NOVAS FUNCIONALIDADES PARA A LINGUAGEM CPREF-SQL COM SUPORTE A PREFERÊNCIAS DO USUÁRIO

VINÍCIUS V. S. DIAS [1], SANDRA DE AMO [2]

RESUMO

A noção de especificação de preferências do usuário vem conquistando um espaço de grande interesse na comunidade científica. Um dos grandes desafios, sem dúvida, é incorporar esses conceitos em uma linguagem de consulta que possibilite a utilização de todas essas funcionalidades. A linguagem CPref-SQL é um bom exemplo disso, sendo capaz de representar e fazer consultas sobre preferências condicionais. Sendo uma preferência um conjunto de regras, um dos problemas que podem surgir é justamente a questão da consistência das regras, ou seja, é importante garantir que não haja contradições entre elas. Para tanto, foi implementado o Teste de Consistência, que determina se um determinado conjunto de regras é consistente ou não. Este teste consiste de duas etapas, que serão explicadas detalhadamente no decorrer deste artigo. Abordamos também a implementação *On-Top* de uma extensão da CPref-SQL, que adiciona operadores de desigualdade à sintaxe das regras de preferência da linguagem, trazendo para esta maior expressividade. Por fim, exemplos de uso da extensão da linguagem são apresentados para um melhor entendimento do seu funcionamento e recursos.

PALAVRAS CHAVE: preferências condicionais, consultas com preferências, extensão sql, operadores de desigualdade, teste de consistência.

1 Faculdade de Computação, Universidade Federal de Uberlândia, Avenida João Naves de Ávila 2121, Uberlândia-MG, CEP: 38400-902, viniciusvdias@comp.ufu.br

2 Faculdade de Computação, Universidade Federal de Uberlândia, Avenida João Naves de Ávila 2121, Uberlândia-MG, CEP: 38400-902, deamo@ufu.br

ABSTRACT

The notion of specification of user preferences has gained a place of great interest in the scientific community. Some of the big challenges, of course, is to incorporate these concepts in a query language that enables the use of all these features. CPref-SQL language is a good example of that, being able to represent and consult under conditional preferences. One of the problems that can arise when using a set of rules for preference specification is precisely the issue of the consistency of the rules, that is, it is important to ensure there are no contradictions between them. To that end, we implemented the Consistency Test, which decides whether a given set of rules is consistent or not. This test consists of two steps, which are explained in detail throughout this article. We discuss also the On-Top implementation of CPref-SQL, which adds inequality operators to the syntax of the preference rules, providing more expressivity power to the query language. Finally, various examples of the use of the language with the new functionalities are presented for a better understanding of their operation and resources.

KEYWORDS: conditional preferences, preference queries, sql extension, inequality operators, consistency test.

1. INTRODUÇÃO

Diversas aplicações atuais, como no comércio eletrônico, demandam o uso de preferências dos usuários como uma importante fonte de informação para construção de perfis. Essa noção de perfis de usuários é observada em sistemas *Web*, como descrito em [10], em que se torna cada vez mais comum a disponibilização de serviços sensíveis à situação, possibilitando que associemos a determinado perfil uma experiência diferente (no que se refere à interação usuário-sistema). Neste caso, é preciso que haja um mecanismo capaz de representar os perfis de maneira eficiente. Uma maneira de se abstrair o conceito de perfil é através de preferências. Assim, o perfil de um dado usuário pode ser representado através de um conjunto de preferências que expressam sua vontade, mediante situações diversas. Além disso, em aplicações para computação móvel sensíveis a contexto *(context-awareness)*, tema abordado em [5], a especificação de preferências de usuário seria, particularmente, interessante. Sistemas como estes usam informações de contexto, como localização, hora do dia, pessoas ou dispositivos próximos, para apresentar ao usuário, de maneira diferenciada, conteúdos variados. Uma possibilidade para este cenário seria associar preferências para determinados contextos, trazendo assim, a possibilidade de consultas mais personalizadas para ambientes de computação móvel, que é uma tendência tecnológica atual. Como podemos perceber, as possibilidades para sistemas que sejam capazes de trabalhar com preferências são inúmeras e por essa razão é importante que exista uma maneira de automatizar um processo como este. Podemos pensar em preferências de maneira intuitiva: dado um conjunto de itens, quero saber quais aqueles que mais se aproximam da minha vontade, e mais, posso querer organizar esses itens através de um *ranking* que determina uma certa ordem de escolha. Assim, a incorporação desses formalismos em uma linguagem de consulta simples *(SQL: Structured Query Language)* que possibilite o seu uso de maneira eficiente é uma tarefa importante. Mais ainda, podemos pensar em preferências como um conjunto de regras que determinam, de maneira objetiva, quais são as prioridades de um usuário. Trabalhos com esse propósito são encontrados em publicações como [7]**.**

A linguagem CPref-SQL (**C**onditional **Pref**erence **SQL**), que foi apresentada em [9]**,** consiste em uma maneira de trabalhar sobre preferências. Esta linguagem está contida em um contexto de preferências condicionais. Dado um conjunto de itens (ou tuplas), queremos determinar: (1) os itens mais preferidos e/ou (2) os *k* itens preferidos. O último item faz parte de uma tendência atual que é justamente de o usuário poder escolher a quantidade de tuplas

que aparecerão em sua resposta, trabalhos como [12] e [8] tratam exatamente deste assunto. A grande contribuição da linguagem é justamente ter implementado operadores que sejam capazes de calcular essas duas funcionalidades. Muitos conceitos e definições que foram utilizados na elaboração lógica foram retirados de [11] , principalmente quando discutimos a questão da *consistência* de um conjunto de regras.

Diversos trabalhos relacionados demonstram interesse nessa área de preferências. Particularmente em [6], é feita uma proposta muito similar à linguagem CPref-SQL, também estruturada no SGBDR (Sistema Gerenciador de Banco de Dados Relacional) *PostgreSQL* e com modos de implementação parecidos. A principal diferença é que, diferente da CPref-SQL, a lógica em preferências usa operadores chamados *skyline* [2], que consideram apenas as tuplas estritamente dominantes em uma relação. Saindo um pouco do campo da implementação, há o interesse também em se estabelecerem fundamentos teóricos para a modelagem e raciocínio sobre preferências, tendo, portanto, um grande engajamento na área de Inteligência Artificial *(IA)* com a robótica, por exemplo. Isso é verificado em [3] e [11].

Este trabalho está integrado com as propostas do Laboratório de Sistemas de Informação da Faculdade de Computação da UFU (Universidade Federal de Uberlândia), que possui uma equipe de professores, mestrandos e alunos de iniciação científica atuando nas áreas de Preferências e Mineração de Dados. A linguagem CPref-SQL foi desenvolvida e implementada pela equipe deste laboratório que atua na área de preferências. Para maiores informações sobre os integrantes da equipe, publicações relacionadas, demos e download do código da linguagem acesse o site do Laboratório de Sistemas de Informação (LSI/CPref-SQL), no endereço *http://www.lsi.ufu.br/cprefsql*.

Nosso principal objetivo é um formalismo que consiga interpretar, expressar e raciocinar sobre essas preferências, tanto com objetos simples quanto qualquer outra forma estruturada de dados (grafos e árvores).

1.1. Exemplo de Motivação

Considere um banco de dados “Hospedagem” que contém informações sobre opções para hospedagem no Brasil, contendo os seguintes atributos: *Hotel* (nome do hotel) *(H* ou *1), Cidade* (cidade onde fica o hotel) *(C* ou *2), Avaliação* (quantas estrelas possui o hotel, de 1 a 5) *(A* ou *3), Preço* (preço da diária em R$) *(P* ou *4)*, *Distância* (distância até o hotel em km) *(D* ou *5)* e *Finalidade* (motivo da hospedagem, por exemplo, férias ou trabalho) *(F* ou *6)*. Este

modelo de banco será usado também em outras partes deste artigo, principalmente na seção onde serão apresentados exemplos da extensão proposta para a linguagem.

Exemplo 1: Seja uma preferência o seguinte conjunto de regras:

- Se a finalidade da hospedagem for férias, então prefiro os hotéis com 5 estrelas do que os de 4 estrelas, independente do valor dos outros atributos.
- Em geral, se todos os atributos coincidirem, prefiro o hotel Tambau do que o Copacabana Palace.
- Se o hotel é 5 estrelas, então prefiro Belo Horizonte do que Joao Pessoa, contanto que a finalidade da hospedagem seja a mesma.

Considerando esta preferência, desejamos fazer uma consulta do tipo: *"Quais as opções de hospedagem que mais se aproximam da minha preferência?"*. Deste modo, a linguagem CPref-SQL é capaz de retornar essa resposta para o usuário através de uma extensão do SQL padrão. A sintaxe da criação da preferência e da consulta final é a seguinte:

```
CREATE PREFERENCES mypref FROM hospedagem AS
IF finalidade='ferias' THEN avaliacao=5 > avaliacao=4 [H,C,P,D] AND
hotel='Tambau' > hotel='Copacabana Palace AND
IF avaliacao=5 THEN cidade='Belo Horizonte' > cidade='Joao Pessoa' [H,P,D];


SELECT * FROM hospedagem
ACCORDING TO PREFERENCES (mypref);
```

Figura 1 - Sintaxe do exemplo de motivação

Esse exemplo mostra de forma simples qual é o nosso objetivo com a linguagem. Como foi dito na sessão 1, as possibilidades de uso são inúmeras. A consulta final deve retornar as hospedagens preferidas de acordo com o conjunto de regras criado *(mypref)*. É importante frisar que o SQL padrão (versão 3.0) é perfeitamente capaz de realizar consultas como a deste exemplo, fazendo uso de operadores recursivos e simulando a criação de preferências através de *visões* (*CREATE VIEW*). Entretanto, se tratam de consultas grandes, complexas, não-intuitivas e que apresentam um desempenho inferior quando comparado à CPref-SQL, como evidenciado em [1].

Agora, considerando este mesmo exemplo, imagine a situação em que a preferência da segunda regra fosse sobre o atributo *preco* do banco "Hospedagem". Por se tratar de um atributo numérico, seria interessante que fosse possível definir essa preferência através de intervalos e não valores discretos apenas, visto que é mais natural dizer que "um conjunto de preços me satisfaz" do que "especialmente este preço me satisfaz". Com isso, a regra poderia

ser expressa através da sentença::

➢ Em geral, se todos os atributos coincidirem, prefiro hospedagens que custem até 350 reais do que hospedagens que custem mais do que 350 reais.

Ou seja, a nova regra está dizendo basicamente que, qualquer hospedagem com preço menor ou igual a 350 reais, em geral é melhor do que as que custam mais do que isso. Fica claro, portanto, que um recurso como este é importante e contribui na expressividade de qualquer linguagem que envolva banco de dados. Além disso, por se tratar de uma linguagem de preferência declarativa (usuários expressam suas preferências), um processo de verificação de confiabilidade do que está sendo declarado é imprescindível, considerando que um usuário pode se equivocar em algum momento. Questões como estas são o centro deste artigo e de suas contribuições, que serão descritas na próxima sessão.

1.2. Contribuições deste artigo

As principais contribuições deste artigo são centradas nos seguintes itens, que correspondem respectivamente às seções 3, 4 e 5 deste artigo.

a) Implementação do Teste de Consistência que dado um conjunto de regras, determina se este pode ou não ser usado como uma preferência válida;

b) Extensão da linguagem de consultas CPref-SQL (http://www.lsi.ufu.br/cprefsql) com a adição dos seguintes operadores de desigualdade: menor (<), maior (>), menor ou igual (<=) e maior ou igual (>=) na sintaxe das regras de preferência da linguagem, garantindo maior expressividade para a linguagem. Exemplos de uso da linguagem com essas características serão apresentados.

1.3. Organização do artigo

Este artigo está organizado em seções, cada uma com um tópico específico, englobando tanto os trabalhos já feitos quanto as contribuições deste. A seção 1 é a introdução, as demais seguem a seguinte descrição:

Seção 2: A Linguagem CPref-SQL

Nesta parte abordamos características e principais funcionalidades da linguagem CPref-SQL e em geral, de preferências condicionais. Se trata de uma visão geral do que já foi feito em [9] por onde foi baseado todas as contribuições do presente artigo.

Seção 3: Consistência de um Conjunto de Regras de Preferências

Aqui enfatizamos a necessidade de se basear em regras de preferência consistentes e para isso apresentamos os algoritmos que decidem se um conjunto de regras pode ser ou não usado como uma preferência válida.

Seção 4: Extensão da Linguagem CPref-SQL

Nesta seção é feita uma explicação do que significa uma implementação *On-Top* e quais as mudanças que foram necessárias para a implementação dos novos operadores na linguagem.

Seção 5: Exemplos de Consultas usando os Novos Operadores

Dada a CPref-SQL e sua extensão, suportando operadores de igualdade e desigualdade, apresentamos alguns exemplos que mostram o ganho de expressividade da linguagem, desde a criação das regras até a utilização das mesma em uma consulta SQL.

Seção 6: Conclusão

Aqui retomamos os principais pontos do trabalho e fazemos uma reflexão sobre as conclusões tiradas desta experiência.

Seção 7: Agradecimentos

Seção 8: Referências Bibliográficas

2. A LINGUAGEM CPREF-SQL

A linguagem CPref-SQL foi inicialmente proposta em [1] com o objetivo de estender uma linguagem de consultas, no caso o SQL, com operadores que permitam a incorporação de preferências de usuário na resposta. Essas preferências são compostas por um conjunto de regras informadas pelo próprio usuário, do tipo *SE <condições (antecedentes)> ENTÃO <prefiro uma coisa à outra (consequente)>*. Uma observação a se fazer é que as condições são opcionais em uma regra de preferência, o que traz a possibilidade de definição de regras mais gerais, ou seja, sem condições ou antecedentes *(SE .. ENTÃO),* sendo formadas apenas pela preferência em si representadas pelo consequente. Esse modelo pode ser verificado na Figura 1 da sessão 1.1, que apresenta um exemplo de sintaxe real da linguagem CPref-SQL, com criação de uma preferência e, em seguida, uma consulta sobre a mesma.

Antes de detalharmos as principais características da linguagem, cabe a definição de dois conceitos importantes: (1) restrições “fortes” e (2) restrições “fracas”. O primeiro se refere a restrições em uma consulta que tem um poder de filtragem elevado da resposta, ou seja, são condições mais significativas. O segundo se refere ao contrário, onde o grau de

filtragem da resposta de uma consulta é menor. Isso significa que, além das cláusulas básicas do comando *SELECT* (*WHERE, GROUP BY, ORDER BY, ...*), que compõem as restrições "fortes", é possível a utilização da cláusula *ACCORDING TO PREFERENCES* fazendo com que após essas restrições, uma filtragem adicional às tuplas resultantes seja feita baseando-se em regras de preferências caracterizando, assim, as chamadas restrições "fracas" da consulta. Dito isso, o novo plano de execução em alto nível fica da seguinte maneira:

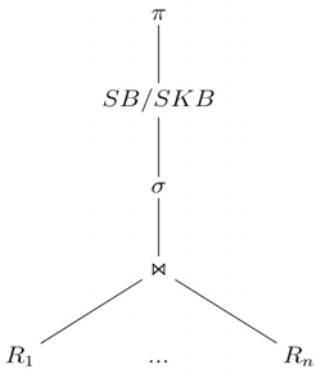

Figura 2 - Plano de execução

Deste modo, primeiro é feita a *junção* das relações envolvidas, depois é feita a seleção da cláusula *WHERE.* Até aqui é exatamente como o esperado em uma consulta SQL padrão, porém, antes de realizar a *projeção* dos atributos desejados, são executados os operadores *Select-Best (SB)* ou *SelectK-Best (SKB)* da cláusula *ACCORDING TO PREFERENCES* que filtram as tuplas que mais se adequam à preferência de usuário usada na consulta. O conjunto de tuplas finais é a resposta retornada pela consulta.

Como o próprio nome sugere, essa linguagem trabalha com as chamadas preferências condicionais (consultas-pc). Resumindo, a linguagem se compromete a, dentre um conjunto de tuplas, calcular:

1. As tuplas mais preferidas: o resultado aqui são as tuplas que podem ser classificadas como dominantes em relação às outras. O conceito de tupla dominante e dominada será melhor detalhado na sessão 2.1. Note que aqui não é considerada nenhuma ordem de apresentação entre as tuplas, apenas selecionamos as que melhores se adequam ao modelo proposto.

2. As tuplas mais preferidas de acordo com um parâmetro *k:* o resultado deste cálculo será as *k* tuplas mais preferidas de acordo com a preferência. Chamadas consultas *top-k*, as tuplas retornadas por esse tipo de consulta são organizadas hierarquicamente de acordo com um *ranking* de preferência. O conceito de *ranking* tem muita utilidade nos meios que utilizam

preferências, quase sempre queremos criar uma ordem do melhor para o pior e basicamente esse é o objetivo de uma consulta deste tipo.

Os operadores que implementam 1 e 2 são, respectivamente, o *Select-Best* e o *SelectK-Best.* Apesar da diferença em relação ao resultado retornado, ambos operadores utilizam uma mesma lógica para o cálculo da dominância entre as tuplas. Essa lógica que determina a relação entre as tuplas é chamada *Ordem de Preferência.*

2.1. Ordem de Preferência

Antes de prosseguir com a definição de ordem de preferência, é importante introduzirmos um conceito que é imprescindível para o entendimento da lógica usada para inferência das tuplas preferidas: *ceteris paribus.* Este termo em latim faz referência aos atributos entre colchetes que aparecem na sintaxe da linguagem CPref-SQL e determina o critério para a comparação de duas tuplas. De forma simplificada, todos os atributos que não aparecem entre colchetes devem coincidir nas duas tuplas para que possamos compará-las, por meio de uma regra.. Este é exatamente o significado desta expressão em latim, que pode ser traduzido literalmente como “todo o resto igual”.

Dadas duas tuplas *t1* e *t2,* dizemos que *t1* é melhor que *t2,* se for possível obter através das regras de preferência, por transitividade ou não, uma dominância entre essas tuplas que indique a preferência de *t1* por *t2*. De uma outra maneira, *t1* é melhor que *t2* se é mais importante a presença de *t1* no resultado de uma consulta do que *t2*. Assim, nosso objetivo é estabelecer exatamente essa relação de dominância entre as tuplas, deste modo, a cada par de tuplas deste conjunto existem duas possibilidades: ou elas são indiferentes entre si, ou seja, não é possível inferir que alguma é melhor do que a outra, ou existe uma relação de dominância entre elas, isto é, uma é preferida à outra. Dado essas preliminares, introduzimos o conceito de tupla *dominante, dominada* e *indiferente.*

Essa relação que tentamos estabelecer entre as tuplas é justamente a *Ordem de Preferência.* De maneira mais formal, é uma relação binária transitiva que determina, dadas duas tuplas *t1* e *t2,* se *t1* é preferida à *t2*, se *t2* é preferida à *t1* ou se *t1* e *t2* são incomparáveis, para tanto, indiferentes entre si. A transitividade desta relação é uma propriedade de extrema importância para a linguagem como um todo, sendo que nem sempre podemos determinar a relação de dominância de duas tuplas diretamente.

Podemos pensar na *relação de dominância* entre tuplas como um grafo direcionado

onde, se existe arco de *t1* para *t2*, então *t1* é preferida à *t2*. Deste modo, se existe um caminho em que eu parto de *t1* e chego a *t2* no grafo, então *t1* é *dominante* e *t2* é *dominada.* Do mesmo modo que, se não existe caminho que ligue *t1* e *t2,* significa que são *incomparáveis.* Na *Figura 3* fica explícito essa noção gráfica da dominância entre tuplas:

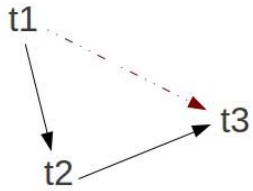

Figura 3 - Grafo de dominância

Podemos observar na *Figura 3* que, por transitividade, *t1* é preferido à *t3* (seta vermelha), pois *t1* é preferido à *t2* e este por sua vez é preferido à *t3*. Como já foi dito, este é o grande trunfo da lógica da linguagem pois, nem sempre fica claro pelas regras quais são as relações de dominância entre as tuplas.

Deste modo, a ideia dos algoritmos que implementam a linguagem é, justamente, percorrer todas as tuplas em questão e calcular a *dominância* entre as mesmas, que é definida para cada par de tuplas consideradas. No final resta a exibição das tuplas escolhidas para a resposta de acordo com o operador usado *(Select-Best* ou *SelectK-Best).*

Portanto, esse é o processo padrão para as funcionalidades da linguagem CPref-SQL, detalhes de trechos específicos de implementação são apresentados nas seções seguintes.

3. CONSISTÊNCIA DE UM CONJUNTO DE REGRAS DE PREFERÊNCIAS

Nesta seção abordaremos as questões envolvidas com a consistência do conjunto de regras de preferência e, consequentemente, o Teste de Consistência. Em primeiro lugar, é preciso deixar claro a importância deste passo no processo de criação de preferências. Já vimos que as preferências são representadas através de um conjunto de regras e que o próprio usuário deve, de alguma forma, expressar essas preferências. Por isso, uma situação possível e indesejada seria a criação de uma preferência pelo usuário que seja contraditória, comprometendo a confiabilidade do resultado.

Dado um conjunto de regras de preferência, pode acontecer indesejavelmente o seguinte. Sejam *r1* e *r2* regras de uma preferência qualquer e suponha que a partir de *r1* inferimos que a tupla *t1* é melhor do que a tupla *t2.* A inconsistência acontece quando a partir

da regra *r2,* por exemplo, chegamos a conclusão de que *t2* é melhor do que *t1.* Aqui fica claro a contradição, afinal, qual das duas tuplas é preferida? Na verdade, nenhuma das respostas está correta. Quando acontece esse tipo de problema, podemos dizer que o conjunto de regras de preferência é inconsistente e não deve ser usado para inferir nada. Deste modo, para garantir a confiabilidade da preferência, antes de sua criação, a linguagem CPref-SQL submete as regras a um *Teste de Consistência.* Reforçamos aqui a necessidade deste teste, pois sem ele, os resultados obtidos através de uma preferência podem não ser confiáveis.

Para ilustrar um cenário como este, considere o banco de dados "Hospedagem", especificado na sessão 1.1. Suponha que um usuário estabeleça sua preferência através de algumas regras:

- Se a cidade é Brasília, então eu prefiro hospedagem 4 estrelas do que 5 estrelas, contanto que a finalidade seja a mesma.
- Em geral, prefiro me hospedar em hotéis 5 estrelas do que 4 estrelas.

Este é um exemplo de conjunto de regras de preferências que é inconsistente, ou seja, é possível inferir a partir dele que uma tupla é *dominante* e *dominada* ao mesmo tempo. Observe que podem existir duas tuplas *t1* e *t2* no banco de dados, instanciadas da seguinte forma:

- *t1 (hotel: Nacional, cidade: Brasília, avaliacao: 4, preco: 460, distancia: 433, finalidade: ferias)*
- *t2 (hotel: Alvorada Towers, cidade: Brasília, avaliacao: 5, preco: 550, distancia: 514, finalidade: ferias)*

Numa situação como esta, não saberíamos dizer qual das hospedagens é melhor, *t1* ou *t2,* pois as tuplas podem ser comparadas pelas duas regras ao mesmo tempo: ambas possuem finalidade férias e ambas são da cidade de Brasília. Por isso, pela primeira regra, *t1* seria melhor do que *t2* (caso em que a preferência está sobre hospedagens 4 estrelas em Brasília). Porém, pela segunda regra *t2* seria melhor que *t1* (caso em que a preferência está sobre hospedagens 5 estrelas, independente de qualquer outro atributo), o que nos leva a um impasse causado pelo que chamamos de inconsistência das regras que foram definidas pelo usuário. Assim, consideramos que nada do que foi inferido está correto, pois partimos de proposições contraditórias.

Enfim, nosso objetivo é explicar os algoritmos envolvidos no *Teste de Consistência,* que irá nos informar se dada preferência (conjunto de regras) poderá ser usada ou não. Ele é

dividido em dois passos: *grafo de dependência* e *teste de consistência local.* Ambas etapas retornam *true* ou *false.* Deste modo, um conjunto de regras é considerado consistente se, e somente se, as duas etapas gerarem resultado *true*, ou seja, o conjunto satisfez os dois testes em questão. A seguir, explicaremos mais detalhadamente os dois passos.

3.1. Grafo de dependência preferencial

Essa etapa do *Teste de Consistência* é a mais simples e por isso é feita em primeiro lugar. Basicamente, construímos o *grafo de dependência* e verificamos se o mesmo é cíclico ou não. Deste modo, de acordo com as definições de dependência preferencial presentes em [11], podemos dizer que o conjunto é consistente *(true)* se o grafo gerado for acíclico, caso contrário inconsistente *(false).*

Input: conjunto de regras de preferência

Output: *true* ou *false*

O *grafo de dependência* é construído percorrendo-se todas as regras do conjunto e, para cada uma:

- para todos os antecedentes da regra (se houver), inserir aresta no grafo (antecedente → consequente).
- todos os atributos que estão entre colchetes *(ceteris paribus)*, são filhos de todos os antecedentes e do consequente da regra.

Uma vez construído o *grafo de dependência*, um algoritmo que detecta ciclos no grafo é executado, retornando assim o resultado para o *Teste de Consistência.* Como já foi dito, se o retorno deste algoritmo for *false,* a segunda etapa não precisa ser verificada, ou seja, já podemos concluir que o conjunto de regras está inconsistente.

Na verdade, o *grafo* está relacionado com a dependência entre os atributos que compõem os predicados das regras, ou seja, antecedentes e consequentes. Neste aspecto é caracterizado bem o termo “preferências condicionais”, em que a decisão de selecionar tuplas para a resposta (nos consequentes das regras) pode variar, dependendo eventualmente de valores dos outros atributos, isto é, de antecedentes. Ainda, as regras podem ser formuladas de modo a ser necessário que um conjunto de condições (antecedentes) seja satisfeito antes de inferirmos alguma preferência, e por isso, o contexto referente aos valores dos atributos das tuplas são relevantes na maioria dos casos. Por exemplo, os atributos das condições descritas nos antecedentes de uma regra sempre terão uma relação de dependência com o atributo do

consequente dessa regra, e isso é denotado por meio das arestas no *grafo de dependência.*

Considere uma relação qualquer *R* que possua três atributos genéricos *(A, B* e *C)* que sejam numéricos. Usaremos *R* nos exemplos 2, 3, 4, 5 e 6 para ilustrar o que é o *grafo de dependência* em um conjunto de regras.

Exemplo 2: Seja o conjunto de regras da *Figura 4* uma preferência sobre a relação *R*. Especificamente neste exemplo, temos uma preferência sobre a relação *R* formada por duas regras. A regra *r1* possui duas condições no antecedente, sobre os atributos A e B de *R* implicando em uma preferência sobre o atributo C. Basicamente, esta regra diz que "Se o valor do atributo A for a1 e o valor do atributo B for b1, então eu prefiro tuplas que possuam c1 como valor de C do que tuplas que possuam c2 neste atributo". A regra *r2* é mais geral, ou seja, não existem condições que devam ser atendidas, apenas uma preferência sobre o atributo C. Ela pode ser representada pela seguinte sentença: "Em geral, independente dos valores dos atributos A e C (*ceteris paribus*) de uma tupla, prefiro as que possuem b1 como valor do atributo B do que as que possuem b3".

```
r1:    SE A=a1 e B=b1 ENTÃO C=c1 > C=c2
r2:    B=b1 > B=b3 [A,C]
```

Figura 4 – Preferência 1 sobre R

Assim, seguindo os dois passos do algoritmo, podemos construir o *grafo de dependência* referente ao conjunto de regras da *Figura 4*, que fica conforme mostrado na *Figura 5.* Referente à *r1*, são adicionadas as arestas (A,C) e (B,C), no sentido antecedente - consequente. Referente à *r2*, são adicionadas as arestas (B,A) e (B,C), no sentido consequente – atributos entre colchetes.

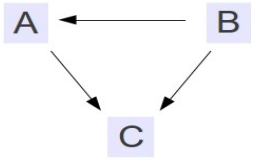

Figura 5 - Grafo de dependência

Exemplo 3: Seja o conjunto de regras da *Figura 6* uma preferência sobre a relação *R*. Neste exemplo, temos uma preferência composta de três regras, nas quais as duas primeiras *(r1* e *r2)* são exatamente as apresentadas no Exemplo 2. A regra *r3* é formada por uma condição no antecedente, sobre o atributo C, implicando em uma preferência sobre o atributo A. Se trata de uma regra onde os antecedentes e consequente são compostos por predicados de

desigualdade, ou seja, representam intervalos de valores possíveis. Podemos representar esta última regra através da seguinte sentença: “Entre as tuplas que possuem um valor estritamente menor do que c3 no atributo C, prefiro as que tenham um valor menor do que a2 em A do que as que possuem valores maiores ou iguais do que a3 neste mesmo atributo, independente de B (*ceteris paribus*)”.

```
r1:    SE A=a1 e B=b1 ENTÃO C=c1 > C=c2
r2:    B=b1 > B=b3 [A,C]
r3:    SE C>c3 ENTÃO A<a2 > A>=a3 [B]
```

Figura 6 – Preferência 2 sobre R

Assim, seguindo o algoritmo, o *grafo de dependência* referente ao conjunto de regras da *Figura 6* é mostrado na *Figura 7.* Referente a *r1*, são adicionadas as arestas (A,C) e (B,C), no sentido antecedente – consequente. Referente à *r2*, são adicionadas as arestas (B,A) e (B,C), no sentido consequente – atributos entre colchetes. Referente à *r3*, são adicionadas as arestas (C,A) no sentido antecedente – consequente, (A,B) no sentido consequente – atributos entre colchetes e (C,B) no sentido antecedente – atributos entre colchetes.

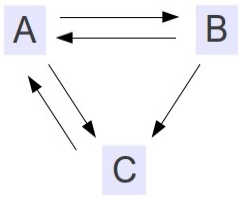

Figura 7 - Grafo de dependência

O conjunto de regras da *Figura 6* é o mesmo da *Figura 4* com a adição de *r3*. Podemos perceber, portanto, que o conjunto de regras de preferência é sensível a qualquer mudança, podendo se tornar inconsistente. Mais uma vez fica claro a importância deste teste para a criação de uma preferência pela linguagem CPref-SQL.

3.2. Consistência Local

A segunda etapa do algoritmo do *Teste de Consistência* diz respeito à *consistência local* do conjunto de regras de preferência. Aqui, o que é levado em consideração são os valores dos atributos localmente, isto é, o domínio de cada atributo da relação. O principal objetivo é, na verdade, garantir que através das regras não seja possível encontrarmos equívocos do tipo: seja *A* um atributo e *a1* um valor, prefiro *a1* a *a1,* o que caracteriza um ciclo no grafo de dominância. Para que isso não aconteça, a preferência é submetida a um

algoritmo, que como no *grafo de dependência,* retorna *true* se o conjunto é consistente localmente e *false,* caso contrário.

Input: conjunto de regras de preferência

Output: *true* ou *false*

Para que um conjunto de regras de preferência seja consistente, este precisa ser consistente em todos os atributos que aparecem nos consequentes das regras, isto é, no lado direito. Por sua vez, para que o conjunto seja consistente para um atributo *x*, devemos fazer o seguinte:

1. Selecionar as regras que possuem o atributo *x* como consequente, chamaremos este de *P′*.

2. Considere também todos os atributos que aparecem como antecedentes das regras de *P′. Deste modo, seja U* o conjunto de todas as instâncias possíveis (valores de tuplas que podem ocorrer) que satisfaçam o antecedente de alguma regra de *P′*.

3. Seja *dom(x),* o domínio do atributo consequente das regras de *P′*.

4. Observe que no consequente de uma regra temos dois termos: o *preferido* e o *não-preferido.* Exemplo: (*A=a1 > A=a2)* → "prefiro *a1* à *a2*" . Chamaremos o consequente preferido de *conseq1* e o não-preferido de *conseq2.*

5. Agora, para um dado elemento *u* do conjunto *U,* seja *P″* o conjunto das regras em que são satisfeitas por *u.* Montaremos um conjunto de tuplas *T* tal que: *(a,b)* pertence a *T* se *a* e *b* são, respectivamente, *conseq1* e *conseq2* de alguma regra de *P″*.

6. Seja a relação *Q* o fecho transitivo de *T.* Consequentemente, se *(a,b)* e *(b,c)* pertencem a *T,* então *(a,b), (b,c)* e ***(a,c)*** pertencerão a *Q.*

7. Assim, dizemos que um conjunto de regras é consistente para um atributo *x* se, para todo elemento de *U,* o conjunto gerado *Q* não contenha uma tupla com elementos iguais (irreflexividade).

O algoritmo foi implementado na função *createPref* e é executada logo depois do *grafo de dependência,* se este não retornar inconsistência. A *Figura 8* mostra um trecho desse algoritmo, que foi implementado em linguagem C.

```
Entrada:        - regras: conjunto de regras de preferência
                - natributos: numero de atributos envolvidos nas regras

Saída:          - true, se o conjunto de regras for compatível
                - false, se o conjunto de regras não for compatível

algoritmo consistencia_local(regras, natributos)
1.      nregras <- tamanho(regras)
2.      for i<-0 to natributos
3.              do      regrasIn <- selectConseq(i, regras, nregras)
4.                      nregrasIn <- tamanho(regrasIn)
5.                      status <- selecionaCompat(regrasIn, 0, natributos, nregrasIn)
6.                      if status = 0
7.                              then    return false
8.      return true
```

Figura 8 - Algoritmo principal do Teste de Consistência Local

Os exemplos seguintes ilustram a ideia da *consistência local* de um conjunto de regras. Mais uma vez ocultamos os atributos *ceteris paribus* por não influenciarem nessa parte.

Exemplo 4: Considere o conjunto de regras de preferência da *Figura 9*. Esta preferência é formada por três regras, duas com condições (antecedentes) e uma geral, ou seja, sem nenhuma condição. A regra *r1* foi descrita no Exemplo 2. A regra *r2* possui uma condição no antecedente, atributo B, que implica em uma preferência sobre o atributo C ("Se o atributo B for igual a b1, então prefiro as tuplas que possuírem c1 como valor de C do que as que forem c3"). A regra *r3* pode ser expressa como: "Em geral, prefiro tuplas que tem c2 em C do que as que tem c1".

```
r1:     SE A=a1 e B=b1 ENTÃO C=c3 > C=c2
r2:     SE B=b1 ENTÃO C=c1 > C=c3
r3:     C=c2 > C=c1
```

Figura 9 – Preferência 3 sobre R

Seguindo os sete passos apresentados acima, e considerando que *u = (A=a1 e B=b1)* seja elemento de *U,* temos o seguinte conjuntos *Q.*

- *Q = {(c3,c2), (c1,c3), (c2,c1), (c1,c2),* ***(c1,c1)****},* para o atributo *C.*

Deste modo, concluímos que esse conjunto de regras não é consistente localmente, pois elas são inconsistentes para em atributo (no caso *C*) . A *Figura 10* mostra o retorno da função *createPref* para este conjunto de regras, sendo rodado diretamente pela interface do *PostgreSQL.*

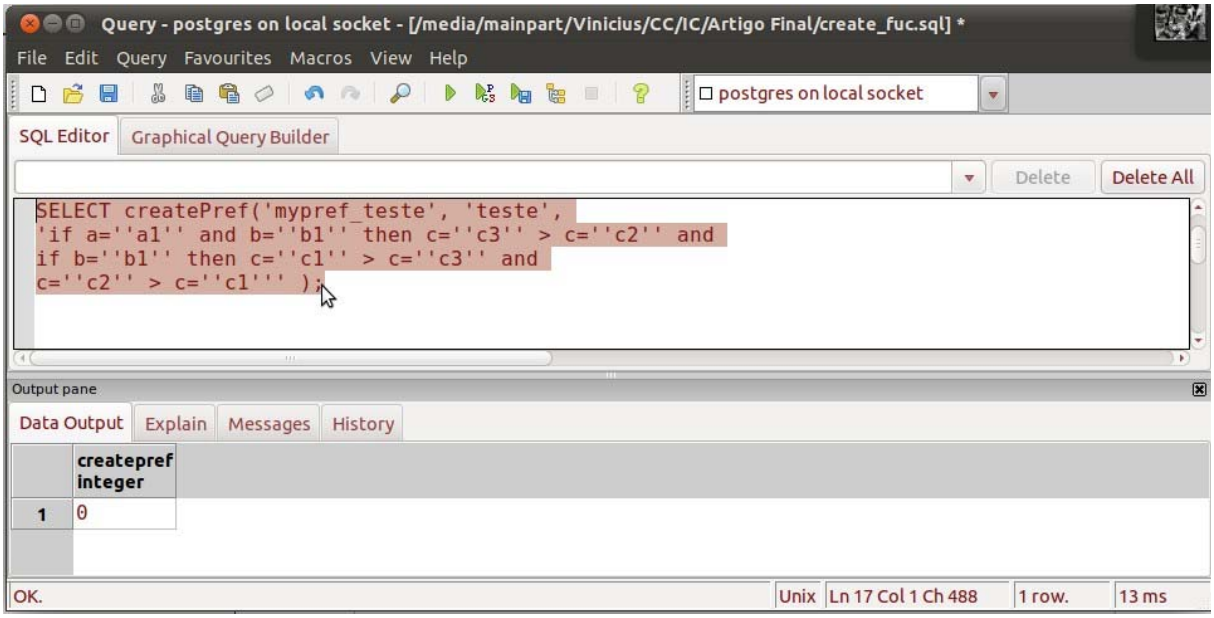

Figura 10 - Retorno 0 (false)

Exemplo 5: Considere o conjunto de regras de preferência da *Figura 11*. Esta preferência é composta por três regras, sendo que as duas primeiras (*r1* e *r2*) foram descritas no Exemplo 4. A regra *r3* pode ser expressa como: "Em geral, prefiro tuplas que tem b2 em B do que as que tem b1".

```
r1:    SE A=a1 e B=b1 ENTÃO C=c3 > C=c2
r2:    SE B=b1 ENTÃO C=c1 > C=c2
r3:    B=b2 > B=b1
```

Figura 11 – Preferência 3 sobre R

Este conjunto de regras será consistente para qualquer elemento do conjunto *U* que escolhermos e para todos os atributos consequentes. Isto é, não é possível achar um contraexemplo para este caso. Assim, essa preferência é consistente localmente. A *Figura 12* mostra o retorno da função *createPref* para este conjunto de regras, sendo rodado diretamente pela interface do *PostgreSQL*.

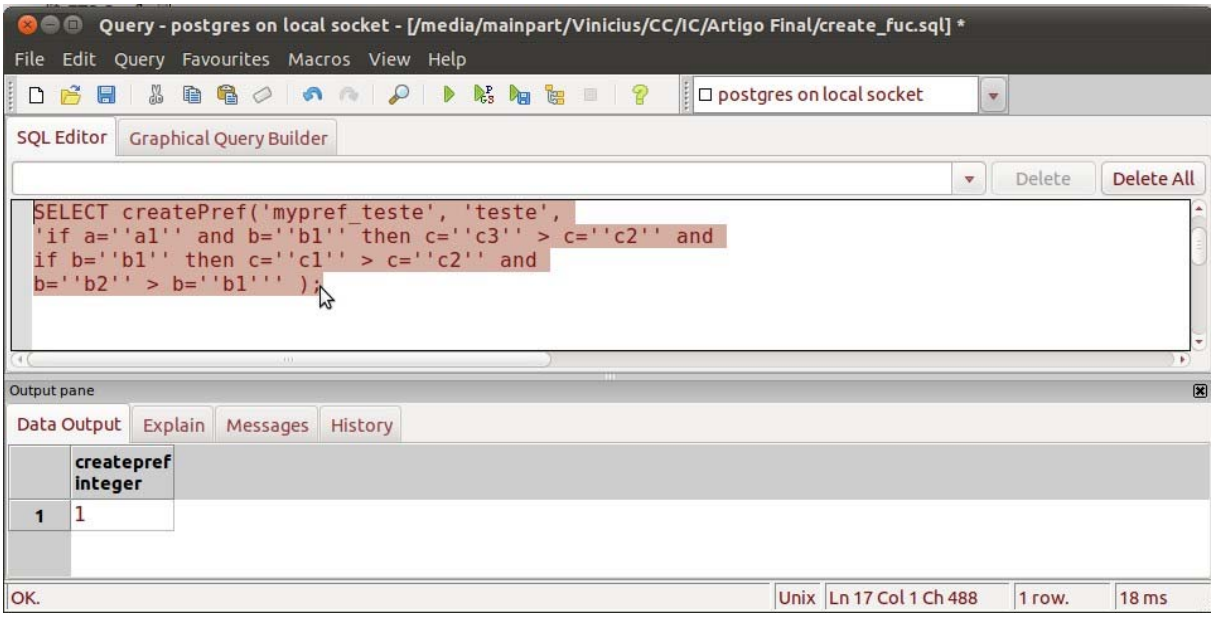

Figura 12 - Retorno 1 (true)

4. EXTENSÃO DA LINGUAGEM CPREF-SQL

Primeiro, é preciso que façamos algumas considerações a respeito de como foi implementada a linguagem CPref-SQL. Vamos abordar aqui a sua implementação *On-Top*. Como já citamos, a linguagem foi estruturada em cima do *PostgreSQL,* que se trata de um SGBDR (Sistema Gerenciador de Banco de Dados Relacional) com código livre. A implementação *On-Top* da linguagem é feita usando os recursos presentes no próprio gerenciador do banco, mais especificamente por funções definidas pelo usuário através de algoritmos escritos em linguagem C e compilados dinamicamente para o uso interno no *PostgreSQL*. Este artifício faz parte das funcionalidades do SGBDR, que está em constante atualização. Por isso, a implementação da linguagem CPref-SQL *On-Top* garante o funcionamento da linguagem independente da versão deste software gerenciador dos dados e essa é uma grande vantagem. Outra maneira de se implementar a linguagem seria modificando o código-fonte do *PostgreSQL* (abordagem *Built-In*), que é uma outra abordagem, diferente do foco deste artigo.

Portanto, podemos relacionar os operadores *Select-Best* e *SelectK-Best* com funções que foram implementadas para a linguagem. Assim, simplificadamente, as funcionalidades da linguagem podem ser divididas em três funções:

1. createPref: responsável por receber um conjunto de regras de preferências. Nesta etapa é necessário realizar a *reescrita das regras* (detalhes a seguir), além de verificar a sua

consistência *(Teste de Consistência)* e depois disso, se sucesso, criar a preferência no catálogo do SGBDR. Ela recebe três parâmetros: o nome desejado para a preferência, as tabelas ou relações envolvidas e as regras propriamente ditas, similar à sintaxe apresentada no exemplo de motivação, porém com os atributos do *ceteris paribus* representados através de números inteiros iniciando com 1 e na ordem com que aparecem na relação. A sintaxe completa para execução no *PostgreSQL* ficaria do seguinte modo:

SELECT createPref(<nome da preferência>,

<lista de tabelas envolvidas>,

<regras de preferência>);

2. mostPref: considerando que as preferências já foram criadas pelo item anterior, essa função usa as regras já reescritas e consistentes para selecionar quais são as tuplas mais preferidas, partindo das tuplas resultantes da seleção "forte". Ela recebe dois parâmetros: o nome da preferência criada no item anterior e a consulta com a seleção "forte", ou seja, o conjunto de tuplas que serão submetidas às preferências. Note a correspondência com a definição do operador *Select-Best*. A sintaxe completa para execução no *PostgreSQL* ficaria do seguinte modo:

SELECT mostPref(<nome da preferência>,

<consulta com a seleção "forte" das tuplas>);

3. topK: como o próprio nome já diz, esta função calcula as *k* tuplas preferidas de acordo com uma preferência criada pelo item 1. Para tanto, recebe três parâmetros: o nome da preferência a ser usada, um inteiro indicando qual será o *k* e, como na *mostPref,* a consulta contendo a seleção "forte" sobre as tuplas da relação. Note a correspondência com a definição do operador *SelectK-Best.* A sintaxe completa para execução no *PostgreSQL* ficaria do seguinte modo:

SELECT topK(<nome da preferência>,

<parametro_k>,

<consulta com a seleção "forte" das tuplas>);

Uma observação importante de se fazer neste ponto é que a sintaxe usada para a execução direta no SGBDR *PostgreSQL* é diferente da sugerida e usada para o usuário. O que deve acontecer em uma aplicação na prática é que exista uma camada intermediária que traduza os comandos, como os apresentados no exemplo de motivação, para a sintaxe utilizada para funções no gerenciador de banco de dados. Deste modo, o usuário da linguagem

não precisará se preocupar com detalhes da implementação da linguagem (*On-Top ou Built-In*), apenas com sua utilidade. Essa é uma característica particular da implementação *On-Top,* que exige essa conversão por questões de padronização com os outros tipos de implementações existentes para a linguagem, bem como para trazer mais clareza na hora da utilização.

Apresentadas essas características gerais e muito importantes da implementação *On-Top,* vamos abordar a principal contribuição deste artigo: a extensão da linguagem CPref-SQL com suporte a operadores de desigualdade. Todas as considerações feitas a partir deste ponto, serão exclusivamente referentes à implementação *On-Top* da linguagem.

Até então, a linguagem CPref-SQL conseguia representar regras de preferência apenas através de um operador de igualdade, ou seja, tanto os antecedentes quanto os consequentes são estruturados através de valores exatos como atributos. Acontece que é muito comum aparecer em um banco de dados atributos que são numéricos e, mais ainda, contínuos de certa forma. A partir disto, podemos querer criar preferências sobre atributos numéricos levando em conta intervalos e não apenas valores exatos. Desta questão surgiu a necessidade de estender a linguagem para trazer mais expressividade, ou seja, englobando mais possibilidades de uso em diversas áreas e contextos diferentes. Assim, essa extensão foi feita usando os mesmos recursos da versão anterior, porém adicionando a possibilidade de se escrever regras e trabalhar com alguns operadores de desigualdade: menor *(<),* maior *(>)*, menor ou igual *(<=)* e maior ou igual *(>=)*.

Apesar da lógica da criação e do cálculo de dominância da linguagem terem continuado iguais, usar intervalos como termos das regras requer alguns cuidados para se garantir o funcionamento correto da CPref-SQL. Uma das modificações é justamente a necessidade da *reescrita de regras.* Para garantir que o *Teste de Consistência* funcione corretamente, antes, precisamos garantir que os intervalos presentes no conjunto de regras para um mesmo atributo sejam disjuntos, ou seja, sem interseção (com exceção à igualdade). Para isso, a primeira coisa após submeter um conjunto de regras para servir como preferência é realizar a *reescrita de regra*, que é feita da seguinte maneira:

1. Transformar os intervalos de cada atributo numérico em intervalos disjuntos.

2. Assim, cada regra que possui um intervalo que foi particionado será substituída por novas regras com os intervalos disjuntos, que representem o original.

3. Repetir o raciocínio para todos os atributos da relação.

A seguir, temos um exemplo que mostra a metodologia da *reescrita de regras*, seguindo os três passos acima.

Exemplo 6: Considere o conjunto de regras de preferência da *Figura 13* sobre a relação *R (A, B* e *C)*. Nesta etapa, a título de exemplificação, ocultaremos os atributos *ceteris paribus* das regras, pois elas não interferem na reescrita.

```
r1:    SE 0<=A<=5 ENTÃO B=b1 > B=b2
r2:    SE 3<A<=5 e B=b3 ENTÃO C=c2 > C=c1
```

Figura 13 – Preferência 3 sobre R

Em *r1* e *r2* da *Figura 13,* existem os seguintes intervalos referentes ao atributo *A:* [0,5] e (3,5]. Esses dois intervalos possuem intersecção e por isso as regras em que estão presentes devem ser reescritas. O primeiro passo é particionar esse intervalo em outros que sejam disjuntos entre si. Assim, podemos transformar esses dois intervalos nos seguintes: [0,3] e (3,5]. Observe que, os novos intervalos não possuem intersecção, [0,5] = [0,3] U (3,5] e (3,5] = (3,5]. A partir dos novos iremos reescrever as regras de modo que o novo conjunto represente exatamente o original. O conjunto final de regras resultantes do algoritmo é mostrado na *Figura 14*.

```
r1:    SE 0<=A<=3 ENTÃO B=b1 > B=b2
r2:    SE 3<A<=5 ENTÃO B=b1 > B=b2
r3:    SE 3<A<=5 e B=b3 ENTÃO C=c2 > C=c1
```

Figura 14 – Conjunto de regras após a reescrita

Essa etapa de reescrita ocorre na criação da preferência (*createPref*) e após ela que acontece o *Teste de Consistência.*

Outra mudança implementada nesta versão foi a maneira de se comparar termos em uma regra, isto é, houve a necessidade de criar uma maneira de determinar se o valor de um atributo satisfaz ou não um determinado intervalo. Para esse propósito foi criado um algoritmo que verifica se um valor satisfaz ou não determinado intervalo e toda comparação feita no processo de inferência da *ordem de preferência* é feita através dele. A *Figura 15* mostra este algoritmo, implementado em linguagem C. A ideia do algoritmo é simples, basicamente ele verifica se existe intersecção entre os dois termos, se sim, significa que satisfaz, senão, não satisfaz.

```
Entrada:        - p1, predicado de intervalo ou igualdade
                - p2, predicado de intervalo ou igualdade

Saída:          - true, se p1 satisfazer p2
                - false, se p1 não satisfazer p2

algoritmo satisfaz(p1, p2)
1.      if p_igualdade(p1) and p_igualdade(p2)
2.              then    if p1.valor1 = p2.valor1
3.                              then    return true
4.                              else    return false
5.      if p1.valor1 < p2.valor2
6.              then    if p_igualdade(p1)
7.                              then    a <- p1.valor1
8.                              else    a <- p1.valor2
9.                      b <- p2.valor1
10.                     tr <- intervalo_direito(p1)
11.                     tl <- intervalo_esquerdo(p2)
12.             else    if p_igualdade(p2)
13.                             then    a <- p2.valor1
14.                             else    a <- p2.valor2
16.                     b <- p1.valor1
17.                     tr <- intervalo_direito(p2)
18.                     tl <- intervalo_esquerdo(p1)
19.     if a > b or ( a=b and i_fechado(tr) and i_fechado(tl) )
20.             then    return true
21.             else    return false
```

Figura 15 – Algoritmo usado na comparação de intervalos (predicados)

Essas foram as principais contribuições para a linguagem CPref-SQL. A partir de agora é possível representar regras com termos contendo intervalos para atributos numéricos contínuos, trazendo assim mais usabilidade para a linguagem. Para ficar claro isso, na próxima seção serão apresentados exemplos que ilustram toda a utilização da linguagem na prática e os benefícios dessa extensão proposta para a linguagem.

5. EXEMPLOS DE CONSULTAS USANDO OS NOVOS OPERADORES

Os exemplos apresentados a seguir são referentes ao banco de dados de “Hospedagem” cuja definição está na seção *1.1.* Para o entendimento dos resultados, considere que o banco está povoado com os dados da *Tabela 1.*

Hotel	Cidade	Avaliação	Preço	Distância	Finalidade
Copacabana Palace	Rio de Janeiro	5	600	992	ferias
Tambau	Joao Pessoa	5	260	2730	ferias
Royal Jardins Boutique	Sao Paulo	4	300	605	trabalho
Belo Horizonte Plaza	Belo Horizonte	5	234	556	trabalho

Ouro Minas Palace	Belo Horizonte	4	234	556	ferias
Royal Jardins Boutique	Sao Paulo	4	260	605	ferias
Nacional	Brasilia	4	460	433	ferias

Tabela 1 - Dados do banco de dados "Hospedagem"

Exemplo 7: Considere o conjunto de regras abaixo. A criação da preferência segue o modelo da *Figura 16.*

- Se a cidade é Belo Horizonte, então eu prefiro hospedagem 5 estrelas do que 4 estrelas, contanto que o preço e a distancia sejam as mesmas.
- Se a distancia for maior do que 600km e considerando hotéis com mesma avaliação, então prefiro diárias abaixo de R$500,00.
- Em geral, prefiro me hospedar em lugares a menos de 700km de distância.

No primeiro item temos uma regra de preferência com uma condição no antecedente (atributo *cidade*) e um consequente (atributo *avaliacao*). A restrição de que a regra é aplicável desde que *preco* e *distancia* sejam as mesmas é mapeada na sintaxe da linguagem entre colchetes, referentes ao *ceteris paribus:* os atributos 1 e 6 (*hotel* e *finalidade*) são os únicos indiferentes, o resto deve ser o mesmo (*preco* e *distancia*). No segundo item temos uma regra com predicados de intervalo, um antecedente sobre *distancia* e o consequente sobre *preco.* Há uma restrição de que a regra se aplica entre hotéis com a mesma *avaliacao,* assim, todo o restante (1, 2, 6 ; *hotel, cidade, finalidade*) é indiferente. No terceiro item temos uma regra geral, com consequente sobre *distancia*, sem nenhuma restrição e portanto, demais atributos são indiferentes e aparecem entre colchetes.

```
CREATE PREFERENCES mypref_hospedagem_1 FROM hospedagem AS
IF cidade='Belo Horizonte' THEN avaliacao=4 > avaliacao=5 [1,6] AND
IF distancia>600 THEN preco<500 > preco>=500 [1,2,6] AND
distancia<700 > distancia>=700 [1,2,4,6];
```

Figura 16 – Comando para createPref

A sintaxe e resultados no *PostgreSQL* são mostrados na *Figura 17.* Note que o retorno da função na parte inferior da janela é 1 *(true),* ou seja, o conjunto de regras é consistente.

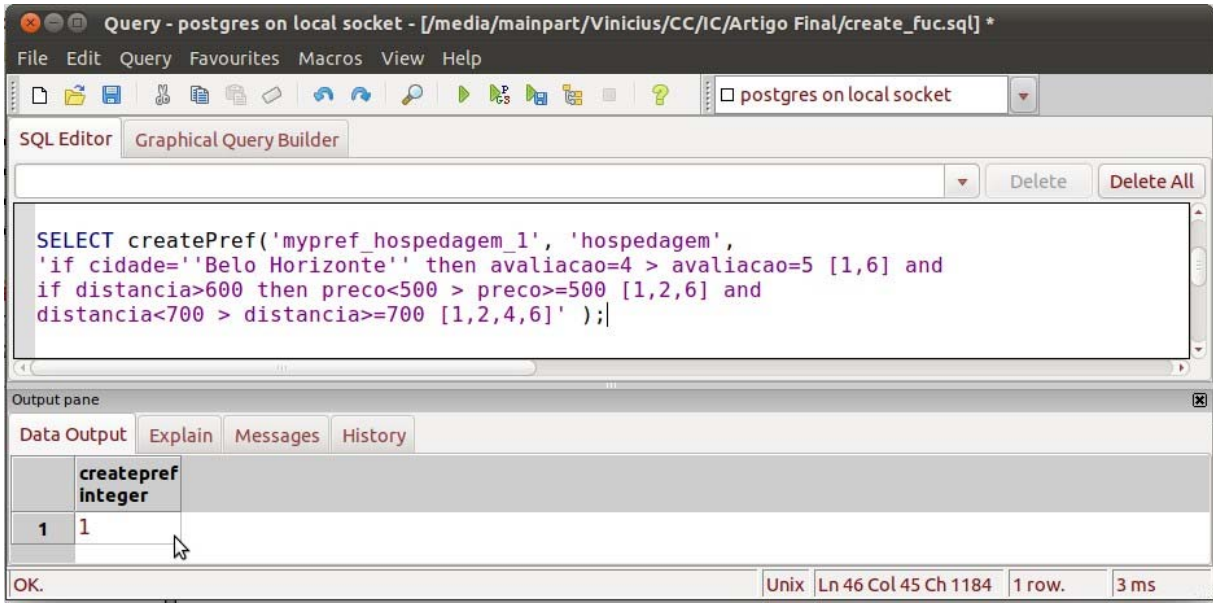

Figura 17 - Criando a preferência "mypref hospedagem 1"

Após a criação desta preferência, queremos saber quais são as tuplas mais preferidas, através do comando apresentado na *Figura 18.*

```
SELECT *
FROM hospedagem
ACCORDING TO PREFERENCES (mypref_hospedagem_1);
```

Figura 18 – Comando para mostpref

O resultado dessa consulta e a sintaxe no *PostgreSQL* são mostrados na *Figura 19.*

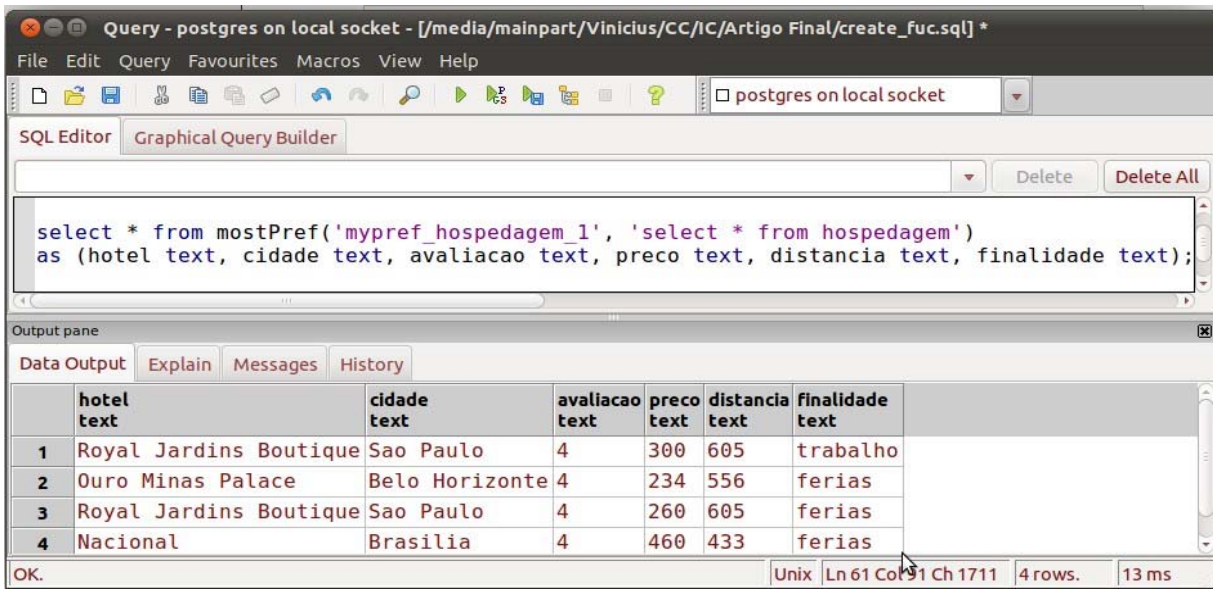

Figura 19 - Tuplas mais preferidas

Uma consulta do tipo *top-k* também pode ser feita, como por exemplo, se desejarmos saber quais as três melhores hospedagens de acordo com essa preferência. Comando na

Figura 20 e resultados (mais sintaxe no *PostgreSQL*) na *Figura 21*.

```
SELECT *
FROM hospedagem
ACCORDING TO PREFERENCES (mypref_hospedagem_1, 3);
```

Figura 20 – Comando para topK

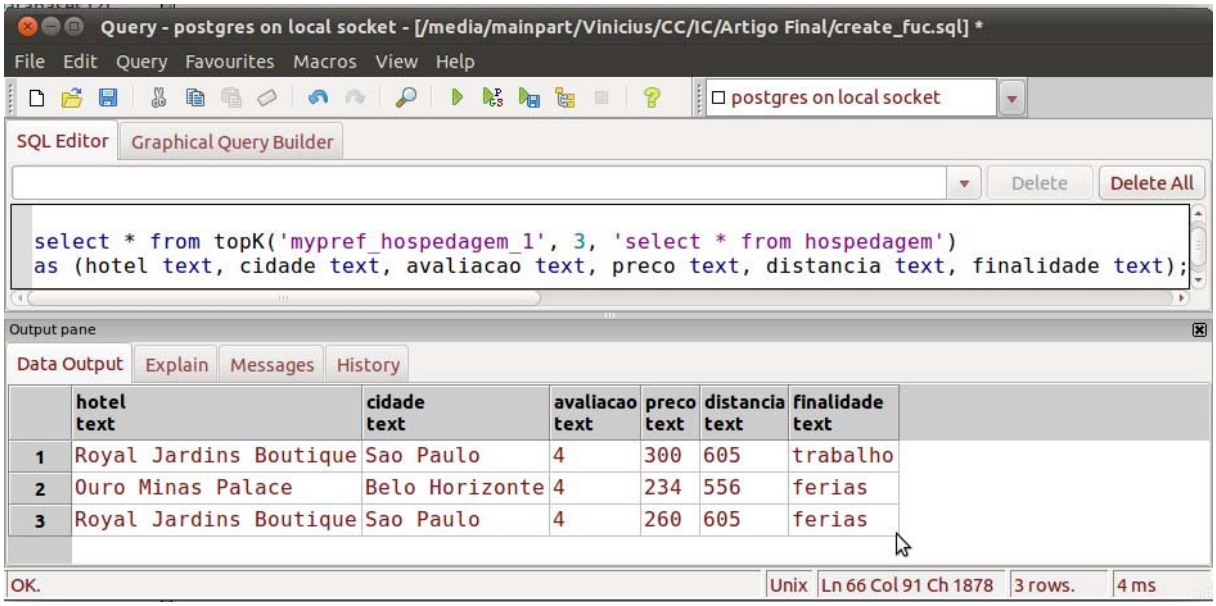

Figura 21 - Top-3 tuplas preferidas

Exemplo 8: Considere o conjunto de regras abaixo. A criação da preferência segue o modelo da *Figura 22*.

➢ Se a distância for menor do que 500km, então prefiro hospedagens que me custem menos de R$250,00 a diária, contanto que a finalidade da viagem seja a mesma.

➢ Em geral, considerando mesmos preço e avaliação do hotel, prefiro estar de férias do que a trabalho.

➢ Se o preço da diária for maior do que R$400,00, então hospedagens 5 estrelas do que 4 estrelas, contanto que a distancia e finalidade sejam iguais.

No primeiro item temos uma regra com predicados de desigualdade, formada por uma condição no antecedente sobre *distancia* e o consequente sobre *preco.* A restrição de que a regra se aplica no contexto em que a *finalidade* é a mesma é resolvida adicionando-se todos os demais atributos, exceto *finalidade* (número 6), nos colchetes. No segundo item temos uma regra geral, com consequente sobre *finalidade* e restrição de que *preco* e *avaliacao* devem ser iguais. Deste modo, adicionamos o restante dos atributos (1, 2, 5 ; *hotel, cidade, distancia*) entre colchetes. No terceiro item temos uma regra com antecedente sobre *preco* e consequente sobre *avaliacao*. Como restrição, é preciso que *distancia* e *finalidade* sejam iguais. Por isso,

hotel e *cidade* (1, 2) devem aparecer entre colchetes, pois são os únicos atributos indiferentes na regra.

```
CREATE PREFERENCES mypref_hospedagem_2 FROM hospedagem AS
IF distancia>500 THEN preco<250 > preco>=250 [1,2,3] AND
finalidade='ferias' > finalidade='trabalho' [1,2,5] AND
IF preco>400 THEN avaliacao=5 > avaliacao=4 [1,2];
```

Figura 22 – Comando para createPref

A sintaxe e resultados no *PostgreSQL* são mostrados na *Figura 23.* Note que o retorno da função na parte inferior da janela é 1 *(true),* ou seja, o conjunto de regras é consistente.

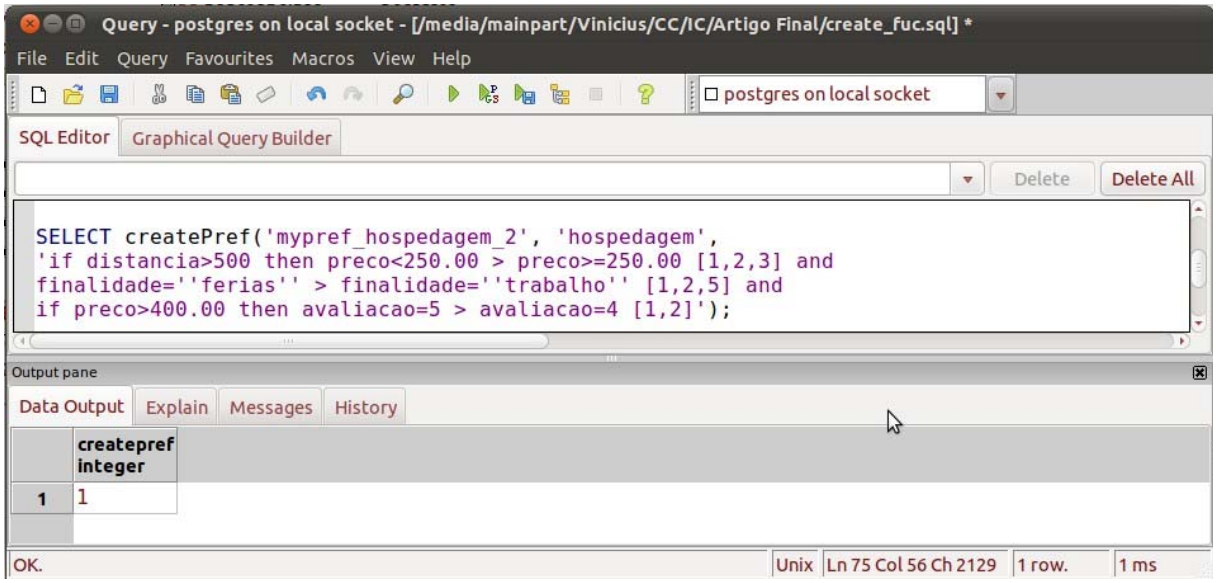

Figura 23 - Criando a preferência "mypref_hospedagem_2"

Após a criação desta preferência, podemos fazer uma consulta do tipo: "*dentre as hospedagens que têm como finalidade férias, quais são as minhas tuplas mais preferidas?".* O comando para execução dessa consulta fica como na *Figura 24.* O resultado e sintaxe no *PostgreSQL* estão na *Figura 25.*

```
SELECT *
FROM hospedagem
WHERE finalidade='ferias'
ACCORDING TO PREFERENCES (mypref_hospedagem_2);
```

Figura 24 – Comando para mostpref

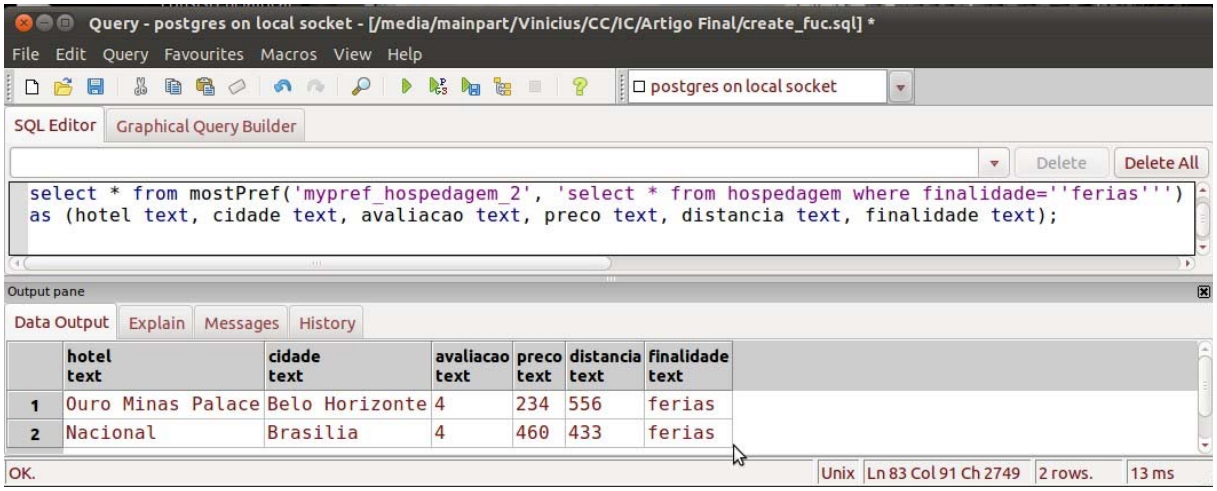

Figura 25 – Tuplas mais preferidas dentre as que tem "férias" como finalidade

Outra possibilidade é querermos obter as quatro melhores hospedagens possíveis no banco. Para isso, realizamos uma consulta *top-k* com o comando apresentado na *Figura 26.* Os resultados e a sintaxe no *PostgreSQL* estão na *Figura 27.*

```
SELECT *
FROM hospedagem
ACCORDING TO PREFERENCES (mypref_hospedagem_2, 4);
```

Figura 26 – Comando para topK

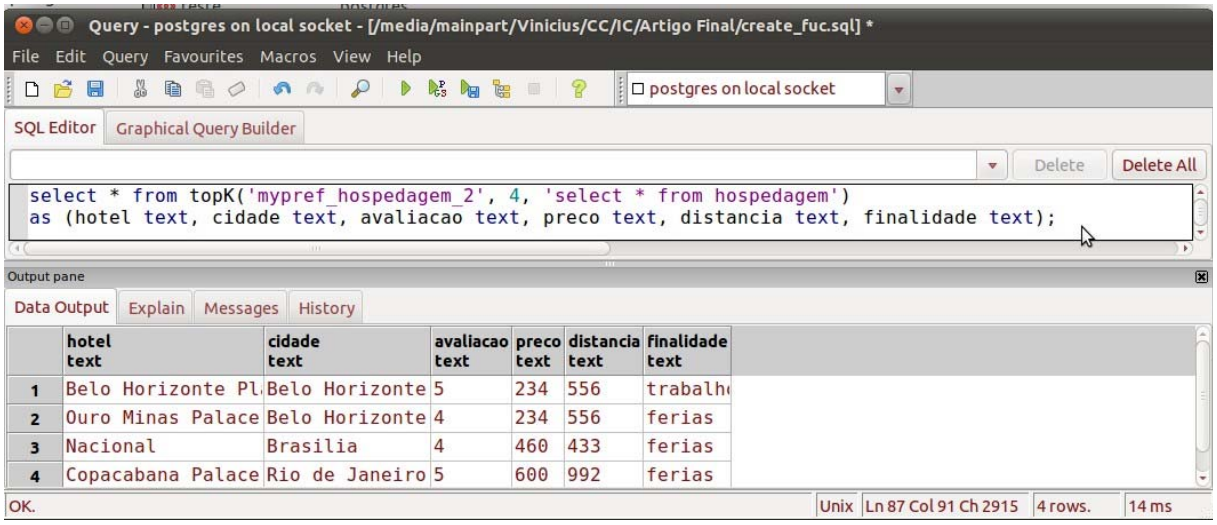

Figura 27 - Top-4 tuplas preferidas

Duas observações podem ser tiradas desses dois exemplos: (1) A sintaxe usada pelo usuário da linguagem não envolve funções, os comandos no *PostgreSQL* não são visualizados em hora nenhuma, foram colocados apenas com o intuito de entendermos o funcionamento interno; (2) podemos fazer seleções na cláusula *WHERE* antes de aplicarmos alguma preferência, como é mostrado no *exemplo 8.*

6. CONCLUSÃO

Há uma grande necessidade de se automatizar o processo com que representamos e raciocinamos em termos de preferência de usuário. Para este propósito foi desenvolvida a linguagem CPref-SQL dentro da equipe de Banco de Dados do Laboratório de Sistemas de Informação (Faculdade de Computação – UFU) que foi implementada com o intuito de fornecer a extensão da linguagem de consulta padrão SQL que suporte esse tipo de funcionalidade, isto é, que suporte a especificação de preferências do usuário, que serão utilizadas pelo processador de consultas de CPref-SQL para personalizar as respostas das consultas feitas pelos usuários. Em complemento a esse trabalho, surgiu a necessidade de incorporar operadores de desigualdade nas regras de preferências, utilizando uma abordagem de implementação *On-Top.* Através dessa contribuição, a adição de operadores pôde tornar a linguagem mais intuitiva e independente do versionamento do SGBDR (*PostgreSQL*), podendo assim ser utilizada de inúmeras formas em diferentes contextos. Consequentemente, essa extensão exigiu alguns ajustes em relação à versão anterior, como a necessidade da *reescrita das regras* e a mudança na maneira de comparar os termos no cálculo da *ordem de preferência.* Apesar da implementação ter sofrido muitas modificações, a lógica continuou a mesma, apenas mais refinada com a adição de alguns conceitos. Os exemplos apresentados na seção 5 ilustram o funcionamento e possibilidades da linguagem na prática.

Outro ponto abordado foi a questão da consistência de um conjunto de regras. Já foi dito que as preferências podem ser representadas através de regras que determinam, em baixo nível, a vontade de um determinado usuário. Porém, isso não basta, para que o resultado de uma consulta baseada em preferências seja considerada confiável, devemos garantir que as regras não se contradigam. Esse é um erro que é comum pois o usuário final geralmente não tem ideia de que isso possa acontecer, comprometendo assim os resultados. Para evitar esse tipo de problema existe o *Teste de Consistência*, filtrando assim, apenas as preferências que possuam regras consistentes entre si, garantindo assim sempre a confiabilidade da resposta.

Portanto, o que foi dito neste artigo aborda bem o funcionamento e lógica da linguagem CPref-SQL, sendo que as contribuições apresentadas no mesmo ajudaram a tornar a linguagem mais acessível e confiável em todos os aspectos.

7. AGRADECIMENTOS

Agradeço primeiramente ao PIBIC (Programa Institucional de Bolsas de Iniciação

Científica/CNPq/UFU), à PROPP-UFU (Pró-Reitoria de Pesquisa e Pós-Graduação da Universidade Federal de Uberlândia) pelo apoio financeiro e sobretudo à professora Sandra de Amo, orientadora deste projeto e coordenadora do Projeto CPref-SQL e Fabíola Fernandes de Souza Pereira, mestranda no Programa de Pós-graduação em Ciência da Computação da UFU que me orientou nos aspectos de implementação do projeto. Este trabalho é uma extensão da dissertação de mestrado de Fabíola [9].

8. REFERÊNCIAS BIBLIOGRÁFICAS

[1] AMO, S.; RIBEIRO, M. R. (2009). CPref-SQL: A query language supporting conditional preferences. In 24th Annual ACM Symposium on Applied Computing (ACM SAC 2009).

Disponível:

http://www.deamo.prof.ufu.br/arquivos/cprefsql-acmsac2009.pdf

[2] BÖRZSÖNYI, S.; KOSSMANN, D.; STOCKER, K. (2001). The skyline operator. In 17th International Conference on Data Engineering (ICDE), pages 421-430.

Diponível:

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.2504&rep=rep1&type=pdf

[3] BOUTILIER, C.; BRAFMAN, R. I.; DOMSHLAK, C.; HOOS, H. H.; POOLE, D. (2004). Cp-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. Journal of Artificial Intelligence Research (JAIR), 21:135-191.

Diponível:

http://www.jair.org/media/1234/live-1234-2225-jair.pdf

[4] BRAFMAN, R. I.; DOMSHLAK, C.; SHIMONY, S. E. (2006). Graphical modeling of preference and importance. On Journal of Artificial Intelligence Research (JAIR), 25:389-424.

Diponível:

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.1658&rep=rep1&type=pdf

[5] CHEN, G.; KOTZ, D. (2000). A survey of context-aware mobile computing research. Technical report tr2000-381, Dept. of Computer Science, Dartmouth College, Hanover, New Hampshire, USA.

Diponível:

http://www.cs.dartmouth.edu/reports/TR2000-381.pdf

[6] EDER, H. (2009). On Extending PostgreSQL with the Skyline Operator. PhD thesis, Vienna University of Technology.
Diponível:
http://skyline.dbai.tuwien.ac.at/thesis-heder-web-200901271.pdf

[7] KIEŸLING, W.; KÖSTLER, G. (2002). Preference sql - design, implementation, experiences. In 28th International Conference International Conference on Very Large Data Bases (VLDB), pages 990-1001.
Diponível:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.81.4360&rep=rep1&type=pdf

[8] PAPADIAS, D.; TAO, Y.; FU, G.; SEEGER, B. (2005). Progressive skyline computation in database systems. ACM Trans. Database Syst., 30(1):41-82.
Diponível:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.1745&rep=rep1&type=pdf

[9] PEREIRA, F. S. F. CPref-SQL: uma linguagem de consulta com suporte a preferências condicionais – Teoria e Implementação. Uberlândia. Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Uberlândia, 2011. 131 p.
Diponível:
http://www.lsi.ufu.br/documentos/dissertacoes_defendidas/dissertacao-Fabiola.pdf

[10] SURYANARAYANA, L.; HJELM, J.(2002). Profiles for the situated web. In Proc. 11th International World Wide Web Conference (WWW 2002), pages 200–209.
Diponível:
http://www2002.org/CDROM/refereed/214/

[11] WILSON, N. (2004). Extending cp-nets with stronger conditional preference statements. In 19th National Conference on Artificial Intelligence (AAAI), pages 735-741.
Diponível:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.58.4154&rep=rep1&type=pdf

[12] YIU, M. L.; MAMOULIS, N. (2007). Efficient processing of top-k dominating queries on multi-dimensional data. In Proceedings of the 33rd international Conference on Very large Data Bases, pages 483-494, Vienna, Austria.
Disponível:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.8020&rep=rep1&type=pdf