra uma construção no formalismo QUEL equivalente à consulta representada pelo estado atual da interface. Dessa forma o usuário pode aprender a sintaxe desta linguagem formal ao mesmo tempo em que usa a interface para construir suas consultas. O botão **Execute** avalia a consulta e abre uma janela para apresentação de seu resultado. O botão **Exit** termina a execução do sistema.

## 5.2 Carga dos Bancos de Dados

Podemos ver na Figura 2 o resultado da ativação do botão **Load**. Foi aberta uma janela apresentando os bancos de dados que podem ser carregados numa *list box* e dois botões, **Load** e **Cancel**.

Os bancos de dados que serão carregados devem ser selecionados dentro da *list box*, posicionando o cursor sobre o nome do banco de dados e pressionando a tecla de espaço[7]. Após a ativação do botão **Load**, a *list box* é atualizada pela remoção dos nomes dos bancos de dados que foram carregados.

[7] A carga de arquivos pode ser feita diretamente na interface original, através do comando *load*

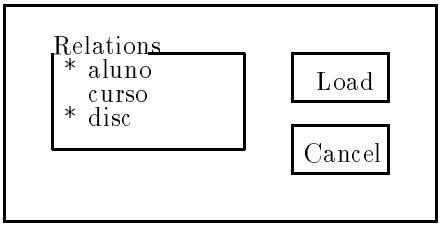


Figura 2: Carga de bancos de dados

### 5.2.1 Formato do Banco de Dados

Cada banco de dados representa uma única relação, sendo implementado por um arquivo do tipo texto com a extensão .QEL. Este arquivo deve estar no seguinte formato:

Deve haver uma linha vazia após o cabeçalho, indicando seu término. Os valores numéricos associados à relação e aos atributos no cabeçalho compõem a informação estatística utilizada no processo de otimização. Esta informação deve ser mantida atualizada por operações que alterem o estado do banco de dados, tais como remoção e inserção de tuplas (estas funções não foram implementadas na versão atual do sistema). Um atributo pode ser do tipo **integer** ou **string**. Os valores associados a cada tupla devem estar em linhas consecutivas, um por linha. Constantes do tipo **string** devem ser colocadas entre aspas simples. O formato não é rígido no que diz respeito ao espaçamento dentro de uma linha (espaços extras são ignorados).

## 5.3 Criação de Variáveis de Tupla

O acionamento do botão **Create** ativa a janela de definição de variáveis. Nesta janela o usuário entra com o nome da variável (ou seleciona o nome de uma variável já definida, para redefini-la) utilizando um controle especial chamado *combo box* descrito mais adiante. A seguir, escolhe o nome de uma das relações presentes na memória na *list box* **Relations**

$<$*nome da relação*$>$ , $<$*número de tuplas*$>$ ( = M)
$< nome\ atributo_1 >$ , $< tipo\ atributo_1 >$ , $< total\ valores\ distintos_1 >$
$< nome\ atributo_2 >$ , $< tipo\ atributo_2 >$ , $< total\ valores\ distintos_2 >$

$\vdots$

$< nome\ atributo_N >$ , $< tipo\ atributo_N >$ ,
$< total\ valores\ distintos_N >$

$< atributo_{1,1} >, < atributo_{1,2} >, \ldots, < atributo_{1,N} >$
$< atributo_{2,1} >, < atributo_{2,2} >, \ldots, < atributo_{2,N} >$

$\vdots$

$< atributo_{M,1} >, < atributo_{M,2} >, \ldots, < atributo_{M,N} >$

e define a variável ativando o botão **Define**. Esta janela é mostrada na Figura 3.

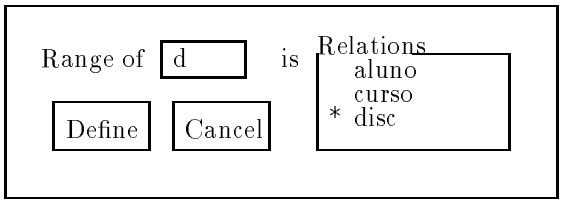


Figura 3: Criação de variáveis de tupla

### 5.4 Seleção das Variáveis de Retorno

O botão **Select** abre a janela de seleção de variáveis de retorno (vide Figura 4). A *list box* **Avaiable** exibe os atributos não selecionados de

cada variável de tupla; um atributo pode ser selecionado através do uso da barra de espaço, após o posicionamento do cursor da *list box* em frente ao seu nome. Uma vez selecionado, o atributo é transferido para o final da list box entitulada **Selected**. Neste controle, um atributo pode ser de-selecionado pelo mesmo processo, retornando para o final da primeira list box.

O botão **Prune** permite o arranjo da ordem dos atributos selecionados na *list box* **Selected**, o que determina a ordem das colunas na apresentação do resultado da consulta. Sua ativação fixa a seleção corrente na *list box* em questão, permitindo seu deslocamento por meio das teclas de movimentação "↓" e "↑". Enquanto a *list box* permanecer neste estado, o botão exibirá o rótulo **Graft**. Uma nova ativação deste botão retornará a *list box* ao seu estado anterior, restaurando o rótulo **Prune**. O botão **OK** aceita as modificações e fecha a janela, voltando a tela principal.

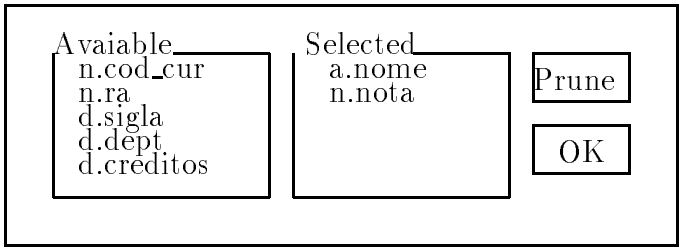


Figura 4: Seleção dos atributos que irão compor a resposta

## 5.5 Edição da Árvore de Consulta

Na Figura 5 vemos a janela de definição de condições de consulta, que pode ser chamada pelos botões **Add** e **Edit** da tela principal. No primeiro caso a janela será ativada num estado padrão; no segundo caso ela será preenchida com o conteúdo da linha corrente da *list box* **Conditi-**

**ons**, que será trocada pelo resultado da edição do usuário. É realizada uma verificação da integridade destas operações, de modo a impedir que a árvore seja colocada num estado inconsistente. Assim, não é permitido transformar um operador lógico que tenha "filhos" numa condição básica, nem acrescentar irmãos à raiz (o que aconteceria se esta fosse ocupada por uma condição básica e ativássemos o botão **Add**).

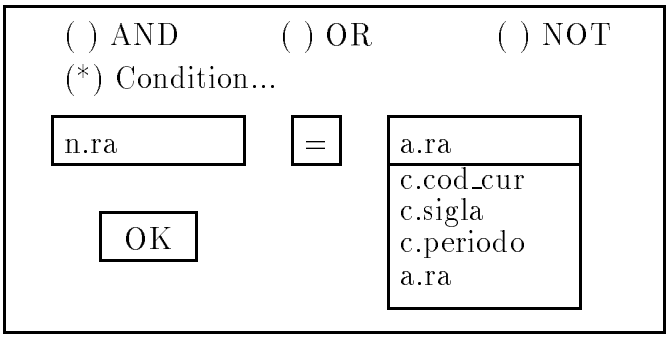


Figura 5: Definição de uma condição de consulta

A primeira opção oferecida ao usuário nesta janela é a escolha de um operador lógico - AND, OR ou NOT - ou de uma condição básica. Neste último caso, o usuário deverá selecionar o nome da primeira variável (dentro do conjunto de atributos das variáveis de tupla definidas), o operador de comparação (um dentre $>, \geq, <, \leq, =$ e $<>$[8]) e o nome da segunda variável ou uma constante. O comando para entrada destes valores foi desenvolvido especialmente para este projeto, e é chamado de *combo box*. Este comando combina a funcionalidade de um campo de edição com a de uma *list box*, pois permite tanto a digitação direta do valor desejado quanto a seleção deste a partir de uma lista. Esta lista fica inicialmente escondida, sendo acionada pela tecla "↓". A partir daí, as teclas de movimentação podem ser utilizadas para selecionar o valor

[8] Este símbolo denota a operação *diferente de*

como numa *list box* comum.

O botão **OK** fecha esta janela, aceitando os valores; o botão **Cancel** faz o mesmo, porém descartando as modificações.

A condição é inserida na *list box* **Conditions**. O comportamento deste controle foi adaptado para exibição de uma árvore, e não de uma simples lista de condições. Assim, a inserção de uma condição será realizada num nível mais interno que o da seleção atual - se esta for um operador lógico - ou no mesmo nível - no caso de uma condição básica. A árvore gerada é n-ária, isto é, um operador lógico pode ter vários filhos. Condições básicas serão sempre folhas.

O botão **Delete** permite a remoção de folhas e sub-árvores inteiras; com isso, podemos ter um operador lógico que não tem filhos. Esta situação é tolerada para que o usuário possa inserir novas condições sob este operador, mas é ignorada na geração da consulta a partir da árvore.

O botão **Prune** tem a mesma funcionalidade descrita na seção anterior, isto é, permite a movimentação da seleção corrente. Neste caso, porém, estamos lidando com uma árvore; temos portanto a restrição de poder movimentar apenas folhas. Por outro lado, esta movimentação pode ser feita tanto na vertical - alterando a ordem dos irmãos dentro de uma sub-árvore - quanto na horizontal - subindo ou descendo o nível do nó, conforme o nível da sub-árvore na qual ele está sendo inserido. Quando no modo de movimentação, o rótulo do botão é alterado para **Graft**. Uma nova ativação deste botão retorna ao modo antigo e restaura o rótulo anterior.

O uso destes botões permite ao usuário uma grande flexibilidade na criação e edição de uma consulta, principalmente no caso de expressões complexas que necessitariam de parentisação e que neste caso tornam-se simples de visualizar devido a sua apresentação na forma de uma árvore. A generalização que permite operadores n-ários torna a árvore mais compacta, e portanto mais fácil de editar e de compreender.

## 5.6 Visualização do Resultado das Consultas

O botão **Execute** monta uma consulta a partir dos valores atuais dos controles da interface e a submete ao gerenciador; o resultado de sua avaliação é apresentado na forma de uma tabela, dentro de uma região de edição de texto (vide Figura 6). Dessa forma podemos ver uma tabela arbitrariamente longa, usando os recursos tradicionais de um editor (página acima, página abaixo, rolamento lateral, busca de palavras, etc.) Acima desta tabela é apresentada a consulta que a originou, no formato QUEL. O botão **OK** fecha a janela e retorna a tela principal.

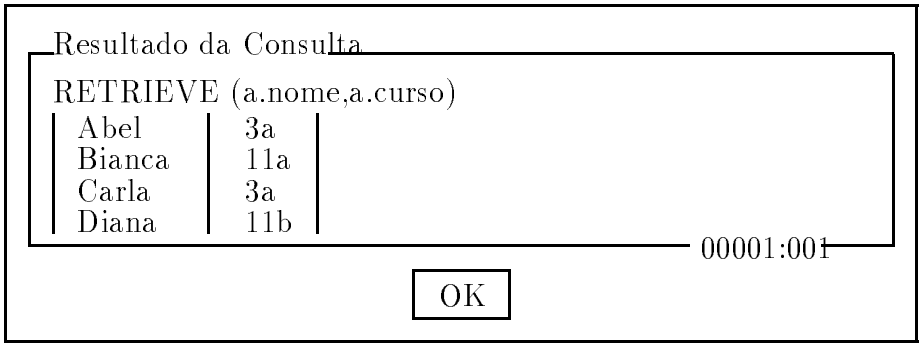


Figura 6: Exibição do resultado de uma consulta

Se durante a análise da consulta for constatado um erro semântico, este é apresentado ao usuário juntamente com o contexto em que foi encontrado. Na Figura 7 vemos uma mensagem decorrente de um conflito de tipos entre duas variáveis presentes na mesma condição básica. Note que o uso da interface impede a ocorrência de erros sintáticos ou o emprego de atributos que não estejam associados a uma determinada variável de tupla.

Para facilitar o aprendizado da sintaxe QUEL pelo usuário, o sistema permite a visualização da consulta atual neste formalismo. Esta funcionalidade está associada ao botão **See QUEL**. Na Figura 8 podemos

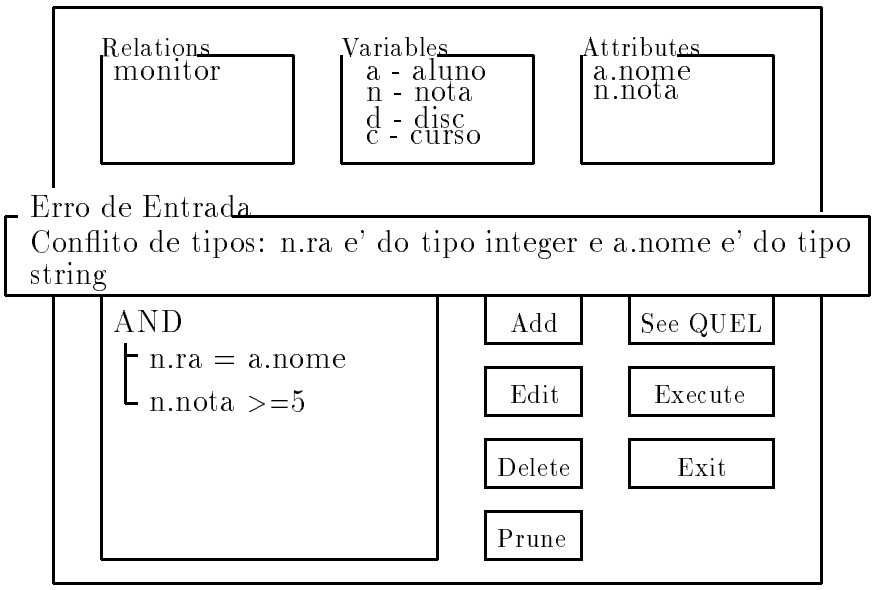


Figura 7: Exemplo de detecção de erro semântico

ver um exemplo de sua utilização: a janela auxiliar entitulada **QUEL command** apresenta uma consulta expressa em QUEL equivalente ao estado atual da interface. Note que é muito mais simples entender a qualificação desta consulta quando expressa como uma árvore de condições do que numa forma textual.

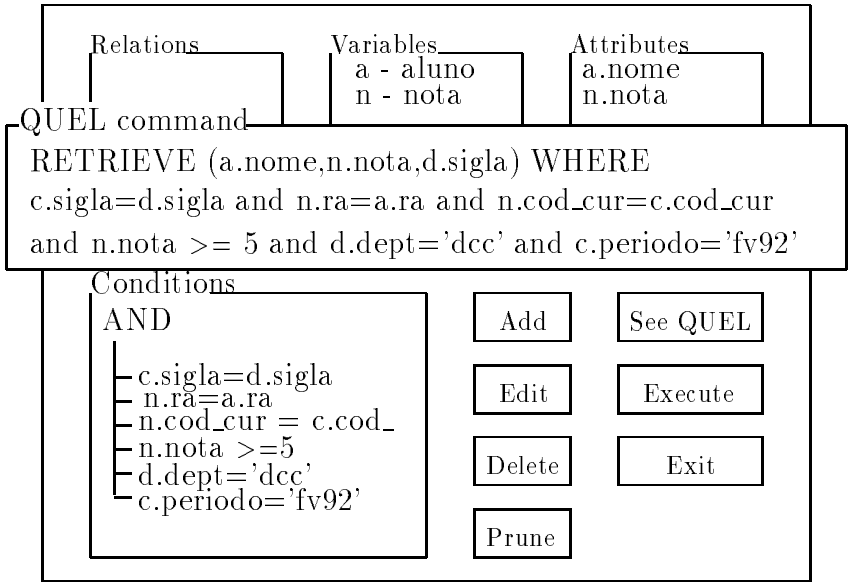


Figura 8: Formato QUEL correspondente a uma consulta

Outro exemplo pode ser visto na Figura 9, que mostra uma consulta envolvendo disjunções e uma conjunção expressa no formalismo QUEL. Este exemplo ilustra outro benefício da geração de consultas através desta interface: a representação da qualificação na forma de uma árvore elimina ambiguidades de escopo de operadores que precisam ser resolvidas por meio de parentisação na representação tradicional.

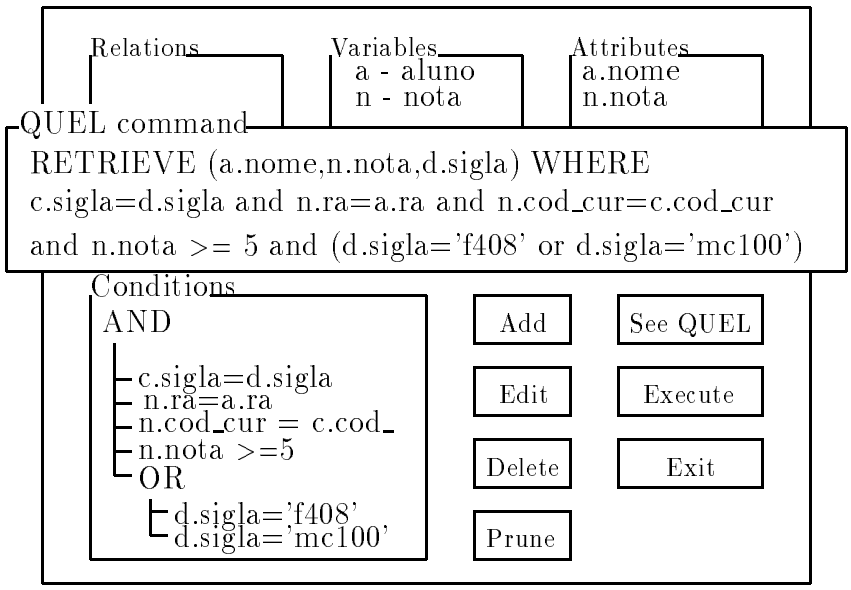


Figura 9: Outro exemplo de uma consulta no formato QUEL

## 5.7 Apresentação de Respostas Indiretas Corretivas

Conforme foi explicado na seção 4.2, uma resposta indireta corretiva é elaborada toda vez que a avaliação de uma consulta gerar um conjunto de respostas vazio. Isto pode ocorrer em dois casos.

Primeiro, quando um domínio associado a um dos termos de projeção da lista de objetivos não contiver nenhum registro. Neste caso não existe nenhuma tupla que possa representar o termo de projeção no conjunto de respostas. Esta situação é detectada pela interface na primeira etapa da elaboração de uma resposta cooperativa. Na Figura 10 podemos ver como esta resposta é apresentada ao usuário.

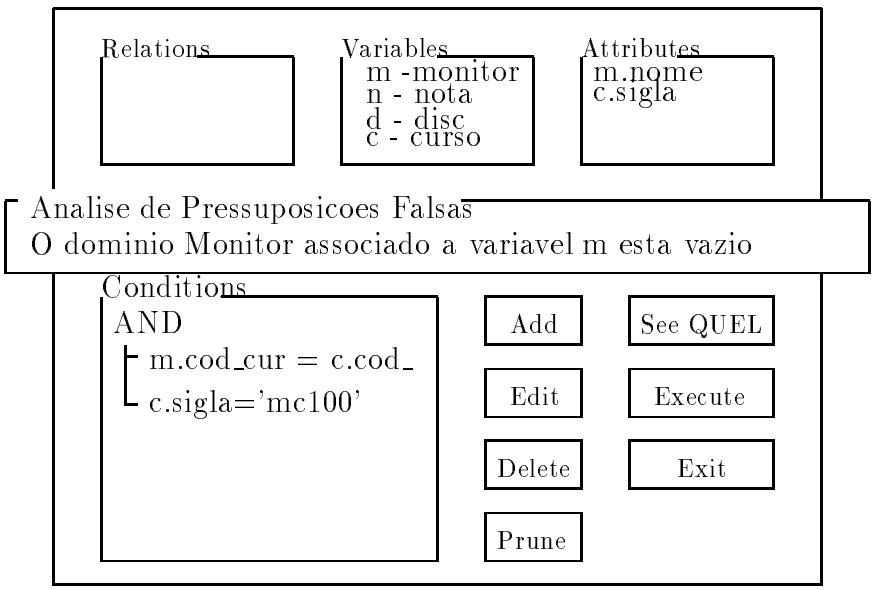


Figura 10: Pressuposição falsa: banco de dados vazio

Segundo, quando um subconjunto das condições que formam a qualificação não puder ser satisfeito por nenhuma combinação das tuplas que

compõem o banco de dados em seu estado atual. Neste caso o subconjunto de menor cardinalidade que não pode ser satisfeito é apresentado ao usuário no formato QUEL. Várias pressuposições falsas podem ser apresentadas simultaneamente, caso elas sejam representadas por subconjuntos de mesma cardinalidade.

Na Figura 11 temos um exemplo deste processo. A interface foi utilizada para montar a seguinte consulta:

> Forneça o nome e a nota dos alunos que foram aprovados em algum curso oferecido nas férias de verão de 1992 pelo Departamento de Ciência da Computação.

O conjunto de respostas resultante da avaliação desta consulta é vazio. O sistema verifica a falsidade de uma de suas pressuposições:

> Foi oferecido um curso nas férias de verão de 1992 pelo Departamento de Ciência da Computação.

Outro exemplo pode ser visto na Figura 12. A consulta representada por este estado da interface é a seguinte:

> Forneça o nome e a nota dos alunos que foram aprovados em algum curso oferecido nas férias de verão de 1992 pelo Departamento de Ciência da Computação ou pelo Departamento de Economia.

Novamente, obtemos um conjunto de respostas vazio. A análise de pressuposições falsas informa o usuário de que a razão da falha desta consulta está na falsidade da pressuposição:

> Foi oferecido um curso nas férias de verão de 1992 pelo Departamento de Ciência da Computação ou pelo Departamento de Economia.

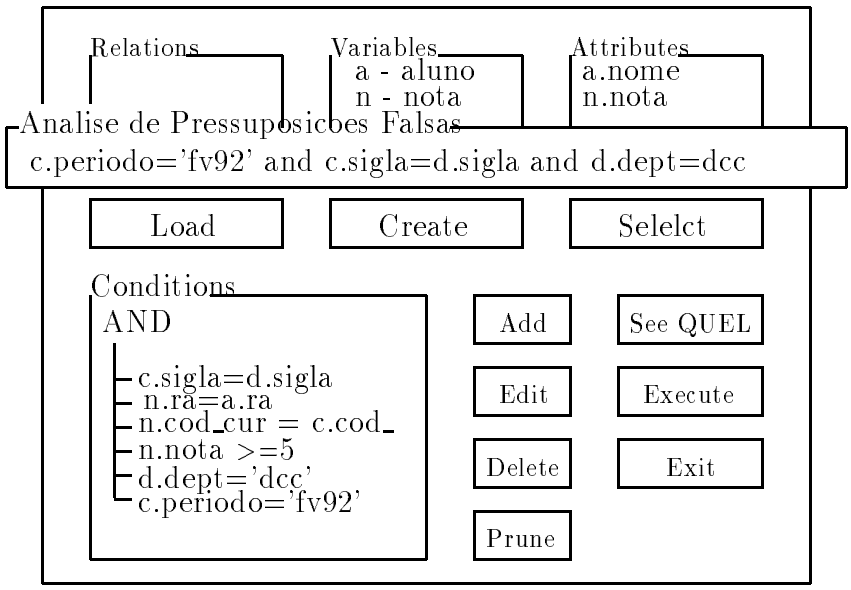


Figura 11: Resposta indireta corretiva

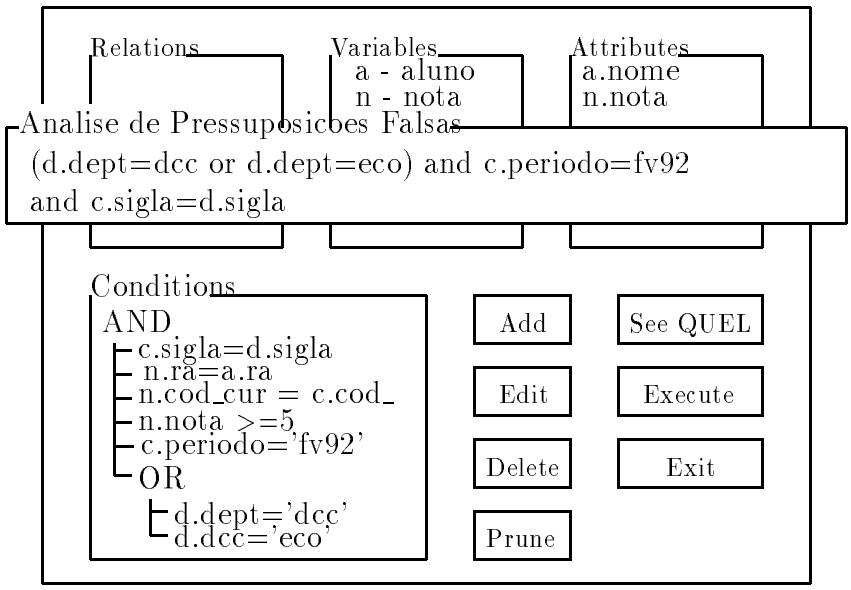


Figura 12: Outro exemplo de resposta indireta corretiva

# 6 Conclusão

Acreditamos que a técnica utilizada neste trabalho para tornar uma interface cooperativa poderia ser de grande utilidade se aplicada a interfaces para bancos de dados não-acadêmicos, pelos seguintes motivos:

- Não é complexa (especialmente quando comparada com a complexidade exigida pelo processamento de linguagem natural). Isto implica que sua implementação não causaria um grande aumento do custo ou do tamanho da interface;
- Não causa impacto significativo na performance da interface. Existem basicamente duas fontes de *overhead* associadas a técnica empregada:
  - A identificação de um conjunto minimal de falha. Este é um processo caro, mas só é necessário no caso da falha de uma consulta (e como não é imprescindível para seu processamento, pode ser desabilitado se o usuário assim o desejar).
  - A manutenção das informações estatísticas associadas às relações. Esta operação é simples e bem definida, podendo ser diretamente associada às operações primitivas de manipulação do banco de dados.
- Contribui para o aumento da produtividade de um usuário do banco de dados, ao identificar prontamente situações de erro conceitual.

Este trabalho poderia ser estendido pela implementação das operações de manipulação do banco de dados que ainda faltam, de modo que uma comparação com outros sistemas fosse mais efetiva. Outras possibilidades de extensão seriam o tratamento de expressões aritméticas na consulta e uma análise mais detalhada da questão da complexidade no processo de identificação de pressuposições falsas.

# A Exemplos

Mostraremos a seguir o resultado das fases de análise semântica e de otimização para algumas consultas. Os exemplos foram selecionados de modo a ilustrar as características mais significativas destas fases. Por motivos de simplicidade, o resultado da fase de análise semântica foi apresentado diretamente na sintaxe da linguagem PROLOG, com a substituição das variáveis não relevantes (que aparecem na cláusula uma única vez) pela variável anônima.

Os predicados utilizados para o cálculo da função de custo nos exemplos abaixo são os seguintes:

```
/* scheme(Relation, List): List e' a lista dos campos
   de Relation associados aos seus respectivos tipos. */

scheme(exemplar, [titulo-string,
                  tombo-number]).
scheme(usuario, [nome-string,
                 ra-number,
                 status-string]).
scheme(emprestimo, [tombo-number,
                    ra-number]).
scheme(reserva, [tombo-number,
                 ra-number]).
scheme(suspenso,[ra-number]).

domainSize (exemplar,titulo,28).
domainSize (exemplar,tombo,35).
domainSize (usuario,nome,10).
domainSize (usuario,ra,10).
domainSize (usuario,status,3).
domainSize (emprestimo,tombo,15).
domainSize (emprestimo,ra,9).
domainSize (reserva,tombo,5).
domainSize (reserva,ra,4).
```

```
domainSize (suspenso,ra,2).

numTuples(exemplar,35).
numTuples(usuario,10).
numTuples(emprestimo,15).
numTuples(reserva,5).
numTuples(suspenso,2).
```

## A.1 Tratamento de Erros de Sintaxe

```
QEL> range of X is exemplar\
     retrieve (X.nome) where X.tombo = 1200
Erro de entrada: nome nao e' um campo valido.


QEL> range of X is aluno\
     retrieve (X.nome) where X.ra = 870840
Erro de entrada: aluno nao e' um dominio conhecido.

QEL> range of U is usuario\
     retrieve (U.nome) where U.ra = 'Nascif'
Conflito de tipos: U.ra e'do tipo number e Nascif
   e' do tipo string
```

## A.2 Negação

### Consulta em linguagem natural

> Quais os livros que não se encontram nem reservados nem emprestados ?

### Entrada em QUEL

```
QEL> range of E is emprestimo\
     range of R is reserva\
```

```
range of X is exemplar\
retrieve (X.titulo) where not(X.tombo = R.tombo) and \
not(X.tombo = E.tombo)
```

### Resultado da Análise Semântica

```
answer(X) :- exemplar(X,Y),
             \+ emprestimo(Y,_),
             \+ reserva(Y,_).
```

O predicado **exemplar** foi inserido antes das negações para garantir o acesso à variável de retorno.

### Resultado da Otimização

```
answer(X) :- exemplar(X,Y),
             snips(\+ emprestimo(Y,_)),
             snips(\+ reserva(Y,_)).
```

Por não possuírem variáveis livres, as negações foram isoladas. Em outras palavras, durante o retrocesso utilizado para recuperar todas as respostas possíveis as negações não serão reavaliadas, passando-se direto a busca de alternativas para o predicado **exemplar**.

## A.3 Auto-junção e isolamento de partes independentes

### Consulta em linguagem natural

> Forneça o nome dos usuários que realizaram empréstimos de ao menos dois livros distintos.

### Entrada em QUEL

```
QEL> range of E1,E2 is emprestimo\
     range of U is usuario\
     retrieve (U.nome) where U.ra = E1.ra and U.ra = E2.ra and\
     E1.tombo <> E2.tombo
```

### Resultado da Análise Semântica

```
answer(X) :- emprestimo(Z,Y),
             emprestimo(W,Y),
             usuario(X,Y,_),
             Z =\= W.
```

As unificações foram tornadas implicítas. Note que duas ocorrências do mesmo predicado tem seus campos associados a variáveis distintas.

### Resultado da Otimização

```
answer(X) :- usuario(X,Y,_),
             snips((emprestimo(Z,Y),
             snips((emprestimo(W,Y),
             snips(Z =\= W))))).
```

O uso do predicado **snips** garante que, mesmo que um usuário possua três ou mais empréstimos, seu nome só será apresentado uma única vez. Isto se deve ao fato de que o retrocesso para geração de novas respostas se dará diretamente sobre o predicado **usuário**, não tentando encontrar alternativas para as instâncias do predicado **empréstimo**.

## A.4 Otimização de disjunções

### Consulta em linguagem natural

> Apresente o título dos livros reservados ou emprestados cujo tombo seja um número entre 1000 e 2000.

### Entrada em QUEL

```
QEL> range of X is exemplar\
     range of E is emprestimo\
     range of R is reserva\
```

```
retrieve (X.titulo) where X.tombo > 1000\
and X.tombo < 2000\
and (X.tombo = E.tombo or X.tombo = R.tombo)
```

**Resultado da Análise Semântica**

```
answer(X) :- exemplar(X,Y),
             Y > 1000,
             Y < 2000,
             (emprestimo(Y,_) ;
              reserva(Y,_)
             ).
```

**Resultado da Otimização**

```
answer(X) :- (reserva(Y,_),
              snips(Y < 2000),
              snips(Y > 1000),
              exemplar(X,Y)
             ) ;
             (emprestimo(Y,_),
              snips(Y < 2000),
              snips(Y > 1000),
              exemplar(X,Y)
             ).
```

Devido ao seu baixo custo, a disjunção foi selecionada para ser executada em primeiro lugar. Os demais predicados foram movidos para dentro de cada um de seus escopos, para que pudessem ser otimizados individualmente.

## Referências


[Arit 87] Arity Corporation. *Using the Arity/Prolog Interpreter and Compiler*. 1987.

[Arit 88] Arity Corporation. *The Arity/Prolog Language Reference Manual*. 1988.

[Cloc 84] Clocksin, W.F. e Mellish, C.S. *Programming in Prolog*. $2^{nd}$ ed. Springer-Verlag, 1984.

[Delo 85] Delobel,C. e Adiba,M. *Relational Database Systems*. North Holland, 1985.

[Kapl 82] Kaplan, S.J. *Cooperative Responses from a Portable Natural Language Query System*. Artificial Intelligence 19, pp.165 - 187, 1982.

[Maie 88] Maier,D. e Warren,D.S. *Computing with Logic - Logic Programming with Prolog*. Benjamin Cummings, 1988.

[Pere 80] Pereira,F.C.N. e Warren,D.H.D. *Definite Clause Grammars for Language Analysis — A Survey of the Formalism and a Comparison with Augmented Transition Networks*. Artificial Intelligence 13, pp. 231–278, 1980.

[Ston 75] Stonebraker, M. *Implementation of Integrity Constraints and Views by Query Modification*. Proc. ACM-SIGMOD Conf., San Jose, 1975.

[Warr 81] Warren, D.H.D. *Efficient Processing of Interactive Relational Database Queries Expressed in Logic*. Proc. $7^{th}$ International Conf. on Very Large Databases. Cannes, France, setembro de 1981.

## Relatórios Técnicos – 1992

01/92 **Applications of Finite Automata Representing Large Vocabularies,** *C. L. Lucchesi, T. Kowaltowski*

02/92 **Point Set Pattern Matching in $d$-Dimensions,** *P. J. de Rezende, D. T. Lee*

03/92 **On the Irrelevance of Edge Orientations on the Acyclic Directed Two Disjoint Paths Problem,** *C. L. Lucchesi, M. C. M. T. Giglio*

04/92 **A Note on Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams,** *W. Jacometti*

05/92 **An $(l, u)$-Transversal Theorem for Bipartite Graphs,** *C. L. Lucchesi, D. H. Younger*

06/92 **Implementing Integrity Control in Active Databases,** *C. B. Medeiros, M. J. Andrade*

07/92 **New Experimental Results For Bipartite Matching,** *J. C. Setubal*

08/92 **Maintaining Integrity Constraints across Versions in a Database,** *C. B. Medeiros, G. Jomier, W. Cellary*

09/92 **On Clique-Complete Graphs,** *C. L. Lucchesi, C. P. Mello, J. L. Szwarcfiter*

10/92 **Examples of Informal but Rigorous Correctness Proofs for Tree Traversing Algorithms,** *T. Kowaltowski*

11/92 **Debugging Aids for Statechart-Based Systems,** *V. G. S. Elias, H. Liesenberg*

12/92 **Browsing and Querying in Object-Oriented Databases,** *J. L. de Oliveira, R. de O. Anido*

## Relatórios Técnicos – 1993

01/93 **Transforming Statecharts into Reactive Systems,** *A. G. Figueiredo Filho, H. Liesenberg*

02/93 **The Hierarchical Ring Protocol: An Efficient Scheme for Reading Replicated Data,** *N. das C. Mendonça, R. de O. Anido*

03/93 **Matching Algorithms for Bipartite Graphs,** *H. A. Baier Saip, C. L. Lucchesi*

04/93 **A lexBFS Algorithm for Proper Interval Graph Recognition,** *C. M. H. de Figueiredo, J. Meidanis, C. P. de Mello*

05/93 **Sistema Gerenciador de Processamento Cooperativo,** *I. M. Carrazana, N. C. Machado, C. C. Guimarães*

*Departamento de Ciência da Computação — IMECC*
*Caixa Postal 6065*
*Universidade Estadual de Campinas*
*13081-970 – Campinas – SP*
*BRASIL*

`reltec@dcc.unicamp.br`