StreamPref: Uma Linguagem de Consulta para Dados em Fluxo com Suporte a Preferências

Marcos Roberto Ribeiro

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2018

Marcos Roberto Ribeiro

StreamPref: Uma Linguagem de Consulta para Dados em Fluxo com Suporte a Preferências

Tese de doutorado apresentada ao Programa de Pós-Graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientadora: Prof.ª Dr.ª Maria Camila Nardini Barioni

Uberlândia
2018

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da UFU, MG, Brasil.

R484s 2018

Ribeiro, Marcos Roberto, 1982-
StreamPref: uma linguagem de consulta para dados em fluxo com suporte a preferências / Marcos Roberto Ribeiro. - 2018.
275 f. : il.

Orientadora: Maria Camila Nardini Barioni.
Tese (Doutorado) - Universidade Federal de Uberlândia, Programa de Pós-Graduação em Ciência da Computação.
Disponível em: http://dx.doi.org/10.14393/ufu.te.2018.759
Inclui bibliografia.

1. Computação - Teses. 2. Simulação (Computadores) - Teses. I. Barioni, Maria Camila Nardini. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU: 681.3

Maria Salete de Freitas Pinheiro – CRB6/1262

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da Tese de doutorado intitulada **"StreamPref: Uma Linguagem de Consulta para Dados em Fluxo com Suporte a Preferências"** por **Marcos Roberto Ribeiro** como parte dos requisitos exigidos para a obtenção do título de Doutor em Ciência da Computação.

Uberlândia, 19 de março de 2018

Prof.ª Dr.ª Maria Camila Nardini Barioni
Orientadora
Universidade Federal de Uberlândia

Prof.ª Dr.ª Gina Maira Barbosa de Oliveira
Universidade Federal de Uberlândia

Prof. Dr. Renato Fileto
Universidade Federal de Santa Catarina

Prof. Dr. Bruno Augusto Nassif Travençolo
Universidade Federal de Uberlândia

Prof. Dr. Daniel dos Santos Kaster
Universidade Estadual de Londrina

Aos meus pais Gezu e Sebastiana, à minha esposa Mírian, ao meu filho Davi e à minha irmã Míriam.

Agradecimentos

A minha orientadora, Professora Doutora Maria Camila Nardini Barioni, pelos conselhos, paciência e incentivos durante a realização destes trabalho.

A minha orientadora inicial, Professora Doutora Sandra de Amo, por me inspirar e iniciar este trabalho comigo.

A professora Doutora Rita Maria da Silva Julia que inspirou soluções para o meu trabalho em sua disciplina de Inteligência Artificial.

Aos professores Doutor Cyril Labbé e Doutora Claudia Roncancio, pelos conselhos e orientação durante meu trabalho na França.

Aos servidores Alessandro e Erisvaldo que facilitaram meu trabalho na UFU.

As meus grandes amigos Crícia, Fabíola, Klérisson e Guilherme, pelo apoio em todas as minhas atividades no laboratório da UFU.

Aos meus amigos Abinoan, Ilka, Catherine, Bruno, Victor, Paola, Mário, Tien, Nicolas, Fatimeh e Amira, por tornarem minha vida mais alegre nos momentos longe de minha família.

Aos meus colegas de trabalho do IFMG Gabriel, Itagildo, Samuel, Ciniro, Daniela, Marlon, Stella e Luciana, pelo incentivo ao iniciar este longo trabalho.

Ao IFMG - Campus Bambuí pela concessão da licença capacitação e pelo apoio durante toda a realização deste trabalho.

A UFU, por me dar toda a estrutura necessária para concretizar este trabalho.

Às instituições CNPq, CAPES e FAPEMIG pelo apoio financeiro.

"If I have seen further it is by standing on the sholders of giants"
"Se eu enxerguei mais longe foi porque me apoiei nos ombros de gigantes."
(Isaac Newton)

Resumo

O trabalho descrito nesta tese está inserido no contexto do processamento de consultas sobre dados em fluxos (chamadas de consultas contínuas) com suporte a preferências. Embora existissem trabalhos nesta área, nenhuma pesquisa tinha explorado a informação temporal implícita nos dados em fluxo através de consultas contendo preferências temporais. O suporte a preferências temporais é interessante porque possibilita ao usuário expressar como os dados em um determinado instante influenciam as preferências em outro momento no tempo. O objetivo principal do trabalho descrito nesta tese foi a criação de um arcabouço teórico e prático que possibilite a realização de consultas contínuas contendo preferências condicionais temporais. Para atingir este objetivo, foi criada a linguagem de consulta StreamPref como uma extensão da *Continuous Query Language (CQL)*. Inicialmente foi criada uma otimização para o processamento de consultas contendo preferências condicionais em bancos de dados tradicionais. Em seguida, foi desenvolvido um novo algoritmo incremental para o processamento de consultas contínuas contendo preferências condicionais. Por fim, foi criada a linguagem StreamPref contendo novos operadores capazes de processar consultas contínuas com preferências condicionais temporais. Também foram propostos algoritmos eficientes para processar os novos operadores e um novo modelo de preferência temporal para a comparação de sequências de tuplas. Foi demonstrado que os novos operadores possuem operações equivalentes em CQL mostrando que a linguagem StreamPref não aumenta o poder de expressão da linguagem CQL. Entretanto, os novos operadores possuem algoritmos específicos que possibilitam o processamento de consultas StreamPref de maneira mais eficiente do que suas contrapartes em CQL. Através da realização de extensos experimentos, foi comprovado que os algoritmos desenvolvidos possuem desempenho superior em relação aos algoritmos do estado-da-arte. Além disto, foram realizados experimentos que demonstraram como diferentes combinações de operadores StreamPref afetam as comparações de sequências e, consequentemente, a resposta para o usuário.

Palavras-chave: Preferências temporais. Linguagens de consulta. Dados em fluxo.

Abstract

The work described herein explored the evaluation of queries over data streams (called continuous queries) with support to preferences. No related work had employed the implicit temporal information in the data streams through queries containing temporal preferences. The use of temporal preferences is interesting because it allows the user to express how the data at a given moment influences the preferences at another time moment. The main goal of the work described herein was to create a theoretical and practical framework that allows continuous queries containing temporal conditional preferences. In order to achieve this goal, we created the StreamPref query language as an extension of the Continuous Query Language (CQL). First, we explored the develomentp of a strategy to optimize the execution of queries containing conditional preferences in traditional databases. Then, we designed a new incremental algorithm that is able to evaluate continuous queries containing conditional preferences. Finally, we proposed the StreamPref language with new operators capable of processing continuous queries with temporal conditional preferences. We also developed an efficient algorithm to evaluate the new operators and a new temporal preference model for the comparison of sequence of tuples. We demonstrated that the new operators have equivalent operations in CQL showing that the StreamPref language does not increase the expression power of the CQL language. However, the new operators have specific algorithms that enable a faster processing of StreamPref queries than their CQL counterparts. Through extensive experiments, it has been proven that our algorithms outperforms the state-of-the-art algorithms. Moreover, we also executed experiments to demonstrate how different combinations of StreamPref operators affect the comparisons of sequences and, by consequence, the response to the user.

Keywords: Temporal preferences. Query languages. Data streams.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

BNL – *block nested loops*
BTG – *better-than graph*
CQL – *Continuous Query Language*
LTP – Lógica Temporal Proposicional
SAP — *Self-Adaptive Partition*
SGBD – Sistemas de Gerenciamento de Bancos de Dados
SGFD – Sistema de Gerenciamento de Fluxos de Dados
SMA – *Skyband Monitoring Algorithm*
SQL – *Structured Query Language*
TMA – *Top-k Monitoring Algorithm*

Lista de Algoritmos

Lista de Códigos

Lista de Notações e Símbolos

A_φ – Atributo de preferência da regra φ
$\mathbf{Att}(S)$ – Conjunto de atributos presentes em S
$\boxplus$ – Operador de janela deslizante
C_φ – Condição da regra φ
$\mathbf{Dom}(A)$ – Domínio de A
F_b^+ – Fórmula preferida da comparação b
F_b^- – Fórmula não preferida da comparação b
F_Γ – Conjunto de fórmulas essenciais de Γ
$F^{\leftarrow}$ – Conjunção das fórmulas de passado que aparecem em F
$F^{\bullet}$ – Conjunção das fórmulas de presente que aparecem em F
$G(\Gamma)$ – Grafo de dependência preferencial de Γ
Γ_A – Conjunto de regras de Γ cujo atributo de preferência é A
$\Gamma(\Phi)$ – Conjunto de regras-pc extraído a partir das regras-pct de Φ
K_Γ – Base de conhecimento sobre Γ
K_Γ^* – Conjunto de todas as comparações possíveis sobre Γ
Φ_φ – Conjunto de regras de Φ temporalmente compatíveis com φ
$Q_\Gamma(A)$ – Conjunto de todos os predicados sobre o atributo A presentes em Γ
Q_φ^+ – Valores preferidos para o atributo de preferência da regra φ
Q_φ^- – Valores não preferidos para o atributo de preferência da regra φ
$S_{Q(A)}$ – Conjunto de valores que satisfazem o predicado $Q(A)$
$s[i]$ – Posição i da sequência s
$|s|$ – Comprimento da sequência s
$\mathbf{Seq}(R)$ – Conjunto de todas as sequências possíveis compostas pelas tuplas sob o esquema relacional R
$\succ_\varphi$ – Ordem de preferência imposta por φ
τ_{now} – Instante atual em ambientes de dados em fluxo
$\mathcal{T}$ – Domínio de tempo ordenado e discreto
$t.A$ – Atributo A da tuplas t
$\mathtt{TS}(t)$ – Instante de t

$\mathbf{Tup}(R)$ – Conjunto de todas as tuplas possíveis sob esquema relacional R

W_b – Atributos indiferentes da comparação b

W_φ – Atributos indiferentes da regra φ

Sumário

Capítulo 1

Introdução

A partir dos anos 2000, houve um crescente interesse em pesquisas relacionadas a consultas sobre dados em fluxo, mais conhecidas como consultas contínuas. Este tipo de consulta é especialmente útil para lidar com dados que ocorrem naturalmente sob a forma de sequência de elementos, como em aplicações relacionadas a bolsa de valores, monitoramento de atletas e telecomunicações. Diversos trabalhos foram propostos para tratar dos problemas relacionados ao processamento de consultas contínuas (CHEN et al., 2000; BABU; WIDOM, 2001; PANDEY; RAMAMRITHAM; CHAKRABARTI, 2003; BABU et al., 2005; JAIN et al., 2008; PETIT; LABBÉ; RONCANCIO, 2012). Com o amadurecimento das pesquisas, começaram a surgir as primeiras linguagens de consulta para dados em fluxo (CHANDRASEKARAN et al., 2003; ABADI et al., 2003; KRÄMER, 2007; GEDIK et al., 2008). Dentre as linguagens propostas, a linguagem declarativa *Continuous Query Language (CQL)* merece destaque por apresentar uma semântica precisa e ser o resultado de diversos trabalhos de pesquisa (BABCOCK et al., 2002; ARASU et al., 2003; ARASU et al., 2004; BABU et al., 2005; ARASU; BABU; WIDOM, 2006; ARASU et al., 2016).

Com o crescente interesse em pesquisas sobre consultas contínuas, vários trabalhos começaram a tratar também da incorporação de preferências no processamento dessas consultas, de forma que os dados retornados atendam às preferências especificadas pelo usuário (SARKAS et al., 2008; AMO; BUENO, 2011; KONTAKI; PAPADOPOULOS; MANOLOPOULOS, 2012; PETIT et al., 2012; LEE; LEE; KIM, 2013). Em geral, as linguagens de consulta baseadas na *Structured Query Language (SQL)* possuem a cláusula `WHERE` que impõe *restrições duras* no processamento da consulta de forma que o resultado deve *obrigatoriamente* atender a essas restrições. Já as preferências do usuário são tratadas como *restrições leves*, isto faz com que os itens de dados sejam ordenados de acordo com uma hierarquia de preferência para que aqueles que melhor atendam aos desejos do usuário sejam retornados.

Apesar da existência de diversos trabalhos tratando do processamento de consultas

contínuas com preferências, uma interessante questão até então não tratada na literatura científica dessa área consistia em explorar a informação temporal implícita nos dados em fluxo através de consultas contendo *preferências temporais*. As preferências temporais permitem expressar como os dados em um determinado instante influenciam as preferências em outro momento no tempo.

1.1 Motivação

Desde o final da década de 1990, tecnologias de monitoramento de dados em fluxo passaram a ser usadas nas mais variadas áreas como transporte, segurança pública e esportes. No caso dos esportes, por exemplo, estas tecnologias são especialmente úteis para treinadores de modalidades que envolvem muitos jogadores, como o futebol. Estes profissionais podem usar sistemas computacionais capazes de acompanhar os movimentos dos jogadores durante uma partida. O treinador pode especificar preferências temporais em relação ao comportamento dos jogadores e os sistemas computacionais passam a monitorar os dados, em tempo real, para buscar os elementos que melhor atendam às preferências especificadas.

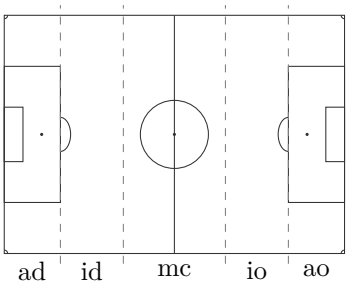
Grande parte dos trabalhos de pesquisa correlatos concentram-se no processamento de consultas *skyline* (BÖRZSÖNYI; KOSSMANN; STOCKER, 2001; LIN et al., 2005; MORSE; PATEL; GROSKY, 2007; LEE; LEE; KIM, 2013). Neste tipo de consulta, o usuário especifica preferências independentes para valores mínimos ou máximos sobre os atributos (CHOMICKI; CIACCIA; MENEGHETTI, 2013). Entretanto, em muitas aplicações o uso de *preferências condicionais* é mais adequado, permitindo ao usuário exprimir determinadas preferências quando uma certa condição for atendida (RIBEIRO, 2008; AMO; RIBEIRO, 2009; PEREIRA; AMO, 2010). Por exemplo, um treinador de futebol pode ter a seguinte preferência: "se um jogador é armador, então é melhor que ele fique no meio de campo do que na área defensiva". Dentre os trabalhos relacionados ao processamento de consultas contínuas, apenas os trabalhos de Amo e Bueno (2011) e Petit et al. (2012) abordaram a incorporação de preferências condicionais. O trabalho de Amo e Bueno (2011) propôs um algoritmo para processar consultas contínuas contendo preferências condicionais de forma incremental. Entretanto, tal algoritmo não foi implementado e não houveram experimentos para comprovar sua eficiência. Em Petit et al. (2012), foram introduzidos operadores de preferência na álgebra *Astral*, originalmente proposta em Petit, Labbé e Roncancio (2010).

Os trabalhos mencionados foram pioneiros no que diz respeito ao processamento de consultas contínuas com preferências condicionais. Contudo, na revisão bibliográfica realizada considerando o período de 2000 a 2017 nos principais veículos de publicação da área, não foram encontrados trabalhos que tratassem especificamente da incorporação de *preferências condicionais temporais* em consultas contínuas. O tema de pesquisa

que mais se aproxima do trabalho descrito nesta tese é o *processamento de consultas contínuas top-k dominantes*. Tanto no tema de pesquisa abordado por esta tese quanto nas consultas top-k dominantes, as preferências especificadas nas consultas são usadas para estabelecer uma hierarquia de preferência sobre as tuplas, para que aquelas com maior *grau de preferência* sejam retornadas. Contudo, nas *consultas top-k dominantes*, os usuário especificam preferências por valores mínimos ou máximos de atributos, não envolvendo condições ou preferências temporais. Os principais trabalhos sobre este tema são abordados no Capítulo 2.

Com a utilização de preferências condicionais temporais é possível explorar a informação temporal inerente aos fluxos de dados durante o processamento das consultas. Considere, por exemplo, um treinador de futebol que tenha acesso a um sistema de informação com dados em tempo real sobre os jogadores durante as partidas. Tal sistema possui o fluxo de dados `posicionamento (jid, local, bola)` com informações sobre o posicionamento dos jogadores. A descrição dos atributos é a seguinte:

`jid`: identificador do jogador;
`local`: região do campo de futebol de acordo com as divisões mostradas pela Figura 1(a), cujas descrições são apresentadas na Figura 1(b);
`bola`: 1 para indicar que o time possui a bola e 0 em caso negativo.



(a) Divisão em locais

área defensiva	ad
intermediária defensiva	id
meio de campo	mc
intermediária ofensiva	io
área ofensiva	ao

(b) Descrição dos locais

Figura 1 – Divisão do campo de futebol em locais

O treinador deseja que os jogadores que estavam posicionados na intermediária defensiva vão para o meio de campo se o time tem a bola. Por outro lado, se o jogador estava na intermediária ofensiva e o time perde a bola, o treinador prefere que os jogadores voltem para o meio de campo para recompor este setor. O Código 1 mostra como as preferências do treinador podem ser incorporadas em uma consulta sobre o fluxo `posicionamento` (maiores detalhes sobre a linguagem usada na consulta são apresentados no Capítulo 6). Observe que as preferências do treinador (na cláusula `ACCORDING TO PREFERENCES`) têm "condições" ligadas a "momentos anteriores", portanto são "preferências condicionais temporais". Uma das principais contribuições do trabalho descrito nesta tese é tratar do processamento de consultas contínuas contendo este tipo de preferência.

Código 1 – Consulta contendo as preferências temporais do treinador

```
SELECT SEQUENCE IDENTIFIED BY jid [RANGE 60 SECOND]
FROM pocisionamento
ACCORDING TO TEMPORAL PREFERENCES
  IF PREVIOUS (local =‘id’) AND (bola = 1)
    THEN (local =‘mc’) > (jogada =‘id’) AND
  IF PREVIOUS (local =‘io’) AND ALL PREVIOUS (bola = 1) AND (bola = 0)
    THEN (local =‘mc’) > (local =‘io’);
```

Além dos mecanismos necessários para a especificação e raciocínio com preferências, outro fator que precisa ser considerado na incorporação de preferências a linguagens de consultas é o desenvolvimento de algoritmos eficientes para o processamento das consultas. No caso das consultas contínuas com preferências este fator deve ser cuidadosamente tratado uma vez que, em cenários de dados em fluxo, podem ocorrer tráfegos intensos de informações demandando um processamento mais rápido do que em cenários de bancos de dados tradicionais.

1.2 Objetivos

O trabalho descrito nesta tese teve como objetivo principal criar um arcabouço teórico e prático que possibilite a realização de consultas contínuas contendo preferências condicionais temporais. Este arcabouço envolve a definição da linguagem de consulta StreamPref juntamente com os formalismos e algoritmos necessários para o processamento eficiente das consultas. Os objetivos específicos cumpridos para atingir o objetivo principal foram os seguintes:

1) Definição de um modelo de preferência para raciocínio com preferências condicionais temporais;
2) Especificação da linguagem de consulta StreamPref considerando os operadores necessários para processar consultas contendo preferências;
3) Desenvolvimento de algoritmos eficientes para os operadores da linguagem especificada.

1.3 Hipótese

Levando em consideração que a linguagem StreamPref tem como base a linguagem CQL, a hipótese do trabalho descrita nesta tese era a seguinte: “é possível estender a linguagem CQL com a incorporação de novos operadores que possibilitem o processamento eficiente de consultas contínuas contendo preferências condicionais temporais”. Para tanto, as seguintes questões foram abordadas:

1) Os novos operadores aumentam o poder de expressão da linguagem CQL?

2) No caso dos novos operadores que possuem operações equivalentes em CQL, as consultas construídas com estes operadores são mais eficientes do que consultas equivalentes usando apenas os operadores da linguagem CQL?

Para responder à primeira pergunta foi necessário encontrar as operações algébricas contendo apenas operadores CQL capazes de retornar o mesmo resultado obtido pelos novos operadores. Já a resposta da segunda pergunta envolveu a especificação de algoritmos para os novos operadores e execução de experimentos comparativos que comprovem o seu desempenho superior.

1.4 Abordagem do problema

Como foi explicado nas seções anteriores, o trabalho descrito nesta tese teve como objetivo desenvolver um arcabouço que permita realizar o processamento de consultas contínuas contendo preferências. O desenvolvimento de tal arcabouço envolveu a especificação da linguagem de consulta StreamPref, bem como de formalismos lógicos, novos operadores e algoritmos eficientes. A Figura 2 apresenta uma visão geral das categorias de operadores presentes na linguagem StreamPref. Os retângulos em cinza representam as novas categorias de operadores incluídas pela linguagem StreamPref. As setas tracejadas indicam tuplas enquanto as setas contínuas indicam sequências.

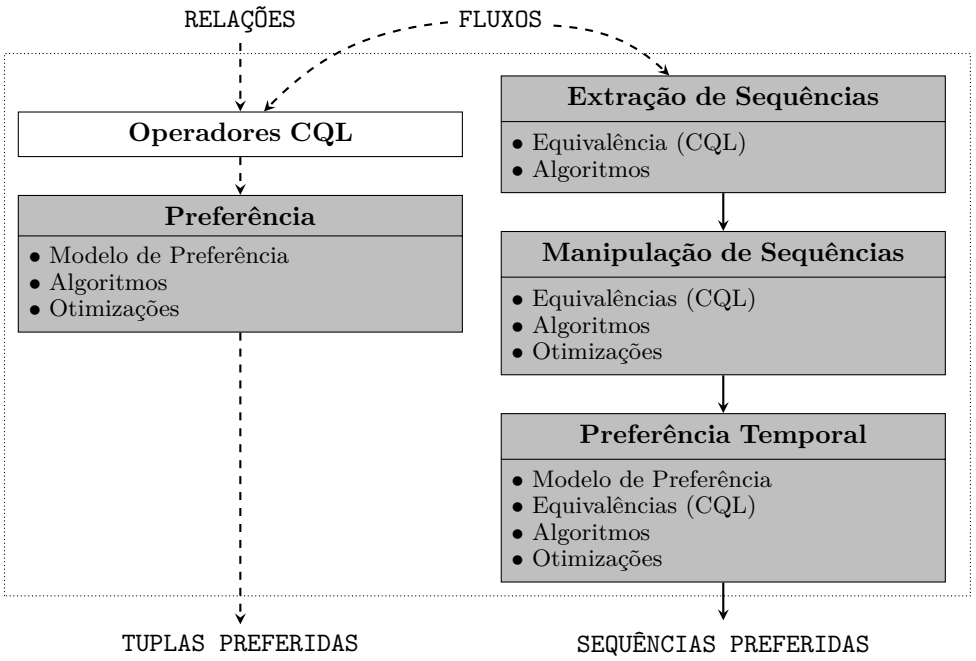


Figura 2 – Visão geral da linguagem StreamPref

É importante salientar que o desenvolvimento da linguagem StreamPref possibilita o processamento de consultas contendo tanto preferências condicionais temporais quanto preferências condicionais simples (não temporais). Isto se deve ao fato de que problemas

relacionados ao raciocínio com preferências condicionais temporais se reduzem a problemas que lidam com preferências condicionais simples. Desta maneira, parte do trabalho de pesquisa descrito nesta tese aborda a solução de problemas ligados a preferências condicionais simples uma vez que a solução de tais problemas beneficia também a solução de problemas relacionados a preferências condicionais temporais.

O desenvolvimento do trabalho descrito nesta tese envolveu três etapas. A primeira etapa descrita na Seção 1.4.1 corresponde a otimização dos algoritmos existentes para o processamento de consultas com preferências condicionais (simples) em bancos de dados tradicionais. Esta otimização é base para o desenvolvimento da segunda etapa explicada na Seção 1.4.2. A segunda etapa contempla o desenvolvimento de algoritmos otimizados para o processamento de consultas contínuas contendo preferências condicionais (simples). Por fim, a terceira etapa, detalhada na Seção 1.4.3, contempla o desenvolvimento de novos operadores e algoritmos necessários para o processamento de consultas contínuas contendo preferências condicionais temporais. Os algoritmos desenvolvidos na terceira etapa utilizam as otimizações obtidas nas etapas anteriores para atingir um melhor desempenho no processamento dos novos operadores.

1.4.1 Otimização de consultas com preferências condicionais em bancos de dados tradicionais

No trabalho de Amo e Ribeiro (2009) foi proposta a linguagem de consulta CPrefSQL como uma extensão da *Structured Query Language (SQL)* contendo um operador de preferência capaz de processar consultas com preferências condicionais. Originalmente, o operador de preferência da CPrefSQL era processado por um algoritmo baseado em grafo. O trabalho de Pereira e Amo (2010) criou um segundo operador de preferência e desenvolveu novos algoritmos para processar os operadores usando a técnica *blocked nested loops (BNL)*. Estes novos algoritmos utilizam laços aninhados para percorrer e comparar as tuplas de acordo com as preferências durante o processamento das consultas. Com estes novos algoritmos, o processamento das consultas com operadores de preferência teve um desempenho superior quando comparado ao processamento das consultas equivalentes na linguagem SQL.

Mesmo com o ganho de desempenho proporcionado pelo trabalho de Pereira e Amo (2010), ainda existiam oportunidades para a obtenção de algoritmos mais eficientes. No referido trabalho, a comparação de cada par de tuplas precisa computar o fecho transitivo imposto pelas preferências. Se o cálculo deste fecho transitivo for realizado antes de iniciar as comparações, os algoritmos podem ter um considerável ganho de desempenho.

O trabalho descrito nesta tese explorou essa questão em aberto desenvolvendo algoritmos com otimizações capazes de melhorar o desempenho dos operadores de preferência. Estas otimizações foram a base para o desenvolvimento de algoritmos mais eficientes no

processamento de consultas contínuas contendo preferências condicionais.

1.4.2 Otimização de consultas contínuas com preferências condicionais

Como foi descrito na Seção 1.1, tanto o trabalho de Amo e Bueno (2011) quanto o trabalho de Petit et al. (2012) abordaram o processamento de consultas contínuas contendo preferências condicionais. Ambos trabalhos propõem o uso de estruturas de dados para processar consultas de forma incremental, evitando assim a reanálise de todas as tuplas a cada instante.

O algoritmo proposto em Amo e Bueno (2011) não foi implementado e o algoritmo de Petit et al. (2012) não foi comparado com outras abordagens. No trabalho descrito nesta tese, as técnicas de otimização criadas para o processamento de consultas com preferências em bancos de dados tradicionais são usadas no desenvolvimento de novos algoritmos para o processamento de consultas contínuas com preferências condicionais. Em seguida, estes novos algoritmos são confrontados com os trabalhos da literatura através da análise de complexidade e de experimentos comparativos. Além disto, as otimizações adaptadas para o contexto de dados em fluxo são utilizadas na última etapa do trabalho que envolve o processamento de consultas contínuas contendo preferências condicionais temporais.

1.4.3 Processamento de consultas contínuas contendo preferências condicionais temporais

A última etapa do trabalho especifica a linguagem de consulta StreamPref como uma extensão da linguagem CQL por meio da incorporação de novos operadores. O primeiro passo desta última etapa é a definição de um modelo de preferência temporal a ser usado pelos operadores de preferência. Este modelo de preferência especifica como os itens de dados são comparados de acordo com as preferências estabelecidas pelo usuário. O modelo de preferência da linguagem StreamPref tem como base o formalismo TPref proposto no trabalho de Amo e Giacometti (2007). O formalismo TPref é uma espécie de evolução do modelo de preferências condicionais utilizado nas duas primeiras etapas. Ademais, as informações temporais implícitas dos dados em fluxo são proporcionam a aplicação conveniente de preferências temporais.

Além dos operadores de preferência temporal, a linguagem StreamPref possui operadores para extração e manipulação de sequências. Os operadores de extração de sequências são necessários para converter as tuplas dos fluxos de dados em sequências que possam ser comparadas através de preferências condicionais temporais. Os operadores de manipulação de sequências estão relacionados com operações adicionais sobre sequências permitindo a elaboração de consultas que atendam às diferentes necessidades dos usuários.

O trabalho descrito nesta tese também envolve o desenvolvimento de algoritmos para processar os novos operadores. As otimizações desenvolvidas nas duas primeiras etapas do trabalho são usadas por alguns algoritmos para proporcionar ganho de desempenho. Além disto, a terceira etapa demonstra que os novos operadores possuem operações equivalentes usando apenas os operadores CQL. Os experimentos conduzidos nesta etapa final comparam o desempenho das consultas com os novos operadores e suas contrapartes em CQL. Também são realizados experimentos para analisar como diferentes combinações de operadores afetam a resposta para o usuário. Os resultados dos experimentos comprovam a eficiência dos algoritmos desenvolvidos.

1.5 Contribuições

Vislumbra-se que as principais contribuições do trabalho descrito nesta tese foram as seguintes:

1) Algoritmos mais eficientes para o processamento de consultas contendo preferências condicionais (não temporais) em bancos de dados tradicionais;
2) Algoritmos mais eficientes para o processamento de consultas contínuas contendo preferências condicionais (não temporais);
3) Arcabouço teórico e prático para a realização de consultas contínuas contendo preferências condicionais temporais.

1.6 Organização da tese

Esta tese está organizada como se segue. No Capítulo 2 são apresentados os principais trabalhos correlatos e fundamentos teóricos. O Capítulo 3 trata das otimizações desenvolvidas para o processamento de consultas com preferências condicionais em bancos de dados tradicionais. No Capítulo 4, o processamento de consultas contínuas contendo preferências condicionais é abordado. O modelo de preferência utilizado pela linguagem StreamPref é detalhado no Capítulo 5. No Capítulo 6, os operadores da álgebra da linguagem StreamPref são explicados. O Capítulo 7 descreve os algoritmos e detalhes de implementação da linguagem StreamPref. O Capítulo 8 apresenta os experimentos realizados para validar os algoritmos desenvolvidos. E, por fim, o Capítulo 9 apresenta as considerações finais.

Capítulo 2

Fundamentação teórica

Este capítulo apresenta os fundamentos teóricos e pesquisas correlacionadas ao trabalho descrito nesta tese. A Seção 2.1 descreve os modelos de preferências condicionais mais próximos daquele utilizado pela linguagem StreamPref, bem como uma análise comparativa entre os mesmos. A Seção 2.2 descreve a linguagem CQL que serviu de base para linguagem StreamPref. Já a Seção 2.3 analisa comparativamente os trabalhos que tratam do processamento de consultas contínuas contendo preferências. Por fim, a Seção 2.4 apresenta as considerações finais sobre o capítulo.

2.1 Modelos de preferência

Esta seção trata dos formalismos lógicos sobre os quais o modelo de preferência da linguagem StreamPref foi fundamentado. A Seção 2.1.1 trata do modelo de preferência da linguagem de consulta CPrefSQL para raciocínio com preferências condicionais sobre tuplas. A Seção 2.1.2 explica o formalismo TPref usado para especificação de preferências condicionais temporais sobre sequências de tuplas.

2.1.1 Preferências sobre tuplas

Existem diversos trabalhos que tratam de modelos de preferências sobre tuplas, dentre eles estão as CP-Nets (BOUTILIER et al., 1999; BOUTILIER et al., 2004), as TCP-Nets (BRAFMAN; DOMSHLAK, 2002), a linguagem de preferências condicionais proposta por Wilson (2004) e também o modelo de preferência por traz da linguagem CPrefSQL (AMO; RIBEIRO, 2009; PEREIRA; AMO, 2010).

As CP-Nets e as TCP-Nets permitem a representação de preferências de forma gráfica e serviram de base para a linguagem de preferências condicionais de Wilson (2004). Tal linguagem utiliza regras que possibilitam especificar preferências mais genéricas do que aquelas representadas pelas CP-Nets e TCP-Nets. O modelo de preferência da linguagem

CPrefSQL foi fundamentado no trabalho de Wilson (2004) e é explicado com mais detalhes no decorrer desta seção.

Regras e teorias de preferências condicionais

A especificação de preferências na linguagem CPrefSQL é feita através de regras de preferências condicionais (Definição 1) compostas por predicados $Q(A)$ no formato $(A\theta a)$ tal que $a \in \mathbf{Dom}(A)$ e $\theta \in \{=, \neq, <, \leq, >, \geq\}$, onde $\mathbf{Dom}(A)$ denota o domínio do atributo A. A notação $S_{Q(A)}$ representa o conjunto de valores que satisfazem o predicado $Q(A)$. Estas regras são combinadas para formar uma teoria de preferências condicionais, conforme a Definição 2. O Exemplo 1 mostra como as preferências de um usuário são expressas por meio de uma teoria de preferências condicionais.

Definição 1 (Regra de preferências condicionais)**.** Dado um esquema relacional $R(A_1, ..., A_l)$. Uma regra de preferências condicionais, ou regra-pc, sobre R é uma expressão no formato $\varphi : C_\varphi \rightarrow Q^+_\varphi \succ Q^-_\varphi[W_\varphi]$, onde:

1) Os predicados Q^+_φ e Q^-_φ representam respectivamente os valores preferidos e os valores não preferidos para o atributo de preferência A_φ tais que $S_{Q^+_\varphi} \cap S_{Q^-_\varphi} = \{\}$;
2) O conjunto W_φ é o conjunto de atributos indiferentes tal que $W_\varphi \subset R$ e $A_\varphi \notin W_\varphi$;
3) A condição C_φ é uma conjunção de predicados $Q(A'_1) \wedge ... \wedge Q(A'_k)$ tal que $\{A'_1, ..., A'_k\} \in R$ e $\{A'_1, ..., A'_k\} \cap \{A_\varphi\} \cap W_\varphi = \{\}$;

Definição 2 (Teoria de preferências condicionais)**.** Seja R um esquema relacional. Uma teoria de preferências condicionais, ou teoria-pc, sobre R é um conjunto finito de regras-pc sobre R.

Exemplo 1 (Teoria-pc)**.** Considere um treinador de futebol que possui as seguintes preferências para a contratação de jogadores:

1) Quanto à função (`funcao`), prefiro armador a atacante, independente do `nome`, do número de `gols` e da `liga` onde o jogador atua;
2) Ainda sobre a função, prefiro atacante a defensor, independente do nome, do número de gols e da liga;
3) Se o jogador é armador prefiro que tenha 20 ou menos gols do que mais de 20 gols, independente do nome;
4) No caso de atacantes, jogadores com mais de 20 gols são melhores do que jogadores com 20 ou menos gols, independente do nome.

As preferências do treinador podem ser representadas pela teoria-pc $\Gamma = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4\}$ onde:

$\varphi_1 : \rightarrow (\texttt{funcao} = \textit{armador}) \succ (\texttt{funcao} = \textit{atacante})[\texttt{nome}, \texttt{gols}, \texttt{liga}]$;
$\varphi_2 : \rightarrow (\texttt{funcao} = \textit{atacante}) \succ (\texttt{funcao} = \textit{defensor})[\texttt{nome}, \texttt{gols}, \texttt{liga}]$;

φ_3 : (funcao = *armador*) $\rightarrow$ (gols ≤ 20) $\succ$ (gols > 20)[nome];
φ_4 : (funcao = *atacante*) $\rightarrow$ (gols > 20) $\succ$ (gols ≤ 20)[nome].

Ordem de Preferência

Cada regra-pc induz uma ordem, chamada de *preferência local*, sobre as tuplas de uma relação. Essa preferência local leva em consideração a semântica *ceteris paribus* ("todo o restante é igual") de forma que duas tuplas podem ser comparadas se elas têm valores coincidentes em todos os atributos, exceto no atributo de preferência e nos atributos indiferentes. Formalmente, dadas duas tuplas t e t' e uma regra-pc φ, a tupla t é preferida à tupla t' (ou t domina t') de acordo com φ, denotado por $t \succ_\varphi t'$, se:

1) As tuplas t e t' satisfazem a condição da regra φ, $t \models C_\varphi$ e $t' \models C_\varphi$;
2) As tuplas t e t' têm valores coincidentes em todos os atributos, exceto no atributo de preferência e nos atributos indiferentes, $t.A_i = t'.A_i$ para todo $A_i \notin (W_\varphi \cup \{A_\varphi\})$ (semântica *ceteris paribus*);
3) A tupla t satisfaz o predicado de valores preferidos e a tupla t' satisfaz o predicado de valores não preferidos, $t \models Q^+_\varphi$ e $t' \models Q^-_\varphi$.

A preferência local está relacionada com as comparações diretas feitas pelas regras de uma teoria-pc. No entanto, é possível também realizar comparações indiretas por transitividade utilizando mais de uma regra, conforme estabelecido pelo Teorema 1. O Exemplo 2 demonstra como as tuplas de uma relação são comparadas considerando todas as regras de uma teoria-pc.

Teorema 1. Seja Γ uma teoria-pc sobre um esquema relacional $R(A_1, ..., A_l)$. Seja $\mathbf{Tup}(R) = \mathbf{Dom}(A_1) \times ... \times \mathbf{Dom}(A_l)$ o conjunto de todas as tuplas possíveis sobre o esquema relacional R. Sejam as tuplas $t, t' \in \mathbf{Tup}(R)$. A tupla t é preferida à tupla t' de acordo com Γ, denotado por $t \succ_\Gamma t'$, se e somente se existem as tuplas $t_1, ..., t_{m+1} \in \mathbf{Tup}(R)$ e as regras-pc $\varphi_1, ..., \varphi_m \in \Gamma$ tais que $t_1 \succ_{\varphi_1} ... \succ_{\varphi_m} t_{m+1}$, $t = t_1$ e $t' = t_{m+1}$ (WILSON, 2004).

Exemplo 2 (Comparação de tuplas)**.** Considere as regras da teoria-pc Γ do Exemplo 1 e as tuplas da relação jogadores apresentada na Figura 3(a). Tomando como exemplo a regra-pc φ_3 e as tuplas t_2 e t_3, é possível constatar que $t_3 \succ_{\varphi_3} t_2$ pois:

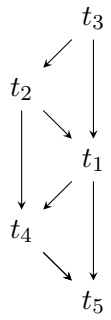
1) Ambas tuplas satisfazem a condição da regra (funcao = *armador*);
2) Ambas tuplas têm o mesmo valor para o atributo liga (atributo *ceteris paribus*);
3) Quanto ao atributo de preferência gols, a tupla t_3 tem o valor preferido (gols ≤ 20) e a tupla t_2 tem o valor não preferido (gols > 20).

O *better-than graph* (BTG) da Figura 3(b) apresenta as possíveis comparações sobre as tuplas da relação jogadores de acordo com Γ. Neste grafo, uma aresta de t para

t' indica que t é preferido a t' de acordo com Γ. As arestas obtidas pelo fecho transitivo não são representadas no grafo.

	nome	funcao	gols	liga
t_1	Messi	atacante	25	espanhola
t_2	Ribeiro	armador	21	espanhola
t_3	Oscar	armador	15	espanhola
t_4	Soares	atacante	14	espanhola
t_5	Silva	defensor	9	brasileira

(a) Relação `jogadores`



(b) Grafo BTG

Figura 3 – Relação `jogadores` e seu grafo BTG

Por meio do Teorema 1, é possível observar que uma teoria-pc Γ induz uma ordem sobre as tuplas de uma relação. Esta ordem é chamada de *ordem de preferência*. Formalmente, a ordem de preferência induzida por uma teoria-pc Γ, denotada por $\succ_\Gamma$, é o fecho transitivo de $\cup_{\varphi\in\Gamma} \succ_\varphi$. No contexto de Banco de Dados, é desejável que esta ordem seja irreflexiva para evitar situações em que uma tupla seja preferida a ela mesma. Contudo, dependendo da combinação das preferências locais a teoria-pc pode induzir uma ordem reflexiva. Quando isto acontece a teoria-pc está *inconsistente*. Como exemplo, considere novamente as tuplas da relação `jogadores` e a teoria-pc $\Gamma_2 = \{\varphi'_1, \varphi'_2, \varphi'_3\}$ onde:

$\varphi'_1 : \rightarrow (\texttt{funcao} = \textit{armador}) \succ (\texttt{funcao} = \textit{atacante})[\texttt{nome}, \texttt{gols}, \texttt{liga}]$;
$\varphi'_2 : \rightarrow (\texttt{funcao} = \textit{atacante}) \succ (\texttt{funcao} = \textit{defensor})[\texttt{nome}, \texttt{gols}, \texttt{liga}]$;
$\varphi'_3 : \rightarrow (\texttt{funcao} = \textit{defensor}) \succ (\texttt{funcao} = \textit{armador})[\texttt{nome}, \texttt{gols}, \texttt{liga}]$.

Apesar de não haver problemas em cada preferência local, a ordem de preferência $\succ_{\Gamma_2}$ obtida pelo fecho transitivo é reflexiva. Desta maneira, é possível inferir que t_1 é preferida a t_1 pelo fato de Γ_2 ser inconsistente. Para evitar situações como esta, é preciso garantir a propriedade irreflexiva da ordem de preferência por meio de um teste de consistência sobre a teoria-pc.

Teste de consistência

No trabalho de Pereira e Amo (2010), o teste de consistência originalmente proposto no trabalho de Wilson (2004) foi empregado para verificar se as preferências das consultas CPrefSQL são consistentes. No entanto, o teste de consistência original foi desenvolvido para teorias-pc contendo apenas predicados de igualdade. Para que este mesmo teste seja aplicado sobre um teoria-pc contendo predicados do tipo $(A\theta a)$, é necessário realizar um processo de reescrita das regras-pc. A ideia é converter os predicados do tipo $(A\theta a)$ em

igualdades de intervalos e eliminar as interseções entre os intervalos distintos sobre um mesmo atributo.

A reescrita das regras-pc de uma teoria-pc Γ segue os seguintes passos:

1) Se $\varphi \in \Gamma$ contém um predicado $Q(A) : (A \neq a)$, então substitua φ pelas regras φ' e φ'' trocando o predicado $Q(A)$ por $Q'(A) : (A < a)$ e $Q''(A) : (A > a)$, respectivamente;
2) Repita o passo anterior até que Γ não contenha mais predicados do tipo $(A \neq a)$;
3) Converta os predicados de todas as regras-pc para igualdades de intervalo (por exemplo, $Q(A) : (A \leq a)$ é convertido para $I(A) : [-\infty, a]$;
4) Se $\varphi \in \Gamma$ contém um predicado $I(A)$ tal que existe um predicado $I'(A)$ em Γ diferente de $I(A)$ e $I(A) \cap I'(A) \neq \{\}$, então substitua φ pelas regras φ' e φ'' trocando o predicado $I(A)$ pelos predicados $I''(A) = I(A) \cap I'(A)$ e $I'''(A) = I(A) - I'(A)$, respectivamente;
5) Repita o passo anterior até que não haja mais reescrita de regras.

O Exemplo 3 demonstra como é feito o processo de reescrita de regras sobre uma teoria-pc.

Exemplo 3 (Reescrita de regras)**.** Considere a teoria-pc $\Gamma_3 = \{\varphi_1, \varphi_2, \varphi_3\}$ onde:

$\varphi_1 : \rightarrow (\texttt{funcao} = \textit{atacante}) \succ (\texttt{funcao} \neq \textit{atacante})[\texttt{nome}, \texttt{gols}, \texttt{liga}]$;
$\varphi_2 : (\texttt{funcao} = \textit{armador}) \rightarrow (\texttt{gols} > 20) \succ (\texttt{gols} \leq 20)[\texttt{nome}]$;
$\varphi_3 : (\texttt{funcao} = \textit{atacante}) \rightarrow (\texttt{gols} > 30) \succ (\texttt{gols} \leq 30)[\texttt{nome}]$.

Após a reescrita das regras a teoria-pc Γ_3 é composta pelas seguintes regras:

$\varphi'_1 : \rightarrow \texttt{funcao} = [\textit{atacante}, \textit{atacante}] \succ \texttt{funcao} = [-\infty, \textit{atacante})[\texttt{nome}, \texttt{gols}, \texttt{liga}]$;
$\varphi''_1 : \rightarrow \texttt{funcao} = [\textit{atacante}, \textit{atacante}] \succ \texttt{funcao} = (\textit{atacante}, +\infty][\texttt{nome}, \texttt{gols}, \texttt{liga}]$;
$\varphi'_2 : \texttt{funcao} = [\textit{armador}, \textit{armador}] \rightarrow \texttt{gols} = (20, 30] \succ [-\infty, 20][\texttt{nome}]$;
$\varphi''_2 : \texttt{funcao} = [\textit{armador}, \textit{armador}] \rightarrow \texttt{gols} = (30, +\infty] \succ [-\infty, 20][\texttt{nome}]$;
$\varphi'_3 : \texttt{funcao} = [\textit{atacante}, \textit{atacante}] \rightarrow \texttt{gols} = (30, +\infty] \succ (20, 30][\texttt{nome}]$;
$\varphi''_3 : \texttt{funcao} = [\textit{atacante}, \textit{atacante}] \rightarrow \texttt{gols} = (30, +\infty] \succ [-\infty, 20][\texttt{nome}]$.

Depois da reescrita das regras, o teste de consistência pode então ser usado. A primeira etapa do teste de consistência utiliza o conceito de *grafo de dependência preferencial* dado pela Definição 3.

Definição 3 (Grafo de dependência preferencial)**.** Seja Γ uma teoria-pc sobre um esquema relacional R. O grafo de dependência preferencial de Γ, denotado por $G(\Gamma)$, é um grafo dirigido $G(\Gamma) = (V, E)$, onde $V = \mathbf{Att}(\Gamma)$ é o conjunto de atributos em Γ e $E = \cup_{\varphi \in \Gamma} E(\varphi)$, sendo que $E(\varphi) = \{(X, A_\varphi) \mid X \in \mathbf{Att}(C_\varphi)\} \cup \{(A_\varphi, Z) \mid Z \in W_\varphi\}$ (WILSON, 2004).

O grafo de dependência preferencial representa a relação de dependência entre os atributos de uma teoria-pc. Dada uma teoria-pc Γ, a primeira etapa do teste de consistência verifica se $G(\Gamma)$ é acíclico. Se $G(\Gamma)$ possui ciclos então, Γ é inconsistente. O Exemplo 4 apresenta grafos de dependência preferencial com e sem ciclos.

Exemplo 4 (Grafo de dependência preferencial)**.** Considere a teoria-pc Γ do Exemplo 1. A Figura 4(a) apresenta o grafo $G(\Gamma)$. Observe que $G(\Gamma)$ não possui ciclos, portanto Γ passa pela primeira etapa do teste de consistência. Considere agora a teoria-pc $\Gamma_4 = \{\varphi_1, \varphi_2\}$, onde:

$\varphi_1 : \rightarrow (\texttt{funcao} = \textit{armador}) \succ (\texttt{funcao} = \textit{atacante})[\texttt{nome}, \texttt{liga}]$;
$\varphi_2 : \rightarrow (\texttt{liga} = \textit{brasileira}) \succ (\texttt{liga} = \textit{espanhola})[\texttt{nome}, \texttt{funcao}]$.

Neste caso, o grafo $G(\Gamma_4)$ da Figura 4(b) não é acíclico. Portanto Γ_4 não é consistente.

(a) Grafo $G(\Gamma)$ (b) Grafo $G(\Gamma_4)$

Figura 4 – Grafos de dependência preferencial

Quando o grafo de dependência preferencial é acíclico, o teste de consistência passa para a segunda etapa que leva em consideração a *consistência local*, conforme a Definição 4. O Exemplo 5 apresenta uma teoria-pc com inconsistência local. Por fim, o Teorema 2 especifica as condições suficientes para verificar se teoria-pc é consistente.

Definição 4 (Consistência local)**.** Seja Γ uma teoria-pc sobre um esquema relacional R. Seja $A \in \mathbf{Att}(\Gamma)$ um atributo que aparece em Γ. A notação Γ_A representa o conjunto de regras-pc de Γ cujo atributo de preferência é A. A teoria-pc Γ é consistente no atributo A se a ordem de preferência $\succ_{\Gamma_A}$ é irreflexiva. Uma teoria-pc é localmente consistente se a mesma for consistente em todos os seus atributos de preferência (WILSON, 2004).

Exemplo 5 (Consistência local)**.** Considere a teoria-pc $\Gamma' = \{\varphi_1, \varphi_2, \varphi_3\}$, onde:

$\varphi_1 : \rightarrow (\texttt{funcao} = \textit{defensor}) \succ (\texttt{funcao} = \textit{armador})$;
$\varphi_2 : (\texttt{liga} = \textit{brasileira}) \rightarrow (\texttt{funcao} = \textit{armador}) \succ (\texttt{funcao} = \textit{atacante})$;
$\varphi_3 : (\texttt{gols} > 20) \rightarrow (\texttt{funcao} = \textit{atacante}) \succ (\texttt{funcao} = \textit{defensor})$.

Considere as seguintes tuplas sobre os atributos `funcao`, `gols` e `liga`:

$t_1 : (\textit{defensor}, 30, \textit{brasileira})$;
$t_2 : (\textit{armador}, 30, \textit{brasileira})$;
$t_3 : (\textit{atacante}, 30, \textit{brasileira})$.

Por meio da ordem de preferência $\succ_{\Gamma'_{\texttt{funcao}}}$ é possível inferir que $t_1 \succ_{\Gamma'_{\texttt{funcao}}} t_2$, $t_2 \succ_{\Gamma'_{\texttt{funcao}}} t_3$ e $t_3 \succ_{\Gamma'_{\texttt{funcao}}} t_1$ portanto a ordem de preferência $\succ_{\Gamma'_{\texttt{funcao}}}$ é reflexiva e Γ' não é consistente no atributo `funcao`. Logo, Γ' não é localmente consistente.

Teorema 2. Seja Γ uma teoria-pc. Se $G(\Gamma)$ é acíclico e Γ é localmente consistente, então Γ é consistente (WILSON, 2004).

2.1.2 Preferências sobre sequências de tuplas

As preferências temporais podem ser utilizadas para raciocínio sobre sequências (AMO; GIACOMETTI, 2007; AMO; GIACOMETTI, 2008) e também em tarefas de planejamento para selecionar os planos de acordo com as preferências do usuário (BIENVENU; FRITZ; MCILRAITH, 2006; BENTON; COLES; COLES, 2012). No caso da linguagem StreamPref, é utilizado um modelo de preferência temporal para realizar raciocínio com preferências sobre sequências de tuplas. O modelo de preferência utilizado tem como base o formalismo TPref proposto por Amo e Giacometti (2007) uma vez que tal modelo é mais adequado para lidar com sequências extraídas de dados em fluxo (maiores detalhes sobre o formalismo da linguagem StreamPref são apresentados no Capítulo 5). No restante desta seção todos os fundamentos teóricos apresentados são baseados no trabalho de Amo e Giacometti (2007).

Condições temporais

No formalismo TPref as preferências do usuário são representadas por meio de *regras de preferências condicionais temporais*, ou *regras-pct*. Estas regras possuem *condições temporais* que, em determinadas posições de uma sequência, tornam a regra válida ou não.

As condições temporais são expressas por meio de uma adaptação da Lógica Temporal Proposicional (LTP) introduzida em Prior (1967), visto que intuitivamente cada posição de uma sequência pode ser vista como um estado no tempo. Na LTP as fórmulas básicas são variáveis proposicionais $P_1, ..., P_n$. No formalismo lógico TPref as fórmulas básicas ou proposições têm o formato $(A = a_i)$, onde A é um atributo e $a_i \in \mathbf{Dom}(A)$. O formato das demais *fórmulas TPref* é dado pela Definição 5. As condições temporais são *satisfeitas* levando em consideração uma sequência de tuplas dada pela Definição 6.

Definição 5 (Fórmula TPref)**.** Uma *fórmula TPref* é definida como se segue:

1) **true** e **false** são fórmulas TPref;
2) Se P é uma proposição, então P é uma fórmula TPref;
3) Se P_1 e P_2 são fórmulas TPref, então $P_1 \wedge P_2$, $P_1 \vee P_2$, $\neg P_1$ e $\neg P_2$ são fórmulas TPref;
4) Se P_1 e P_2 são fórmulas TPref, então P_1 **until** P_2 e P_1 **since** P_2 são fórmulas TPref.

Definição 6 (Sequência de tuplas)**.** Uma *sequência de tuplas* é uma estrutura que consiste em um conjunto de tuplas $\{t_1, ..., t_n\}$ obedecendo a uma ordenação temporal $t_1 < ... < t_n$.

Uma sequência de tuplas s pode ser denotada simplesmente por $s = \langle t_1, ..., t_n \rangle$. A notação $|s|$ denota a dimensão temporal (comprimento) de s, ou seja, o número de tuplas

que s possui e $s[i]$ indica a i-ésima tupla de s. Seja t uma tupla. A notação $t.A_i$ indica o valor do atributo A_i na tupla t. A notação $\mathbf{Seq}(R)$ denota o conjunto de todas as possíveis sequências constituídas pelas tuplas pertencentes a $\mathbf{Tup}(R)$.

Uma fórmula TPref F é satisfeita por uma sequência de tuplas $s = \langle t_1, ..., t_n \rangle$ em uma posição $i \in \{1, ..., n\}$, denotado por $(s, i) \models F$ quando:

1) $(s, i) \models (A = a)$ se e somente se $s[i].A = a$;
2) $(s, i) \models F \wedge G$ se e somente se $(s, i) \models F$ e $(s, i) \models G$;
3) $(s, i) \models F \vee G$ se e somente se $(s, i) \models F$ ou $(s, i) \models G$;
4) $(s, i) \models \neg F$ se e somente se $(s, i) \not\models F$;
5) $(s, i) \models F$ **until** G se e somente se existe j tal que $i < j \leq |s|$ e $(s, j) \models G$ e para todo k tal que $i \leq k < j$ tem-se $(s, k) \models F$;
6) $(s, i) \models F$ **since** G se e somente se existe j tal que $1 \leq j < i$ e $(s, j) \models G$ e para todo k tal que $j < k \leq i$ tem-se $(s, k) \models F$;

A fórmula **true** sempre é satisfeita e a fórmula **false** nunca é satisfeita. A partir das fórmulas TPref definidas anteriormente, são definidas as seguintes fórmulas derivadas:

Prev F: equivale a **false since** F e referencia a posição anterior, ou seja, **Prev** F é satisfeita na posição i quando F for satisfeita na posição imediatamente anterior $(i - 1)$;

Next F: equivale a **false until** F e referencia a próxima posição, ou seja, **Next** F é satisfeita na posição i quando F for satisfeita na posição imediatamente posterior $(i + 1)$;

First: equivale a $\neg$ **Prev true** e referencia a primeira posição, ou seja, **First** é satisfeita na posição i somente quando a posição i for a primeira posição $(i = 1)$;

Last: equivale a $\neg$ **Next true** e referencia a última posição, ou seja, **Last** é satisfeita na posição i somente quando a posição i for a última posição;

$\blacklozenge F$: equivale a **true since** F e referencia alguma posição anterior, ou seja, $\blacklozenge F$ é satisfeita na posição i quando F for satisfeita em alguma posição anterior a i;

$\lozenge F$: equivale a **true until** F e referencia alguma posição posterior, ou seja, $\lozenge F$ é satisfeita na posição i quando F for satisfeita em alguma posição posterior a i;

$\blacksquare F$: equivale a $\neg\blacklozenge\neg F$ e referencia todas as posições anteriores, ou seja, $\blacksquare F$ é satisfeita na posição i quando F for satisfeita em todas as posições anteriores a i;

$\square F$: equivale a $\neg\lozenge\neg F$ e referencia todas as posições posteriores, ou seja, $\square F$ é satisfeita na posição i quando F for satisfeita em todas as posições posteriores a i.

Teoria de Preferências Condicionais Temporais

No formalismo TPref a especificação de preferências temporais é realizada por meio de *regras de preferências condicionais temporais* dadas pela Definição 7. Estas regras,

quando combinadas, compõem uma *teoria de preferências condicionais temporais* conforme a Definição 8.

Definição 7 (Regra de preferências condicionais temporais)**.** Seja $R(A_1, ..., A_l)$ um esquema relacional. Uma *regra de preferências condicionais temporais*, ou simplesmente *regra-pct*, é uma expressão na forma $\varphi : F \rightarrow (A_i = a) > (A_i = a')$, onde $(A_i \cup \mathbf{Att}(F)) \in R$, $\{a, a'\} \subseteq \mathbf{Dom}(A)$ e F é uma fórmula TPref.

Definição 8 (Teoria de preferências condicionais temporais)**.** Uma *teoria de preferências condicionais temporais*, ou simplesmente *teoria-pct*, é um conjunto finito de regras-pct simples.

Ordem Induzida

Uma teoria-pct Φ induz implicitamente uma ordem parcial estrita sobre as sequências de $\mathbf{Seq}(R)$, permitindo assim que tais sequências possam ser comparadas por meio de Φ. Cada regra-pct φ induz uma relação de ordem $\succ_\varphi$ sobre $\mathbf{Seq}(R)$. Esta relação possibilita comparar duas sequências que diferem em apenas uma posição e em apenas um atributo, conforme é mostrado na Definição 9.

Definição 9 (Comparação de sequências)**.** Dadas duas sequências $s = \langle t_1, ..., t_n \rangle$ e $s' = \langle t'_1, ..., t'_n \rangle$ sob um esquema relacional R e uma regra-pct $\varphi : F \rightarrow (A = a) > (A = a')$, $s \succ_\varphi s'$ se e somente se existe $j \in \{1, ..., n\}$ tal que:

1) $t_j \neq t'_j$ e $t_i = t'_i$ para todo $i \in \{1, ..., n\} \setminus \{j\}$;
2) $(s, j) \models F_\varphi$ e $(s', j) \models F_\varphi$;
3) $t_j[A_\varphi] = a_\varphi$ e $t'_j[A_\varphi] = a'_\varphi$;
4) Para todo $A' \in R \setminus \{A_\varphi\}$, $t_j[A'] = t'_j[A']$.

Por exemplo, considerando as sequências $s_1 = \langle (1, \textit{lateral}), (1, \textit{frente}), (0, \textit{traz}) \rangle$ e $s_2 = \langle (1, \textit{lateral}), (1, \textit{traz}), (0, \textit{traz}) \rangle$ sobre os atributos `bola` e `direcao` para os quais o atributo `bola` possui o valor 1 quando o time tem a posse de bola e 0 em caso negativo e o atributo `direcao` pode ter os valores *lateral*, *frente* ou *traz*. Considerando também a regra $\varphi : (\mathbf{Prev}(\texttt{bola} = 1)) \rightarrow (\texttt{direcao} = \textit{frente}) > (\texttt{direcao} = \textit{traz})$. A única posição diferente nas duas sequências é a segunda posição. A condição da regra-pct φ é satisfeita por ambas sequências nesta posição. O único atributo com valores diferentes na segunda posição é o atributo `direcao`. A regra-pct φ indica que o valor *frente* é preferido ao valor *traz* para o atributo `direcao`, portanto $s_1 \succ_\varphi s_2$.

A comparação de sequências que são diferentes em mais de uma posição é feita considerando o fecho transitivo da união das relações de ordem $\succ_\varphi$ para todas as regras-pct φ de Φ. Isto é, uma sequência s é preferida a uma sequência s' de acordo com Φ se $s \succ_\Phi s'$, onde $\succ_\Phi$ é o fecho transitivo de $\bigcup_{\varphi \in \Phi} \succ_\varphi$. Utilizando o conceito de *Cadeia de*

melhoramento de sequências da Definição 10, o Lema 1 apresenta a condição necessária e suficiente para que duas sequências de mesmo comprimento sejam comparadas de acordo com Φ. O Exemplo 6 demostra como comparar duas sequências usando uma cadeia de melhoramento de sequências.

Definição 10 (Cadeia de melhoramento de sequências)**.** Sejam s e s' duas sequências de comprimento n. Existe uma *cadeia de melhoramento de sequências* de s' para s de acordo com Φ se existe um conjunto de sequências $\{s_1, ..., s_{p+1}\}$ e um conjunto de regras-pct $\{\varphi_1, ..., \varphi_p\}$ em Φ tal que $s_1 = s', s_{p+1} = s$ e $s_k \succ_{\varphi_k} s_{k+1}$ para todo $k \in \{1, ..., p\}$.

Lema 1. Seja Φ uma teoria-pct. Sejam s e s' duas sequências de comprimento n. Então $s \succ_\Phi s'$ se e somente se existe uma *cadeia de melhoramento de sequências* de s' para s.

Exemplo 6 (Cadeia de melhoramento de sequências)**.** Considere as sequências $s = \langle(1, \mathit{lateral}), (0, \mathit{frente}), (0, \mathit{lateral})\rangle$, $s' = \langle(1, \mathit{lateral}), (0, \mathit{traz}), (0, \mathit{lateral})\rangle$ e a teoria-pct $\Phi = \{\varphi_1, \varphi_2, \varphi_3\}$, onde:

$\varphi_1 :$ $(\mathbf{Prev}(\texttt{bola} = 1)) \rightarrow (\texttt{direcao} = \mathit{frente}) > (\texttt{direcao} = \mathit{lateral})$;
$\varphi_2 :$ $(\mathbf{Prev}(\texttt{bola} = 1)) \rightarrow (\texttt{direcao} = \mathit{lateral}) > (\texttt{direcao} = \mathit{traz})$;
$\varphi_3 :$ $(\mathbf{Prev}(\texttt{bola} = 0)) \rightarrow (\texttt{direcao} = \mathit{traz}) > (\texttt{direcao} = \mathit{lateral})$.

É possível concluir que $s \succ_\Phi s'$ pela existência da cadeia de melhoramento de sequências $s_1 \succ_{\varphi_1} s_2 \succ_{\varphi_2} s_3 \succ_{\varphi_3} s_4$, onde:

$s_1 = s' = \langle(1, \mathit{lateral}), (0, \mathit{frente}), (0, \mathit{lateral})\rangle$;
$s_2 = \langle(1, \mathit{lateral}), (0, \mathit{lateral}), (0, \mathit{traz})\rangle$;
$s_3 = \langle(1, \mathit{lateral}), (0, \mathit{traz}), (0, \mathit{traz})\rangle$;
$s_4 = s = \langle(1, \mathit{lateral}), (0, \mathit{traz}), (0, \mathit{lateral})\rangle$.

Até agora, foram apresentados diversos conceitos e propriedades importantes principalmente quanto à representação e comparação de sequências. Contudo, em muitas situações, é preciso verificar se a ordem induzida por uma teoria-pct Φ é irreflexiva. Quando isto acontece Φ é uma teoria-pc *consistente*. O conceito de consistência é introduzido na Definição 11.

Definição 11 (Teoria-pct consistente)**.** Seja Φ uma teoria-pct, Φ é *consistente* se e somente se $\succ_\Phi$ é irreflexiva, isto é, $\succ_\Phi$ é uma ordem parcial estrita sobre $\mathbf{Seq}(R)$.

O trabalho de Amo e Giacometti (2007) propôs um teste de consistência viável apenas para um fragmento da linguagem TPref. Isto significa que não é possível testar a consistência de todas as teorias-pc expressas usando o formalismo TPref. Este é um dos pontos tratados pelo trabalho descrito nesta tese.

2.2 A linguagem de consulta *Continuous Query Language (CQL)*

No início da década de 1990 surgiram as primeiras propostas de trabalho relacionadas a consultas contínuas. Tais propostas apontavam para a inviabilidade da utilização de Sistemas de Gerenciamento de Bancos de Dados (SGBD) tradicionais no processamento de consultas contínuas, uma vez que este tipo de consulta apresenta características muito específicas. No caso das consultas contínuas, cada consulta é submetida uma única vez e, a partir de então, passa a ser processada continuamente, monitorando as mudanças nos dados e reportando o resultado ao usuário (TERRY et al., 1992).

A partir da década de 2000, com o aumento considerável da geração de informações instantâneas, surgiram propostas de aplicações para lidar especificamente com este tipo de informação. Nestas aplicações, a informação é representada por meio de sequências, potencialmente infinitas, de dados produzidas continuamente. Desde então, muitos trabalhos de pesquisa passaram a tratar do processamento de consultas contínuas (BABCOCK et al., 2002; GOLAB; ÖZSU, 2003; ARASU; BABU; WIDOM, 2006; JAIN et al., 2008; PETIT; LABBÉ; RONCANCIO, 2010; PETIT; LABBÉ; RONCANCIO, 2012; ARASU et al., 2016). Alguns destes trabalhos especificaram as primeiras linguagens de consulta para dados em fluxos (CHANDRASEKARAN et al., 2003; ABADI et al., 2003; KRÄMER, 2007; GEDIK et al., 2008).

O trabalho descrito nesta tese tem como base a linguagem de consulta *Continuous Query Language (CQL)* (ARASU; BABU; WIDOM, 2006; ARASU et al., 2016). A CQL é uma linguagem de consulta declarativa para dados em fluxo desenvolvida por pesquisadores do *Information Laboratory (InfoLab)* da Universidade de Stanford. Uma das vantagens da linguagem CQL está relacionada ao fato de ser uma linguagem de consulta de propósito geral. Os trabalhos anteriores à linguagem CQL tratavam apenas situações específicas ou não apresentavam uma semântica precisa (ARASU; WIDOM, 2004; ARASU; BABU; WIDOM, 2006; ARASU et al., 2016). Contudo, a linguagem CQL não possui operadores adequados para lidar com o processamento de consultas com preferências.

A linguagem StreamPref, introduzida pelo trabalho descrito nesta tese, estende a linguagem CQL com novos operadores. Estes operadores permitem o processamento eficiente de consultas contínuas contendo preferências. No Capítulo 6 será mostrado que os operadores da linguagem StreamPref possuem equivalências na linguagem CQL. Entretanto, será mostrado também que o desempenho destas equivalências é muito inferior em relação aos operadores StreamPref. Além do mais, a grande maioria das equivalências CQL são obtidas usando diversas operações intermediárias, isto faz com que a escrita de consultas pelo usuário se torne uma tarefa muito complexa. No restante desta seção todos os fundamentos teóricos apresentados são baseados nos trabalhos de Arasu et al. (2016), Arasu, Babu e Widom (2006).

A linguagem CQL manipula dados de relações e fluxos. As atualizações de dados ocorrem em um domínio de tempo ordenado e discreto $\mathcal{T}$. O *instante* ou *timestamp* de uma tupla é um valor em $\mathcal{T}$. As Definições 12 e 13 explicam de maneira formal o que são fluxos e relações, respectivamente. O instante de tempo não faz parte do esquema do fluxo e podem existir nenhuma ou múltiplas tuplas em um mesmo instante, desde que seja um número finito de tuplas.

Definição 12 (Fluxo)**.** Um fluxo S é um multiconjunto (possivelmente infinito) de tuplas associadas a instantes. Dada uma tupla $t \in S$, $\texttt{TS}(t) \in \mathcal{T}$ denota o instante de t.

Definição 13 (Relação)**.** Uma relação R é um mapeamento de $\mathcal{T}$ para um multiconjunto finito de tuplas pertencentes ao esquema de R.

Dados uma relação R e um instante $i \in \mathcal{T}$, $R_{\tau=i}$ é uma *relação instantânea* representando o multiconjunto de tuplas de R no instante i. Uma relação instantânea é análoga a uma relação da álgebra relacional tradicional. A representação de multiconjunto é usada para lidar com duplicatas.

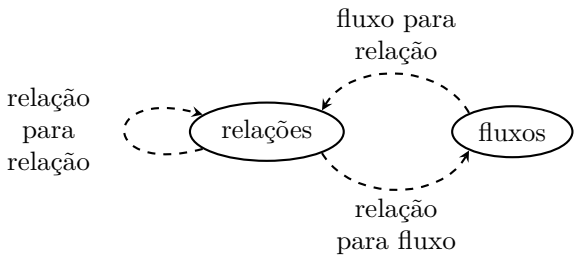


Figura 5 – Categorias de operadores da linguagem CQL

A linguagem CQL possui três tipos de operadores: *fluxo-para-relação*, *relação-para-relação* e *relação-para-fluxo*. A Figura 5 mostra as categoria de operadores da linguagem CQL e a relação entre as mesmas.

Um operador *fluxo-para-relação* toma um fluxo S como entrada e produz uma relação R como saída com o mesmo esquema de S. Em qualquer instante i, $R_{\tau=i}$ pode ser computada sobre $S_{\tau\leq i}$. Sendo que $S_{\tau\leq i}$ é o multiconjunto de tuplas de S com instante menor ou igual a i, ou seja, $\{t \in S \mid \texttt{TS}(t) \leq i\}$.

Um operador *relação-para-relação* toma uma ou mais relações $R_1, ..., R_n$ como entrada e produz uma relação R como saída. Em qualquer instante i, $R_{\tau=i}$ pode ser computada a partir das tuplas das relações $R_1, ..., R_n$ no instante i.

Um operador *relação-para-fluxo* toma uma relação R como entrada e produz um fluxo S como saída com o mesmo esquema de R. Em qualquer instante i, $S_{\tau=i}$ pode ser computado a partir de $R_{\tau\leq i}$. Sendo que $R_{\tau\leq i} = R_{\tau=0} \cup ... \cup R_{\tau=i}$ e $S_{\tau=i}$ é o multiconjunto de tuplas de S com instante igual a i.

Os operadores fluxo-para-relação têm como base o conceito de *janela deslizante* sobre um fluxo. A janela deslizante é uma porção delimitada do fluxo e os principais

tipos de janela são: baseada em tempo e baseada em contagem. As janelas baseadas em contagem sobre um fluxo S tomam como parâmetro um número inteiro positivo n e retornam uma relação R. A cada instante a relação R contém as últimas n tuplas de S. Quando o fluxo S possui várias tuplas em um mesmo instante, a janela pode ser não-determinística, pois a seleção das tuplas mais antigas com mesmo instante é feita de forma arbitrária.

As janelas baseadas em tempo operam sobre um fluxo S e retornam uma relação R considerando uma abrangência temporal n e um intervalo de deslocamento d. A abrangência temporal e o intervalo de deslocamento são usados para computar os fatores $\texttt{START}_d(t) = (\lfloor \texttt{TS}(t)/d \rfloor \times d)$ e $\texttt{END}_{n,d}(t) = (\texttt{START}_d(t) + n - 1)$. A notação $\boxplus_{n,d}(S)$ é usada para representar uma janela deslizante de abrangência temporal n e intervalo de deslocamento d sobre um fluxo S. A cada instante i, a relação R contém as tuplas de S com $\texttt{START}_d$ menor ou igual a i e $\texttt{END}_{n,d}$ maior ou igual a i, ou seja, $R_{\tau=i} = \boxplus_{n,d}(S) = \{t \in S \mid \texttt{START}_d(t) \leq i \leq \texttt{END}_{n,d}(t)\}$. A Figura 6 apresenta exemplos de janelas deslizantes sobre o fluxo `posicionamento(jid, local)`. O instante de tempo de cada tupla é representado pela coluna TS().

TS()	posicionamento	$\boxplus_{3,1}$(posicionamento)	$\boxplus_{4,2}$(posicionamento)
0	$(2, mc)$	$(2, mc)$	$(2, mc)$
1	$(5, id)$	$(2, mc)$ $(5, id)$	$(2, mc)$ $(5, id)$
2	$(10, io)$ $(6, io)$	$(2, mc)$ $(5, id)$ $(10, io)$ $(6, io)$	$(2, mc)$ $(5, id)$ $(10, io)$ $(6, io)$
3	$(8, mc)$	$(5, id)$ $(10, io)$ $(6, io)$ $(8, mc)$	$(2, mc)$ $(5, id)$ $(10, io)$ $(6, io)$ $(8, mc)$
4	$(7, ao)$ $(4, ao)$	$(10, io)$ $(6, io)$ $(8, mc)$ $(7, ao)$ $(4, ao)$	$(10, io)$ $(6, io)$ $(8, mc)$ $(7, ao)$ $(4, ao)$
5	$(11, io)$	$(8, mc)$ $(7, ao)$ $(4, ao)$ $(11, io)$	$(10, io)$ $(6, io)$ $(8, mc)$ $(7, ao)$ $(4, ao)$ $(11, io)$
⋮	⋮	⋮	⋮

Figura 6 – Exemplo de janelas deslizantes sobre um fluxo `posicionamento`

Os operadores relação-para-relação da linguagem CQL são derivações dos operadores da álgebra relacional tradicional. Seja O_r um operador da álgebra relacional tradicional sobre as relações $R_1, ..., R_n$, o operador CQL correspondente produz uma relação R no instante i considerando as tuplas das relações $R_1, ..., R_n$ no instante i.

A linguagem CQL possui três operadores relação-para-fluxo: **ISTREAM**, **DSTREAM** e **RSTREAM**. Assumindo a operação de diferença $(-)$ para multiconjuntos,

os operadores relação-para-fluxo são definidos da seguinte maneira:

- O operador **ISTREAM** (de *insert stream*) retorna as tuplas de uma relação que foram inseridas no último instante. O resultado da operação **ISTREAM**(R) no instante i é $R_{\tau=i} - R_{\tau=i-1}$;
- Analogamente, o operador **DSTREAM** (de *delete stream*) retorna as tuplas que foram removidas de uma relação no último instante. O resultado da operação **DSTREAM**(R) no instante i é $R_{\tau=i-1} - R_{\tau=i}$;
- Já o operador **RSTREAM** (de *relation stream*) retorna as tuplas presentes em uma relação no instante atual. O resultado da operação **RSTREAM**(R) no instante i é $R_{\tau=i}$;

A Figura 7 mostra a aplicação dos operadores relação-para-fluxo sobre a relação `jogador(jid, local)`.

TS()	jogador	**ISTREAM**(jogador)	**DSTREAM**(jogador)	**RSTREAM**(jogador)
0	$(2, mc)$	$(2, mc)$		$(2, mc)$
1	$(2, mc)$ $(5, id)$	$(5, id)$		$(2, mc)$ $(5, id)$
2	$(10, io)$ $(8, mc)$	$(10, io)$ $(8, mc)$	$(2, mc)$ $(5, id)$	$(10, io)$ $(8, mc)$
3	$(7, ao)$	$(7, ao)$	$(10, io)$ $(8, mc)$	$(7, ao)$
⋮	⋮	⋮	⋮	⋮

Figura 7 – Exemplo de operadores relação-para-fluxo sobre a relação `jogador`

Além da especificação da própria linguagem CQL, diversos outros trabalhos foram feitos pelo mesmo grupo de pesquisa no que diz respeito a Sistemas de Gerenciamento de Dados em Fluxo (SGDF). Tais trabalhos tratam de temas como sincronismo de *timestamp* (SRIVASTAVA; WIDOM, 2004), e gerenciamento de memória (MOTWANI; THOMAS, 2004; BABU et al., 2005).

2.3 Consultas contínuas com suporte a preferências

As consultas com suporte a preferência podem ser agrupadas em três tipos principais: consultas *skyline*, consultas *top-k* e consultas *top-k* dominantes (YIU; MAMOULIS, 2007). Uma consulta *top-k* considera uma função *score* f cujo resultado é calculado sobre os valores dos atributos das tuplas. O resultado de uma consulta *top-k* são as k tuplas com os menores (ou maiores) valores para f (HRISTIDIS; KOUDAS; PAPAKONSTANTINOU, 2001). As consultas *top-k* apresentam como vantagem a possibilidade de limitar o número de tuplas retornadas pelo parâmetro k, por outro lado a definição de uma função *score* adequada não é uma tarefa trivial.

As consultas *skyline* retornam as tuplas que não são dominadas por nenhuma outra, também chamadas de tuplas dominantes (BÖRZSÖNYI; KOSSMANN; STOCKER, 2001).

As consultas *skyline* não necessitam da definição de uma função *score*, porém o usuário não tem como limitar o número de tuplas a ser retornado.

Nas consultas *top-k* dominantes as tuplas são retornadas com base na relação de preferência existente entre elas (CHAN et al., 2006). Normalmente, neste tipo de consulta, são retornadas as k tuplas com o maior grau de dominância, sendo que o grau de dominância de uma tupla t é o número de tuplas dominadas por t (YIU; MAMOULIS, 2007). No caso das consultas *top-k* dominantes, o número de tuplas retornadas pode ser controlado pelo parâmetro k sem a necessidade de definir uma função *score*. As consultas da linguagem CPrefSQL podem ser consideradas consultas *top-k* dominantes, pois estas consultas atribuem um nível para as tuplas de forma análoga ao grau de dominância. Nas Seções 2.3.1 a 2.3.3 os principais trabalhos na área de consultas contínuas com preferências são organizados pelo tipo de consulta.

2.3.1 Consultas contínuas *top-k*

No trabalho de Mouratidis, Bakiras e Papadias (2006) foram propostos os algoritmos *Top-k Monitoring Algorithm (TMA)* e *Skyband Monitoring Algorithm (SMA)* para computar consultas contínuas *top-k* considerando janelas. O algoritmo TMA recalcula as *top-k* tuplas sempre que uma delas expira. Já o algoritmo SMA usa o conceito de *k-skyband* para responder às consultas. O *k-skyband* é o conjunto de tuplas dominadas por no máximo $k-1$ tuplas (PAPADIAS et al., 2005). Experimentos comprovaram que o consumo de memória gasto do SMA para manter o *k-skyband* é recompensado pelo tempo de resposta inferior ao apresentado pelo TMA.

Em Shen et al. (2012) foi apresentado um *framework* para responder a múltiplas consultas *top-k* com diferentes parâmetros para funções *score*, para tamanhos de janela e para valores para k. Este *framework* também utiliza o conceito de *k-skyband*, mas em conjunto com técnicas mais eficientes capazes de lidar com fluxos contendo tuplas fora de sincronia.

A técnica *self-adaptive partition (SAP)* foi proposta no trabalho de (ZHU et al., 2017). Esta técnica particiona o janela sobre o fluxo em sub-janelas e mantém um pequeno número de candidatos com os maiores *scores* em cada janela. A vantagem desta técnica é o auto-ajuste das partições para processar fluxos com diferentes distribuições de dados e consultas com diferentes parâmetros. Os algoritmos desenvolvidos com a técnica SAP apresentaram um melhor desempenho do que os algoritmos propostos nos trabalhos de Mouratidis, Bakiras e Papadias (2006) e Shen et al. (2012).

2.3.2 Consultas contínuas *skyline*

Os trabalhos pioneiros no processamento de consultas contínuas *skyline* foram propostos por Lin et al. (2005) e por Tao e Papadias (2006). Em Lin et al. (2005) foi

explorado o problema n-de-N consultas *skyline*, no qual pode ser feita uma consulta *skyline* considerando as n tuplas mais recentes, com n menor do que um limite N. Em Tao e Papadias (2006) foram desenvolvidos algoritmos que atualizam incrementalmente as tuplas dominantes considerando uma única janela com as tuplas mais recentes.

Em Morse, Patel e Grosky (2007), foi apresentado um algoritmo para processar consultas contínuas *skyline* usando o conceito de *tempo-intervalo*. Neste tipo de consulta cada tupla do fluxo possui, além do *timestamp*, um intervalo de validade. O modelo tempo-intervalo é mais genérico do que os modelos apresentados em Lin et al. (2005) e Tao e Papadias (2006). Além disto, o algoritmo *LookOut* proposto por Morse, Patel e Grosky (2007) se mostrou superior na maioria das situações quando comparado com os algoritmos desenvolvidos em Tao e Papadias (2006).

Um método para o processamento de consultas contínuas *skyline* sobre atributos categóricos e parcialmente ordenados foi apresentado em Sarkas et al. (2008). Tal método considera uma janela deslizante sobre os dados em fluxo e propõe uma estrutura de índice baseada em grade para computar eficientemente as tuplas dominantes.

No trabalho de Lee, Lee e Kim (2013) foi proposto um método para processar múltiplas consultas contínuas. Este método usa uma técnica de filtragem para descartar tuplas que não têm chance de aparecer como dominantes em futuras consultas.

2.3.3 Consultas contínuas *top-k* dominantes

Em Kontaki, Papadopoulos e Manolopoulos (2012) os autores desenvolveram algoritmos para processar consultas contínuas *top-k* dominantes onde cada tupla possui seu *timestamp* e intervalo de validade. Os algoritmos desenvolvidos operam sobre uma janela baseada em contagem com as tuplas mais recentes dos dados em fluxo. Alguns dos algoritmos desenvolvidos retornam respostas aproximadas considerando um limite de acurácia, porém apresentam um desempenho superior àqueles que retornam respostas exatas. O trabalho de (SANTOSO; CHIU, 2014) utiliza uma estrutura de grafo para consultas contínuas *top-k* dominantes. Esta estrutura é utilizada por um algoritmo que supera o desempenho do algoritmo proposto em Kontaki, Papadopoulos e Manolopoulos (2012).

No trabalho de Amo e Bueno (2011) foi proposto o algoritmo *UpdPref* para processar consultas contínuas na linguagem CPrefSQL de forma incremental. Para cada tupla t, o algoritmo mantém em memória o nível de t e as tuplas antecessoras de t (tuplas que dominam t). A cada instante i, o algoritmo utiliza as informações do instante $i-1$ para atualizar o nível e a lista de antecessores das tuplas não expiradas e das tuplas recém-chegadas.

Em Petit et al. (2012) os operadores **Select-Best** e **SelectK-Best** foram introduzidos na álgebra *Astral* (originalmente proposta em Petit, Labbé e Roncancio (2010)) permitindo a realização de consultas contínuas com preferências condicionais. Para realizar

o processamento destes operadores, os autores implementaram um algoritmo que trabalha de forma incremental mantendo em memória um grafo com arestas (t, t') quando t domina t'. A cada instante o algoritmo percorre o grafo e atualiza as arestas considerando as tuplas expiradas e as tuplas recém-chegadas.

2.4 Considerações finais

Este capítulo apresentou uma visão geral sobre os modelos de preferência, linguagens de consulta para dados em fluxo e processamento de consultas contínuas com suporte a preferências. O próximo capítulo trata da otimização dos algoritmos para processamento de consultas CPrefSQL em bancos de dados tradicionais. Esta otimização serviu de base para o processamento eficiente de consultas contínuas com preferências descritas nos capítulos posteriores.

O Capítulo 5 descreve o modelo de preferência usado pela linguagem StreamPref e o Capítulo 6 apresenta os operadores da linguagem StreamPref. Nestes capítulos os trabalhos correlatos são abordados novamente através de análises comparativas.

Capítulo 3

Otimização do processamento de consultas CPrefSQL

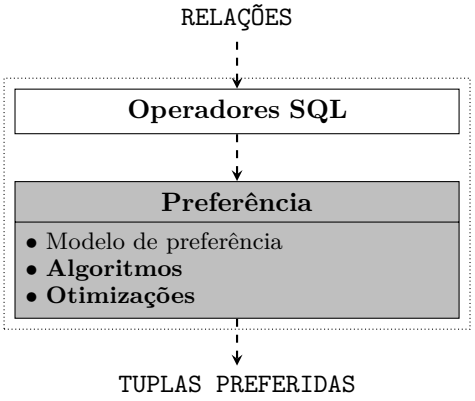


Figura 8 – Visão geral da primeira etapa do trabalho

Conforme discutido no Capítulo 1, a primeira etapa do trabalho descrito nesta tese diz respeito à otimização do processamento de consultas com preferências condicionais em bancos de dados tradicionais. Esta otimização está relacionada com os operadores de preferência da CPrefSQL destacados (em cinza) na Figura 8. A linguagem CPrefSQL é uma extensão da linguagem SQL contendo operadores de preferência e algoritmos específicos para processar tais operadores.

A proposta inicial da linguagem CPrefSQL contava apenas com o operador de preferência **BEST** (originalmente chamado de **SelectBest**) que retorna as tuplas dominantes de uma relação. Lembrando que, dada uma relação R, uma tupla $t \in R$ é dominante quando não existe outra tupla $t' \in R$ tal que t' domina t. O primeiro algoritmo desenvolvido para processar o operador **BEST** constrói um BTG com as tuplas da relação para, posteriormente, selecionar as tuplas dominantes (AMO; RIBEIRO, 2009).

O trabalho de Pereira e Amo (2010) propôs o operador de preferência **TOPK** (originalmente chamado de **SelectK-Best**) que retorna as *top-k tuplas* de uma relação

de acordo com a hierarquia de preferência estabelecida por uma teoria-pc. O conjunto das top-k tuplas de uma relação R é composto pelas k tuplas de R com o menor *nível de preferência* (Definição 14).

Definição 14 (Nível de preferência)**.** Dadas uma teoria-pc Γ e uma tupla t de uma relação R. O nível de preferência de t, denotado por $level(t)$, de acordo com Γ é definido indutivamente da seguinte maneira:

1) Se $\nexists t' \in R$ tal que $t' \succ_\Gamma t$, então $level(t) = 0$;
2) Caso contrário, $level(t) = \max\{level(t') \mid t' \succ_\Gamma t\} + 1$.

Este capítulo apresenta o teste de dominância (comparação de tuplas) baseado em busca utilizado no trabalho de Pereira e Amo (2010) e também uma otimização que utiliza o conceito de base de conhecimento (RIBEIRO; PEREIRA; DIAS, 2016). O teste de dominância baseado em busca é descrito na Seção 3.1. As demais seções explicam a otimização com base de conhecimento (RIBEIRO; PEREIRA; DIAS, 2016). A Seção 3.2 explica como construir uma base de conhecimento sobre uma teoria-pc e como usar esta base para realizar o teste de dominância. Já a Seção 3.3 apresenta os algoritmos otimizados que utilizam a base de conhecimento. A Seção 3.4 analisa a complexidade relacionada dos testes de dominância e dos novos algoritmos desenvolvidos. Por fim, A Seção 3.5 apresenta as considerações finais sobre o capítulo.

3.1 Teste de dominância baseado em busca

No trabalho de Pereira e Amo (2010) foram propostos os algoritmos *BNL*** e *R-BNL*** para processar os operadores **BEST** e **TOPK**, respectivamente. Estes algoritmos foram projetados utilizando a técnica *block nested loops (BNL)* que emprega laços aninhados para comparar cada par de tuplas da relação de entrada. A comparação das tuplas é feita através de um programa Datalog implementado por meio de uma estratégia de busca em profundidade considerando as regras de preferências informadas na consulta conforme mostrado no algoritmo *SearchDom* (Algoritmo 1).

Algoritmo 1 – $SearchDom(\Gamma, t, t')$

Entrada: Teoria-pc Γ, tuplas t e t'
Saída: **true**, se t domina t', caso contrário, **false**

```
1: if IsGoal(t, t') then                          //Verifica se a meta t' foi atingida
2:     return true                                //Retorna true
3: else
4:     for all φ ∈ Γ do                           //Para cada regra φ ∈ Γ
5:         if t ⊨ C_φ and t ⊨ Q_φ^+ then          //Verifica se t satisfaz C_φ e Q_φ^+
6:             t'' ← Change(φ, t)                 //Modifica t usando φ
7:             return SearchDom(Γ − {φ}, t'', t') //Chamada recursiva
8: return false                                   //Retorna false
```

Inicialmente, o algoritmo usa a função *IsGoal* para verificar se a tupla t atingiu a meta de busca (t'). Em caso afirmativo, o algoritmo retorna **true**. Caso contrário, são geradas novas tuplas usando as regras da teoria-pc Γ. Para cada regra-pc $\varphi \in \Gamma$, o algoritmo testa se a tupla t satisfaz a condição da regra (C_φ) e o predicado de valores preferidos (Q_φ^+). Quando a tupla atende a estes requisitos, a função *Change* (Algoritmo 2) é chamada para criar uma nova tupla t'' a partir da tupla t usando a regra φ. Em seguida, o algoritmo faz uma chamada recursiva, usando a nova tupla t'' e a teoria-pc Γ sem a regra φ, uma vez que a mesma já foi usada.

Algoritmo 2 – *Change*(φ, t)

Entrada: Regra-pc φ, tupla t
Saída: Tupla t modificada de acordo com φ

1: $t' \leftarrow t$ //Cria cópia de t
2: $t'.A_\varphi \leftarrow Interval(Q_\varphi^-)$ //Atribui o intervalo equivalente aos valores não preferidos
3: **for all** $A_i \in W_\varphi$ **do** //Para cada atributo indiferente A_i de φ
4: $\quad t'.A_i \leftarrow [-\infty, +\infty]$ //Atribui intervalo coringa ao atributo A_i
5: **return** t' //Retorna a tupla modificada

A função *Change* cria uma cópia t' da tupla t e modifica o atributo de preferência A_φ usando o intervalo correspondente ao predicado de valores não preferidos (Q_φ^-). Em seguida, todos os atributos indiferentes de t' recebem o intervalo coringa $[-\infty, +\infty]$ uma vez que estes atributos podem ter qualquer valor durante a comparação. O Exemplo 7 apresenta uma possível execução para o algoritmo *SearchDom*.

Exemplo 7 (Execução do algoritmo *SearchDom*)**.** Considere a teoria-pc $\Gamma = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4\}$ onde:

$\varphi_1 : \rightarrow (\texttt{funcao} = \textit{armador}) \succ (\texttt{funcao} = \textit{atacante})[\texttt{nome}]$;
$\varphi_2 : \rightarrow (\texttt{funcao} = \textit{atacante}) \succ (\texttt{funcao} = \textit{defensor})[\texttt{nome}, \texttt{gols}]$;
$\varphi_3 : (\texttt{funcao} = \textit{defensor}) \rightarrow (\texttt{gols} \leq 20) \succ (\texttt{gols} > 20)[\texttt{nome}]$;
$\varphi_4 : (\texttt{funcao} = \textit{atacante}) \rightarrow (\texttt{gols} > 20) \succ (\texttt{gols} \leq 20)[\texttt{nome}]$.

Considere também as tuplas $t = (\textit{Ribeiro}, \textit{armador}, 21)$ e $t' = (\textit{Soares}, \textit{atacante}, 14)$ sob o esquema relacional `jogadores(nome, funcao, gols)`. A Figura 9 apresenta a árvore de busca executada pelo algoritmo *SearchDom* para realizar o teste de dominância $t \succ_\Gamma t'$. Como a meta de busca (em cinza) é atingida, o algoritmo retorna **true**. Observe que nem todos os atributos da tupla t' são exatamente iguais aos atributos da meta. No entanto, os valores destes atributos estão todos dentro dos intervalos da meta.

Com respeito à complexidade, o custo da função *Change* é $O(l)$ onde l é o número de atributos. Já o algoritmo *SearchDom* precisa considerar os caminhos partindo da raiz para todos os demais nós da árvore de busca. No pior cenário, a profundidade da árvore e o grau dos nós é $O(m)$ onde $m = |\Gamma|$ é o número de regras. Portanto, a complexidade do algoritmo *SearchDom* é $O(lm^m)$.

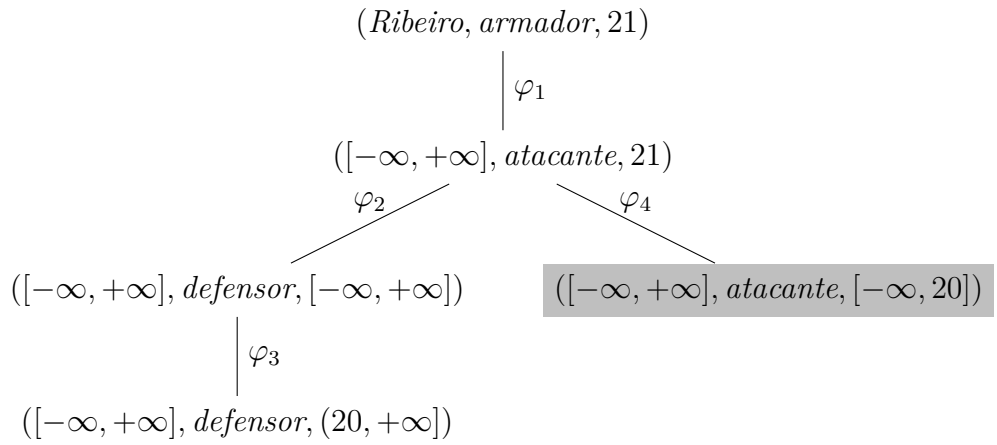


Figura 9 – Árvore de busca executada pelo algoritmo *SearchDom*

Com respeito ao algoritmo *BNL***, o teste de dominância baseado em busca é executado para cada par de tuplas. Desta maneira, a complexidade do algoritmo *BNL*** é $O(n^2 \times lm^m)$. Basicamente, o algoritmo *R-BNL*** consiste em executar o algoritmo *BNL*** um certo número de vezes. Este número de vezes é exatamente o nível de preferência máximo imposto pela teoria-pc Γ que, no pior caso, é $O(m)$. Logo, a complexidade do algoritmo *R-BNL*** é $O(m \times n^2 \times lm^m) = O(n^2 \times lm^{m+1})$.

3.2 Base de conhecimento

O teste de dominância baseado em busca precisa computar o fecho transitivo imposto pelas preferências sempre que duas tuplas são comparadas. O trabalho descrito nesta tese apresenta um novo teste de dominância que utiliza uma base de conhecimento. Esta base de conhecimento contém todas as comparações possíveis entre tuplas (incluindo aquelas feitas por transitividade). A construção da base de conhecimento também precisa calcular o fecho transitivo, porém isto é feito uma única vez e o teste de dominância precisa apenas varrer a base de conhecimento.

O primeiro passo para construção da base de conhecimento é a extração das *fórmulas essenciais* (Definição 15). O Exemplo 8 apresenta uma teoria-pc e suas respectivas fórmulas essenciais.

Definição 15 (Fórmulas essenciais)**.** Seja Γ uma teoria-pc sobre um esquema relacional $R(A_1, ..., A_l)$. Seja $Q_\Gamma(A_i)$ o conjunto de todos os predicados sobre o atributo A_i presentes em Γ. Uma fórmula essencial é definida de forma indutiva como:

1) Um predicado $Q(A_i)$ é uma fórmula essencial tal que $Q(A_i) \in Q_\Gamma(A_i)$ e $A_i \in R$;
2) Se F é uma fórmula essencial e $A_i \in R$ não aparece em F, então $F \wedge Q(A_i)$ é uma fórmula essencial tal que $Q(A_i) \in Q_\Gamma(A_i)$.

A notação F_Γ representa o conjunto de todas as fórmulas essenciais sobre Γ.

Exemplo 8 (Fórmulas essenciais)**.** Considere o esquema relacional `jogadores(nome, funcao, liga)`. Os atributos `nome`, `funcao` e `liga` indicam o nome, função e liga onde o jogador atua, respectivamente. Considere a teoria-pc Γ sobre o esquema relacional `jogadores` contendo as seguintes regras-pc:

$\varphi_1 : \rightarrow (\texttt{funcao} = \textit{armador}) \succ (\texttt{funcao} = \textit{atacante})[\texttt{nome}]$;
$\varphi_2 : \rightarrow (\texttt{funcao} = \textit{atacante}) \succ (\texttt{funcao} = \textit{defensor})[\texttt{nome}]$;
$\varphi_3 : (\texttt{funcao} = \textit{atacante}) \rightarrow (\texttt{liga} = \textit{espanhola}) \succ (\texttt{liga} = \textit{brasileira})[\texttt{nome}]$.

O conjunto de fórmulas essenciais F_Γ sobre Γ é composto pelas seguintes fórmulas:

F_1: $(\texttt{funcao} = \textit{armador})$;
F_2: $(\texttt{funcao} = \textit{atacante})$;
F_3: $(\texttt{funcao} = \textit{defensor})$;
F_4: $(\texttt{liga} = \textit{espanhola})$;
F_5: $(\texttt{liga} = \textit{brasileira})$;
F_6: $(\texttt{liga} = \textit{espanhola}) \wedge (\texttt{funcao} = \textit{armador})$;
F_7: $(\texttt{liga} = \textit{brasileira}) \wedge (\texttt{funcao} = \textit{armador})$;
F_8: $(\texttt{liga} = \textit{espanhola}) \wedge (\texttt{funcao} = \textit{atacante})$;
F_9: $(\texttt{liga} = \textit{brasileira}) \wedge (\texttt{funcao} = \textit{atacante})$;
F_{10}: $(\texttt{liga} = \textit{espanhola}) \wedge (\texttt{funcao} = \textit{defensor})$;
F_{11}: $(\texttt{liga} = \textit{brasileira}) \wedge (\texttt{funcao} = \textit{defensor})$.

As fórmulas essenciais podem ser comparadas diretamente por meio de regras-pc. Considere as fórmulas essenciais $F : G \wedge H \wedge W$ e $F' : G' \wedge H \wedge W'$ tais que G, G', H, W e W' são conjunções de predicados. Considere também a notação $\mathbf{Att}(F)$ usada para denotar o conjunto de atributos presentes na fórmula F. A fórmula F é preferida a fórmula F' de acordo com uma regra-pc φ, denotado por $F \succ_\varphi F'$, se as seguintes condições forem atendidas:

1) $G = C_\varphi \wedge Q^+_\varphi$;
2) $G' = C_\varphi \wedge Q^-_\varphi$;
3) $\mathbf{Att}(W) \subseteq W_\varphi$;
4) $\mathbf{Att}(W') \subseteq W_\varphi$.

O próximo passo para a construção da base de conhecimento é obter as possíveis *comparações* (Definição 16) entre as fórmulas essenciais. Intuitivamente, uma comparação entre duas fórmulas significa que tais fórmulas são comparáveis (de forma direta ou indireta) de acordo com uma teoria-pc.

Definição 16 (Comparação)**.** Seja Γ uma teoria-pc. Uma comparação sobre Γ é uma instrução no formato $b : (F^+_b \succ F^-_b)[W_b])$ sendo que existem as fórmulas essenciais $F_1, ..., F_{m+1} \in F_\Gamma$ e as regras-pc $\varphi_1, ..., \varphi_m \in \Gamma$ obedecendo às seguintes condições:

1) A fórmula F_b^+ é preferida a fórmula F_b^- de forma que $F_b^+ = F_1$, $F_b^- = F_{m+1}$ e $F_1 \succ_{\varphi_1} ... \succ_{\varphi_m} F_{m+1}$;
2) O conjunto $W_b = (W_{\varphi_1} \cup ... \cup W_{\varphi_n}) \cup (\{A_{\varphi_1}\} \cup ... \cup \{A_{\varphi_n}\})$ representa o conjunto de atributos indiferentes da comparação.

Assim como as regras-pc, as comparações também podem ser usadas para verificar se uma tupla é preferida a outra. Considere uma comparação b sobre uma teoria-pc Γ. Duas tuplas t e t' podem ser comparadas por b, denotado por $t \succ_b t'$, se $t \models F_b^+$, $t' \models F_b^-$ e $t.A_i = t'.A_i$ para todo $A_i \notin W_b$. O Lema 2 garante a existência de uma comparação quando duas tuplas são diretamente comparáveis por uma regra-pc.

Lema 2. Seja Γ uma teoria-pc e $\varphi \in \Gamma$ uma regra-pc. Se $t \succ_\varphi t'$ então existe uma comparação b tal que $t \succ_b t'$ (RIBEIRO; PEREIRA; DIAS, 2016).

Além das comparações diretas, a base de conhecimento precisa conter as comparações por transitividade. A notação K_Γ^* representa o conjunto de todas as possíveis comparações (diretas e transitivas) sobre Γ. Uma tupla t é preferida a uma tupla t' de acordo com K_Γ^*, denotado por $t \succ_{K_\Gamma^*} t'$, se existe uma comparação $b \in K_\Gamma^*$ tal que $t \succ_b t'$. O Teorema 3 garante que duas tuplas são comparáveis por K_Γ^* sempre que as mesmas forem comparáveis de acordo com Γ.

Teorema 3. Considere duas tuplas t e t' e uma teoria-pc Γ. A comparação $t \succ_{K_\Gamma^*} t'$ é possível se e somente se $t \succ_\Gamma t'$ (RIBEIRO; PEREIRA; DIAS, 2016).

O conjunto K_Γ^* pode conter comparações desnecessárias devido aos atributos indiferentes. Considere, por exemplo, as comparações $b : (\texttt{funcao} = \textit{armador}) \succ (\texttt{funcao} = \textit{defensor})[\texttt{funcao}, \texttt{nome}]$ e $b' : (\texttt{liga} = \textit{espanhola}) \wedge (\texttt{funcao} = \textit{armador}) > (\texttt{liga} = \textit{espanhola}) \wedge (\texttt{funcao} = \textit{defensor})[\texttt{funcao}, \texttt{nome}]$. Observe que $\succ_{b'} \subset \succ_b$, isto acontece porque b é *mais genérica* que b'. Uma comparação $b : (F^+ \succ F^-)[W_b] \in K_\Gamma^*$ é mais genérica do que uma comparação $b' : (G^+ \wedge H^+ \succ G^- \wedge H^-)[W_{b'}] \in K_\Gamma^*$ se uma das seguinte condições for verificada:

1) $F^+ = G^+$, $F^- = G^-$, $H^+ = H^-$ e $W_{b'} \subseteq W_b$;
2) $F^+ = G^+$, $F^- = G^-$, $(\mathbf{Att}(H^+) \cup W_{b'}) \subseteq W_b$ e $(\mathbf{Att}(H^-) \cup W_{b'}) \subseteq W_b$.

A base de conhecimento é então construída usando apenas as comparações *essenciais* de K_Γ^* sendo que uma comparação $b \in K_\Gamma^*$ é essencial se não existe $b' \in K_\Gamma^*$ tal que b' é mais genérica que b. A Definição 17 formaliza o conceito de base de conhecimento e o Exemplo 9 apresenta uma base de conhecimento de uma teoria-pc.

Definição 17 (Base de conhecimento)**.** Seja K_Γ^* o conjunto de comparações sobre uma teoria-pc Γ. A base de conhecimento K_Γ é o conjunto de todas as comparações essenciais de K_Γ^*.

Exemplo 9 (Base de conhecimento)**.** Considere novamente a teoria-pc Γ e suas fórmulas essenciais mostradas no Exemplo 8. A base de conhecimento K_Γ sobre esta teoria-pc é composta pelas seguintes comparações essenciais:

b_1: $(\texttt{liga} = \textit{espanhola}) \wedge (\texttt{funcao} = \textit{atacante}) \succ (\texttt{liga} = \textit{brasileira}) \wedge (\texttt{funcao} = \textit{defensor})[\texttt{funcao}, \texttt{liga}, \texttt{nome}]$

b_2: $(\texttt{liga} = \textit{espanhola}) \wedge (\texttt{funcao} = \textit{armador}) \succ (\texttt{liga} = \textit{brasileira}) \wedge (\texttt{funcao} = \textit{defensor})[\texttt{funcao}, \texttt{liga}, \texttt{nome}]$

b_3: $(\texttt{liga} = \textit{espanhola}) \wedge (\texttt{funcao} = \textit{armador}) \succ (\texttt{liga} = \textit{brasileira}) \wedge (\texttt{funcao} = \textit{atacante})[\texttt{funcao}, \texttt{liga}, \texttt{nome}]$

b_4: $(\texttt{funcao} = \textit{atacante}) \succ (\texttt{funcao} = \textit{defensor})[\texttt{funcao}, \texttt{nome}]$

b_5: $(\texttt{funcao} = \textit{armador}) \succ (\texttt{funcao} = \textit{defensor})[\texttt{funcao}, \texttt{nome}]$

b_6: $(\texttt{funcao} = \textit{armador}) \succ (\texttt{funcao} = \textit{atacante})[\texttt{funcao}, \texttt{nome}]$

b_7: $(\texttt{liga} = \textit{espanhola}) \wedge (\texttt{funcao} = \textit{atacante}) \succ (\texttt{liga} = \textit{brasileira}) \wedge (\texttt{funcao} = \textit{atacante})[\texttt{liga}, \texttt{nome}]$

O Teorema 4 garante que qualquer comparação feita usando o conjunto completo de comparações K_Γ^* também pode ser feita utilizando a base de conhecimento K_Γ contendo apenas as comparações essenciais. Portanto, uma vez construída a base de conhecimento K_Γ, o teste de dominância entre duas tuplas pode ser feito percorrendo as comparações de K_Γ.

Teorema 4. Seja Γ uma teoria-pc sobre um esquema relacional R. Se $t \succ_{K_\Gamma^*} t'$ então $t \succ_{K_\Gamma} t'$ para todo par de tupla $t, t' \in \mathbf{Tup}(R)$ (RIBEIRO; PEREIRA; DIAS, 2016).

3.3 Novos algoritmos desenvolvidos

A primeira otimização foi o desenvolvimento de novas versões dos algoritmos *BNL*** e *R-BNL***, chamadas de *BNL-KB* e *R-BNL-KB*, respectivamente. Estas otimizações realizam o teste de dominância usando a base de conhecimento, evitando assim as chamadas recursivas do teste de dominância baseado em busca.

Além da modificação nos algoritmos *BNL*** e *R-BNL***, foram desenvolvidos novos algoritmos que utilizam a base de conhecimento em conjunto com uma *técnica de particionamento*. No caso dos algoritmos baseados na técnica BNL, cada tupla da relação precisa ser comparada com todas as demais. A técnica de particionamento agrupa as tuplas em partições de acordo com os atributos *ceteris paribus* das comparações (aqueles não presentes no conjunto de atributos indiferentes). Desta maneira, os atributos *ceteris paribus* das tuplas de uma partição possuem os mesmos valores e as tuplas desta partição podem ser comparadas diretamente por meio das fórmulas da comparação.

O algoritmo *PartitionBest* (Algoritmo 3) utiliza a técnica de particionamento para processar o operador **BEST**. A função *Partition* (Algoritmo 4) é usada de forma

incremental para obter as tuplas dominantes. Para cada comparação $b \in K_\Gamma$ é feita uma chamada a esta função que elimina as tuplas dominadas de acordo com b. Ao final, o conjunto D^+ conterá apenas as tuplas dominantes de R.

Algoritmo 3 – *PartitionBest*(Γ, R)

Entrada: Relação R, teoria-pc Γ
Saída: Conjunto de tuplas dominantes em R de acordo com Γ

1: $D^+ \leftarrow R$ //Copia tuplas de R para D^+
2: **for all** $b \in K_\Gamma$ **do** //Para cada comparação b de K_Γ
3: $D^+, D^- \leftarrow Partition(D^+, b)$ //Particiona D^+ usando a comparação b
4: **return** D^+ //Retorna as tuplas dominantes

A função *Partition* recebe uma relação de tuplas T e separa estas tuplas em dominantes (D^+) e dominadas (D^-). O primeiro passo da função é construir a tabela *hash* de partições $P_\#$ por meio da função *CreatePartitions* (Algoritmo 5). Em seguida, as tuplas de cada partição $P \in P_\#$ são separadas em três grupos: aquelas que possuem os valores preferidos (P^+), aquelas que possuem os valores não preferidos (P^-) e as tuplas incomparáveis (P^*). As tuplas de P^* são dominantes em P porque elas não podem ser comparadas usando b. Como as tuplas de uma mesma partição possuem os mesmos valores para os atributos *ceteris paribus*, as tuplas de P^+ dominam as tuplas de P^-. Se P^+ for vazio, significa que todas as tuplas de P são dominantes de acordo com b. Caso contrário, as tuplas de P^+ e P^* são armazenadas no conjunto de tuplas dominantes (D^+) e as tuplas de P^- são armazenadas no conjunto de tuplas dominadas (D^-).

Algoritmo 4 – *Partition*(T, b)

Entrada: Conjunto de tuplas T, comparação b
Saída: Conjunto de tuplas dominantes (D^+) e conjunto de tuplas dominadas (D^-) em T de acordo com b

1: $P_\# \leftarrow CreatePartitions(T, b)$ //Cria as partições em $P_\#$
2: $D^+ \leftarrow \{\}$ //Conjunto de tuplas dominantes (D^+)
3: $D^- \leftarrow \{\}$ //Conjunto de tuplas dominadas (D^-)
4: **for all** $p \in P_\#$ **do** //Para cada partição p de $P_\#$
5: $P \leftarrow P_\#(p)$ //Obtém as tuplas da partição p
6: $P^+ \leftarrow \{t \in P \mid t \models F_b^+\}$ //Tuplas com valores preferidos
7: $P^- \leftarrow \{t \in P \mid t \models F_b^-\}$ //Tuplas com valores não preferidos
8: $P^* \leftarrow P - P^+ - P^-$ //Tuplas incomparáveis
9: **if** $P^+ = \{\}$ **then** //Checa se não existem tuplas em P^+
10: $D^+ \leftarrow D^+ \cup P$ //Neste caso, todas são dominantes
11: **else** //Senão, as tuplas de P^+ dominam as tuplas de P^-
12: $D^+ \leftarrow D^+ \cup P^+ \cup P^*$ //Tuplas dominantes
13: $D^- \leftarrow D^- \cup P^-$ //Tuplas dominadas
14: **return** D^+, D^-

A função *CreatePartitions* toma cada tupla $t \in T$ e a insere na partição adequada dentro da tabela *hash* $P_\#$. Primeiro a função obtém o identificador de partição p usando

os atributos de t não presentes em W_b. Em seguida, a função verifica se a partição p já existe. Em caso afirmativo, t é adicionada à partição existente e, em caso negativo, a função cria uma nova partição contendo t. O Exemplo 10 apresenta uma execução do algoritmo *PartitionBest*.

Algoritmo 5 – *CreatePartitions*(T, b)

Entrada: Conjunto de tuplas T, comparação b
Saída: Tabela *hash* com partições de T de acordo com b

```
1: P_# ← NewHashTable(T, b)                    //Cria tabela hash
2: for all t ∈ T do                            //Para cada tupla t ∈ T
3:     p ← t/W_b                               //Obtém identificador da partição p
4:     if p ∈ P_# then                         //Verifica se a partição p já existe
5:         P_#(p) ← P_#(p) ∪ {t}               //Adiciona t à partição existente
6:     else
7:         P_#(p) ← {t}                        //Cria uma nova partição contendo t
8: return P_#
```

Exemplo 10 (Execução do algoritmo *PartitionBest*)**.** Considere novamente a relação `jogadores` da Figura 10(a). Considere também a teoria-pc Γ apresentada no Exemplo 8. A Figura 10(b) mostra as partições criadas durante a execução do algoritmo *PartitionBest* tendo como parâmetros a relação `jogadores` e a teoria-pc Γ.

	nome	funcao	liga
t_1	Messi	atacante	espanhola
t_2	Ribeiro	armador	espanhola
t_3	Oscar	armador	brasileira
t_4	Willian	atacante	brasileira
t_5	Silva	defensor	espanhola

(a) Relação `jogadores`

	Partições
b_1	$P_{()} : \{t_1, t_2, t_3, t_4, t_5\}$ $\quad P^+\ P^*\ P^*\ P^*\ P^*$
b_2	$P_{()} : \{t_1, t_2, t_3, t_4, t_5\}$ $\quad P^*\ P^+\ P^*\ P^*\ P^*$
b_3	$P_{()} : \{t_1, t_2, t_3, \mathbf{t_4}, t_5\}$ $\quad P^*\ P^+\ P^*\ \mathbf{P^-}\ P^*$
b_4	$P_{(\texttt{liga}=brasileira)} : \{t_3\}$ $\quad P^*$ $P_{(\texttt{liga}=espanhola)} : \{t_1, t_2, \mathbf{t_5}\}$ $\quad P^+\ P^*\ \mathbf{P^-}$
b_5	$P_{(\texttt{liga}=brasileira)} : \{t_3\}$ $\quad P^+$ $P_{(\texttt{liga}=espanhola)} : \{t_1, t_2\}$ $\quad P^*\ P^+$
b_6	$P_{(\texttt{liga}=brasileira)} : \{t_3\}$ $\quad P^+$ $P_{(\texttt{liga}=espanhola)} : \{\mathbf{t_1}, t_2\}$ $\quad \mathbf{P^-}\ P^+$
b_7	$P_{(\texttt{funcao}=armador)} : \{t_2, t_3\}$ $\quad P^*\ P^*$

(b) Partições criadas pelo algoritmo *PartitionBest*

Figura 10 – Relação `jogadores` e partições criadas pelo algoritmo *PartitionBest*

Para cada comparação b da base de conhecimento K_Γ, é feita uma chamada à função *Partition*. No caso das três primeiras comparações, a função cria uma única partição contendo todas as tuplas. Isto ocorre porque não existem atributos *ceteris paribus* nas comparações b_1, b_2 e b_3 ($W_{b_1} = W_{b_2} = W_{b_3} = \{\texttt{nome}, \texttt{funcao}, \texttt{liga}\}$). Nos dois primeiros particionamentos, nenhuma tupla possui os valores não preferidos e todas permanecem

como dominantes. Na partição criada para a comparação b_3, a tupla t_2 possui os valores preferidos e a tupla t_4 (em negrito) possui os valores não preferidos, portando t_2 domina t_4. Como t_4 foi dominada, ela não aparece no próximo particionamento.

No caso das comparações b_4, b_5 e b_6, são criadas as partições $P_{(\texttt{liga}=brasileira)}$ e $P_{(\texttt{liga}=espanhola)}$ de acordo com os valores do atributo `liga`. Na partição $P_{(\texttt{liga}=espanhola)}$ criada para comparação b_4, a tupla t_5 (em negrito) é dominada e descartada. O mesmo acontece com a tupla t_1 na partição $P_{(\texttt{liga}=espanhola)}$ criada para a comparação b_6. O último particionamento é feito para a comparação b_7, mas nenhuma tupla é dominada. Assim, o resultado final do algoritmo contém as tuplas t_2 e t_3.

O algoritmo *PartitionTopk* (Algoritmo 6) realiza o processamento do operador **TOPK** por meio da técnica de particionamento. A lista L, usada pelo algoritmo, permite que as tuplas já processadas sejam ordenadas pelo nível de preferência. O algoritmo inicializa L como uma lista vazia e, na primeira iteração do laço mais externo, adiciona as tuplas com nível 0 à lista.

Algoritmo 6 – *PartitionTopk*(Γ, R, k)

Entrada: Relação R, teoria-pc Γ, número de top-k tuplas k
Saída: Top-k tuplas de R de acordo com Γ

1: $D^+ \leftarrow R$ //Copia relação R para D^+
2: $L \leftarrow NewList()$ //Cria lista L
3: **while** $(|L| < k)$ **and** $(D^+ \neq \{\})$ **do** //Enquanto houverem tuplas a serem processadas
4: $T \leftarrow \{\}$ //Conjunto para armazenar tuplas dominadas
5: **for all** $b \in K_\Gamma$ **do** //Para cada comparação b de K_Γ
6: $D^+, D^- \leftarrow Partition(D^+, b)$ //Separa as tuplas de D^+ em dominantes (D^+) e dominadas (D^-)
7: $T \leftarrow T \cup D^-$ //Adiciona as tuplas dominadas em T
8: $L \leftarrow L.append(D^+)$ //Acrescenta as tuplas dominantes no final da lista L
9: $D^+ \leftarrow T$ //As tuplas dominadas (T) serão separadas no próximo nível
10: **return** $L.getFirst(k)$ //Retorna as primeiras k tuplas de L

O laço mais interno do algoritmo *PartitionTopk* tem a função de percorrer todas as comparações de K_Γ e mover as tuplas dominadas de D^+ para T. A cada nova iteração do laço mais externo, as tuplas dominantes são acrescentadas ao final da lista L e as tuplas dominadas são processadas novamente para obtenção das tuplas do próximo nível. As iterações do laço mais externo cessam quando a lista L possui pelo menos k tuplas ou quando todas as tuplas já foram processadas. O Exemplo 11 demonstra a execução do algoritmo *PartitionTopk*.

Exemplo 11 (Execução do algoritmo *PartitionTopk*)**.** Considere novamente a relação `jogadores` da Figura 10(a) e a teoria-pc Γ do Exemplo 8. O processamento da operação $\mathbf{TOPK}_{\Gamma,3}(\texttt{jogadores})$ pelo algoritmo *PartitionTopk* é realizado conforme os seguintes passos:

1) A primeira iteração do laço mais externo produz as mesmas partições mostradas na Figura 10(b), neste ponto $L = [t_2, t_3]$ e $D^+ = \{t_1, t_4, t_5\}$;
2) Na segunda iteração do laço mais externo as tuplas de D^+ são processadas novamente. A Figura 11 mostra as partições criadas para D^+ durante as chamadas à função *Partition*. Neste momento $L = [t_2, t_3, t_1]$ e $D^+ = \{t_4, t_5\}$. Como $|L| = k$, o algoritmo para as iterações e as tuplas t_2, t_3 e t_1 são retornadas.

	Partições
b_1	$P_{()} : \{ \underset{P^+}{t_1}, \underset{P^*}{t_4}, \underset{P^*}{t_5} \}$
b_2	$P_{()} : \{ \underset{P^*}{t_1}, \underset{P^*}{t_4}, \underset{P^*}{t_5} \}$
b_3	$P_{()} : \{ \underset{P^*}{t_1}, \underset{P^-}{t_4}, \underset{P^*}{t_5} \}$
b_4	$P_{(\texttt{liga}=brasileira)} : \{ \underset{P^*}{t_4} \}$ $P_{(\texttt{liga}=espanhola)} : \{ \underset{P^+}{t_1}, \underset{\mathbf{P^-}}{\mathbf{t_5}} \}$
b_5	$P_{(\texttt{liga}=brasileira)} : \{ \underset{P^*}{t_4} \}$ $P_{(\texttt{liga}=espanhola)} : \{ \underset{P^*}{t_1} \}$
b_6	$P_{(\texttt{liga}=brasileira)} : \{ \underset{P^-}{t_4} \}$ $P_{(\texttt{liga}=espanhola)} : \{ \underset{P^-}{t_1} \}$
b_7	$P_{(\texttt{funcao}=atacante)} : \{ \underset{P^+}{t_1}, \underset{\mathbf{P^-}}{\mathbf{t_4}} \}$

Figura 11 – Partições criadas durante a segunda iteração do laço mais externo do algoritmo *PartitionTopk*

3.4 Análise de complexidade

A análise de complexidade descrita nesta seção leva em consideração o número de regras em Γ, o número de atributos da relação R e o número de tuplas recebidas denotados respectivamente por m, l e n. Inicialmente é preciso analisar o tamanho e o custo de construção da base de conhecimento K_Γ usada pelos algoritmos.

O primeiro passo para construir a base de conhecimento é obter as fórmulas essenciais de Γ com base no conjunto de predicados $Q_\Gamma(A_i)$ para cada atributo A_i. No pior caso, todos os atributos de R aparecem em Γ, $|\,\mathbf{Att}(\Gamma)| = l$, e todo atributo possui um predicado diferente em cada regra, $|Q_\Gamma(A_i)| = m$. Desta forma, a obtenção das fórmulas essenciais precisa combinar os m predicados dos l atributos. Portanto, no pior caso, o custo da construção das fórmulas essenciais e o número de fórmulas essenciais é $O(m^l)$.

Após a obtenção das fórmulas essenciais, a base de conhecimento é construída em duas etapas. Primeiro, é preciso formar as comparações diretas testando cada par de fórmulas essenciais. No pior cenário, todo par de fórmulas essenciais gera uma comparação, totalizando $O(m^{2l})$ comparações diretas. Em seguida, o algoritmo de Floyd-Warshall é usado para computar o fecho transitivo em tempo cúbico (HÖFNER; MÖLLER, 2012).

Portanto, no pior caso, o custo para construção da base de conhecimento é de $O(m^l + m^{2l} + (m^{2l})^3) = O(m^{6l})$. Além disto, caso todas as comparações construídas sejam essenciais, o tamanho da base de conhecimento K_Γ é $O(m^{4l})$.

Como foi explicado na Seção 3.1 a complexidade do algoritmo *BNL*** para processar o operador **BEST** é de $O(n^2 \times lm^m)$. No caso do algoritmo *BNL-KB*, o teste de dominância é feito através de uma varredura na base de conhecimento. Além disto, a base de conhecimento precisa se construída, portanto a complexidade do algoritmo *BNL-KB* é de $O(m^{6l} + n^2 \times m^{4l})$.

A complexidade do algoritmo *PartitionBest* também está relacionada com o tamanho e o custo de construção da base de conhecimento. O algoritmo faz uma chamada à função *Partition* para cada comparação de K_Γ. Esta função precisa varrer o conjunto de tuplas recebido e, para cada tupla, ler seus atributos para determinar a partição da tupla e verificar se as fórmulas da comparação são satisfeitas. Assim, o custo da função *Partition* é de $O(ln)$. Desta maneira, no pior caso, a complexidade do algoritmo *PartitionBest* é de $O(m^{6l} + ln \times m^{4l})$.

A complexidade dos algoritmos *PartitionTopk* e *R-BNL-KB* é a mesma complexidade dos algoritmos *PartitionBest* e *BNL-KB*, respectivamente, multiplicada pelo nível de preferência máximo imposto pela teoria-pc. Todos os algoritmos são afetados pelo número de tuplas recebidas (n) e pelo número de regras-pc (m). Além disto, o uso da base de conhecimento é impactado pelo número de atributos da relação (l). Na prática, o número de tuplas recebidas chega a ser milhões de vezes maior do que o número de regras ou atributos ($n \gg l$ e $n \gg m$). Isto faz com que os algoritmos baseados em particionamento tenham um desempenho superior em relação aos algoritmos que usam a técnica BNL. O Capítulo 8 apresenta os experimentos comparativos que comprovam esta afirmação.

3.5 Considerações finais

Neste capítulo foram apresentadas as otimizações para o processamento de consultas CPrefSQL sobre bancos de dados tradicionais. O capítulo apresentou um novo teste de dominância que usa uma base de conhecimento gerada a partir das regras-pc. Esta base de conhecimento permite realizar todas as comparações possíveis entre tuplas, inclusive por transitividade. O algoritmo *BNL*** original foi modificado, usando este novo teste de dominância, dando origem ao algoritmo *BNL-KB*. Além disto, foi apresentado o algoritmo *PartitionBest* que utiliza uma técnica de particionamento capaz de processar consultas CPrefSQL com menos varreduras no banco de dados.

De acordo com a análise de complexidade, o algoritmo *PartitionBest* deve apresentar um desempenho melhor do que algoritmo *BNL*** em situações práticas onde, normalmente, o número de tuplas é muito superior ao número de regras de preferência. O Capítulo 8 apresenta experimentos comparativos entre tais algoritmos que evidenciam a performance

superior do algoritmo *PartitionBest*. No próximo capítulo será abordado como a técnica de particionamento pode ser usada para o processamento de consultas CPrefSQL no contexto de dados em fluxo.

Capítulo 4

Processamento de consultas contínuas contendo preferências condicionais

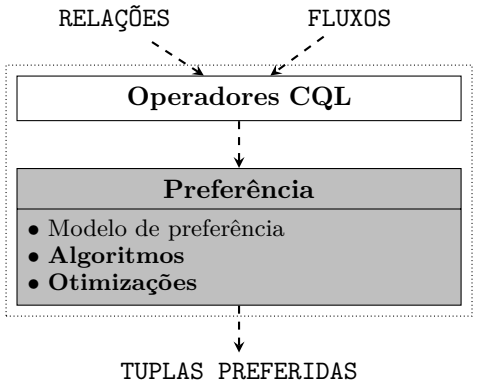


Figura 12 – Visão geral da linguagem StreamPref destacando segunda etapa do trabalho

A segunda etapa do trabalho descrito nesta tese diz respeito ao processamento de consultas contínuas contendo preferências condicionais. A Figura 12 destaca (em cinza) os operadores da linguagem StreamPref relacionados com esta etapa. A ideia principal é utilizar a técnica de particionamento descrita no Capítulo 3 para desenvolver algoritmos incrementais eficientes no contexto de dados em fluxo. A principal contribuição deste capítulo é o desenvolvimento de um algoritmo incremental que utiliza uma hierarquia de preferência baseada na técnica de particionamento apresentada no Capítulo 3. Além disto, os demais algoritmos do estado-da-arte para processamento de consultas contínuas com preferências condicionais são explicados.

O processamento de consultas contínuas contendo preferências pode ser realizado por algoritmos tradicionais ou por algoritmos incrementais, conforme mostrado na Figura 13. Os algoritmos tradicionais precisam percorrer todas as tuplas da relação a cada instante. No caso dos algoritmos incrementais, o processamento mantém em memória uma estrutura que representa a hierarquia de preferência (H) e analisa apenas as remoções (Δ^-) e inserções (Δ^+). O algoritmo *IncPref* (Algoritmo 7) mostra a ideia geral do processamento

incremental. Basicamente, os algoritmos incrementais processam as alterações (remoções e inserções) e, em seguida, retornam as tuplas preferidas do instante atual.

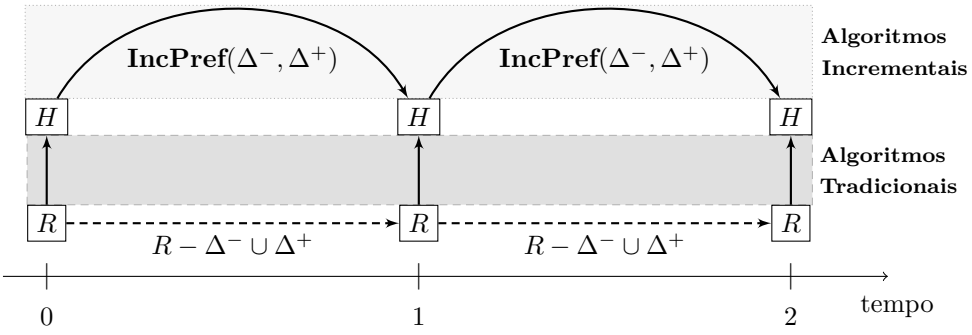


Figura 13 – Processamento de consultas contínuas contendo preferências

Algoritmo 7 – *IncPref*(Δ^-, Δ^+)

Entrada: Conjunto de tuplas removidas Δ^-, conjunto de tuplas inseridas Δ^+
Atualização: Hierarquia de preferência H
Saída: Tuplas preferidas em R no instante atual
1: Processe as remoções (Δ^-) e atualize a hierarquia de preferência H
2: Processe as inserções (Δ^+) e atualize a hierarquia de preferência H
3: Retorne as tuplas preferidas de acordo com a hierarquia de preferência H

Como foi descrito no Capítulo 1, tanto o trabalho de Amo e Bueno (2011) quanto o trabalho de Petit et al. (2012) propuseram algoritmos incrementais para o processamento de consultas contínuas contendo preferências condicionais. As principais diferenças entre estes trabalhos estão na representação da hierarquia de preferência e na semântica utilizada durante a comparação das tuplas. O algoritmo *IncAncestorsBest* desenvolvido no trabalho de Amo e Bueno (2011), originalmente chamado de *UpdPref*, utiliza listas de antecessores para armazenar a hierarquia de preferência. Já o algoritmo *IncGraphBest* proposto no trabalho de Petit et al. (2012), originalmente chamado de *KBest*, utiliza uma estrutura de grafo para esta mesma tarefa.

Quando à semântica na comparação de tuplas, o algoritmo *IncAncestorsBest* considera a ordem de preferência sobre o conjunto de todas as tuplas possíveis de um esquema relacional. O algoritmo *IncGraphBest*, por sua vez, considera uma ordem de preferência apenas sobre as tuplas existentes na relação. Por exemplo, considere uma relação R. Considere também as tuplas $t_1, t_2, t_3 \in \mathbf{Tup}(R)$ tais que $t_1, t_3 \in R$ e $t_2 \notin R$. Suponha que uma consulta contenha uma teoria-pc Γ tal que $t_1 \succ_\Gamma t_2 \succ_\Gamma t_3$. O algoritmo *IncAncestorsBest* consegue comparar as tuplas t_1 e t_3 mesmo quando $t_2 \notin R$, já no algoritmo *IncGraphBest* isto não é possível. Esta alteração foi necessária para manter a mesma semântica em todos os algoritmos. Além disto, do ponto de vista do usuário, pode ser mais interessante que as comparações entre tuplas considerem todas as tuplas possíveis de um esquema relacional.

O algoritmo *IncAncestorsBest* não passou por experimentos e o algoritmo *IncGraphBest* não foi comparado com algoritmos propostos em trabalhos correlatos. Este capítulo revisita os algoritmos *IncAncestorsBest* e *IncGraphBest* e, principalmente, apresenta o algoritmo *IncPartitionBest* desenvolvido como parte do trabalho descrito nesta tese. O algoritmo *IncPartitionBest* funciona de forma incremental e utiliza a técnica de particionamento apresentada no Capítulo 3 para processar consultas contínuas contendo preferências condicionais. Posteriormente, no Capítulo 8, são apresentados experimentos que evidenciam a performance superior do algoritmo *IncPartitionBest*.

A Seção 4.1 apresenta os algoritmos baseados em listas de antecessores. Já a Seção 4.2 trata dos algoritmos baseados em grafos. A Seção 4.3 aborda os algoritmos baseados em particionamento. Por fim, a Seção 4.4 faz as considerações finais sobre o capítulo.

4.1 Algoritmos baseados em listas de antecessores

Dada uma tupla t e uma relação R, sua lista de antecessores $Anc_{\#}(t)$ é o conjunto de todas as tuplas de R que dominam t. A partir da lista de antecessores $Anc_{\#}(t)$ é possível calcular o nível de preferência da tupla t. Se uma tupla t não possui antecessores, $|Anc_{\#}(t)| = 0$, então seu nível é zero. Por outro lado, se $|Anc_{\#}(t)| > 0$, então seu nível é $\max\{Lev_{\#}(t') \mid t' \in Anc_{\#}(t)\} + 1$.

Operação de remoção

O algoritmo *AncestorsDelete* (Algoritmo 8) realiza a operação de remoção utilizando as listas de antecessores. O conjunto *Best* tem o propósito de manter em memória as tuplas dominantes da relação. Todos os demais algoritmos incrementais utilizam este conjunto *Best* para facilitar o processamento do operador **BEST**. Para cada tupla removida $t \in \Delta^-$, o algoritmo remove t do conjunto *Best* e atualiza as listas de antecessores das demais tuplas. Esta atualização é necessária porque as listas de antecessores podem conter referências às tuplas removidas. Quando uma tupla removida t é encontrada uma lista de antecessores $Anc_{\#}(t')$, t é removida de $Anc_{\#}(t')$ e o nível de t' é reinicializado com -1. O valor -1 indica que o nível precisa ser recalculado.

Na linha 5 do algoritmo *AncestorsDelete* é utilizada uma estratégia de poda para evitar buscas desnecessárias nas listas de antecessores. Quando o nível de uma tupla t é maior ou igual ao nível de uma tupla t', significa que t não pode ser dominada por t' e, portanto, t não está na lista de antecessores $Anc_{\#}(t')$. Após a execução do algoritmo *AncestorsDelete*, a lista U contém as tuplas cujo nível de preferência precisa ser atualizado. Por questões de eficiência, a atualização dos níveis é adiada para depois da tarefa de inserção.

Algoritmo 8 – *AncestorsDelete*(Δ^-)

Entrada: Conjunto de tuplas removidas Δ^-
Atualização: Listas de antecessores $Anc_{\#}$, níveis das tuplas $Lev_{\#}$, listas de tuplas a serem atualizadas U, conjunto de tuplas dominantes *Best*

```
1: for all t ∈ Δ⁻ do                                   //Para cada tupla t removida
2:     if Lev#(t) = 0 then                              //Verifica se t é dominante
3:         Best ← Best − {t}                            //Remove t das tuplas dominantes
4:     for all t' ∈ R do                                //Para cada tupla t' remanescente
5:         if Lev#(t) < Lev#(t') then                   //Poda (somente se houver chances de t ≻ t')
6:             if t ∈ Anc#(t') then                     //Verifica se t domina t'
7:                 Anc#(t') ← Anc#(t') − {t}            //Remove t de Anc#(t')
8:                 U.append(t')                         //Acrescenta t' no final da lista de atualização U
9:                 Lev#(t') ← −1                        //Reinicializa nível de t'
```

Operação de inserção

O algoritmo *AncestorsInsert* (Algoritmo 9) efetua a operação de inserção fazendo uso das listas de antecessores. Como as novas tuplas não possuem listas de antecessores e níveis de preferência, o algoritmo precisa inicializar estas estruturas. Ademais, as tuplas de R que são afetadas pelas inserções são colocadas na lista de atualização e possuem seus níveis reinicializados.

Algoritmo 9 – *AncestorsInsert*(Γ, Δ^+)

Entrada: Teoria-pc Γ e conjunto de tuplas inseridas Δ^+
Atualização: Listas de antecessores $Anc_{\#}$, níveis das tuplas $Lev_{\#}$, listas de tuplas a serem atualizadas U, conjunto de tuplas dominantes *Best*

```
1: for all t ∈ Δ⁺ do                                   //Para cada tupla t inserida
2:     Anc#(t) ← {}                                     //Inicializa a lista de antecessores de t
3:     Lev#(t) ← −1                                     //Inicializa o nível de preferência de t
4:     U.append(t)                                      //Acrescenta t à lista de atualização
5:     for all t' ∈ R do                                //Para cada tupla t' existente
6:         if t' ≻Γ t then                              //Verifica se t' domina t
7:             Anc#(t) ← Anc#(t) ∪ {t'}                 //Acrescenta t' a lista Anc#(t)
8:         else if t ≻Γ t' then                         //Verifica se t domina t'
9:             Anc#(t') ← Anc#(t') ∪ {t}                //Acrescenta t a lista Anc#(t')
10:            U.append(t')                             //Acrescenta t à lista de atualização
11:            if Lev#(t') = 0 then                     //Verifica se t' é dominante
12:                Best ← Best − {t'}                   //Remove t' do conjunto de tuplas dominantes
13:            Lev#(t') ← −1                            //Reinicializa nível de t'
14: UpdateLevels(U)                                     //Chama função de atualização de níveis
```

O último passo do algoritmo *AncestorsInsert* é chamar a função *UpdateLevels* (Algoritmo 10). Esta função é responsável por recalcular o nível de preferências das tuplas contidas na lista de atualização U. Depois das tarefas de remoção e inserção, a lista U contém todas as tuplas inseridas e também as tuplas de R que foram afetadas por alguma remoção ou inserção.

Algoritmo 10 – *UpdateLevels*(U)

Entrada: Lista de tuplas a serem atualizadas U
Atualização: Níveis das tuplas $Lev_{\#}$ e conjunto de tuplas dominantes *Best*

```
 1: while |U| > 0 do                                  //Enquanto a lista U não for vazia
 2:     t ← U.popFirst()                              //Retira a primeira tupla t da lista
 3:     if |Anc#(t)| = 0 then                         //Verifica se t é dominante
 4:         Lev#(t) ← 0                               //Atribui zero ao nível de t
 5:         Best ← Best ∪ {t}                         //Acrescenta t ao conjunto de tuplas dominantes
 6:     else
 7:         l ← −1                                    //Supõe que o nível de t permanece inalterado
 8:         for all t' ∈ Anc#(t) do                   //Para cada antecessor t' de t
 9:             if Lev#(t') = −1 then                 //Verifica se o nível de t' não foi atualizado
10:                 l ← −1                            //O nível de t ainda não pode ser calculado
11:                 break
12:             else
13:                 l ← max{Lev#(t'), l}              //Pré-calcula nível de t
14:         if l ≠ −1 then                            //Verifica se o nível de t está pré-calculado
15:             Lev#(t) ← l + 1                       //Calcula nível correto de t
16:         else
17:             U.append(t)   //Acrescenta t ao final da lista para ser processada posteriormente
```

O algoritmo *UpdateLevels* retira sempre a primeira tupla t da lista de atualização e tenta atualizar o nível de preferência de t. Quando a lista de antecessores de t é vazia ou quando todos os antecessores de t possuem nível diferente de -1, o nível de t pode ser calculado. Caso contrário, t é inserida no final da lista U para ser reprocessada posteriormente. O algoritmo termina quando todas as tuplas da lista de atualização tiveram seus níveis de preferência calculados.

Obtenção das top-k tuplas

O processamento do operador **BEST** pode ser feito diretamente por meio do conjunto *Best* que contém as tuplas dominantes. No caso do operador **TOPK** é preciso considerar as k tuplas com os menores níveis de preferência. O algoritmo *IncAncestorsTopk* (Algoritmo 11) realiza esta tarefa. Primeiro o algoritmo adiciona as tuplas do conjunto *Best* que possuem nível zero à lista L. Depois, o algoritmo verifica se L já possui pelo menos k tuplas. Em caso negativo, o algoritmo insere todas as tuplas de R em L ordenadas pelo nível. Ao final, o algoritmo retorna as k primeiras tuplas de L. O Exemplo 12 mostra a execução do algoritmo *IncAncestorsTopk* em um cenário de dados em fluxo.

Exemplo 12 (Execução do algoritmo *IncAncestorsTopk*)**.** Considere uma aplicação financeira em um cenário de dados em fluxo contendo a relação `acoes(nome, pais, taxa, preco)`. Os atributos `nome`, `pais`, `taxa` e `preco` indicam nome da ação, país, taxa de negociação e preço, respectivamente. Suponha que um inves-

Algoritmo 11 – *IncAncestorsTopk*(R, k)

Entrada: Relação R, número de top-k tuplas desejadas k
Saída: Top-k tuplas da relação R

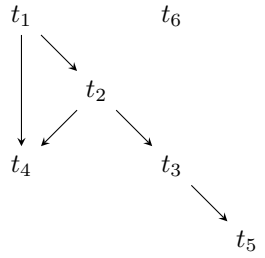
```
1: L ← NewList(Best)                    //Acrescenta as tuplas de nível 0 em L
2: l ← 1                                //Próximo nível a ser processado é 1
3: while (|L| < k) and (|L| < |R|) do   //Enquanto |L| < k e |L| < |R|
4:     for all t ∈ R do                 //Para cada tupla t ∈ R
5:         if Lev#(t) = l then          //Verifica se t possui o nível l
6:             L.append(t)              //Acrescenta t no final de L
7:     l++                              //Incrementa o nível l
8: return L.getFirst(k)                 //Retorna as k primeiras tuplas de L
```

tidor tenha preferências sobre ações representadas pela teoria-pc Γ contendo as seguintes regras-pc:

φ_1: $(\texttt{preco} = \textit{médio}) > (\texttt{preco} = \textit{alto})[\texttt{nome}, \texttt{pais}]$;
φ_2: $(\texttt{preco} = \textit{alto}) \rightarrow (\texttt{pais} = \textit{Brasil}) > (\texttt{pais} = \textit{EUA})[\texttt{nome}]$;
φ_3: $(\texttt{pais} = \textit{Brasil}) \rightarrow (\texttt{taxa} = A) > (\texttt{taxa} = B)[\texttt{nome}]$.

	nome	pais	taxa	preco
t_1	Ação 1	Brasil	A	médio
t_2	Ação 2	Brasil	A	alto
t_3	Ação 3	Brasil	B	alto
t_4	Ação 4	EUA	A	alto
t_5	Ação 5	EUA	B	alto
t_6	Ação 6	França	B	baixo

(a) Relação `acoes` no instante i



(b) BTG sobre tuplas da relação `acoes` no instante i

Figura 14 – Relação `acoes` e seu BTG no instante i

Em um instante i, a relação `acoes` possui as tuplas mostradas na Figura 14(a). O grafo BTG da Figura 14(b) mostra a hierarquia de preferência imposta por Γ sobre as tuplas da relação `acoes` no instante i. Neste momento, as listas de antecessores das tuplas são as seguintes:

- $Anc_{\#}(t_1) = Anc_{\#}(t_6) = \{\}$;
- $Anc_{\#}(t_2) = \{t_1\}$;
- $Anc_{\#}(t_3) = Anc_{\#}(t_4) = \{t_1, t_2\}$;
- $Anc_{\#}(t_5) = \{t_1, t_2, t_3\}$.

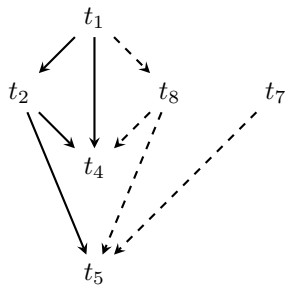
Já os níveis de preferência das tuplas são os seguintes:

- $Lev_{\#}(t_1) = Lev_{\#}(t_6) = 0$ (uma vez que $|Anc_{\#}(t_1)| = 0$ e $|Anc_{\#}(t_6)| = 0$);
- $Lev_{\#}(t_2) = \max\{Lev_{\#}(t_1)\} + 1 = 1$;
- $Lev_{\#}(t_3) = Lev_{\#}(t_4) = \max\{Lev_{\#}(t_1), Lev_{\#}(t_2)\} + 1 = 2$;
- $Lev_{\#}(t_5) = \max\{Lev_{\#}(t_1), Lev_{\#}(t_2), Lev_{\#}(t_3)\} + 1 = 3$.

Considere a operação $\mathbf{TOPK}_{\Gamma,3}(\texttt{acoes})$ no instante i, seu resultado é o conjunto de tuplas $\{t_1, t_6, t_2\}$. Suponha agora que no instante $i+1$ as tuplas t_3 e t_6 sejam removidas e que as tuplas t_7 e t_8 mostradas na Figura 15(a) sejam inseridas. A Figura 15(b) mostra o BTG da relação ações após as remoções e inserções.

	nome	pais	taxa	preco
t_7	Ação 7	EUA	B	médio
t_8	Ação 8	Brasil	A	alto

(a) Tuplas inseridas no instante $i+1$



(b) BTG da relação `acoes` no instante $i+1$

Figura 15 – Relação `acoes` e seu BTG no instante $i+1$

O processamento do operador **TOPK** no instante $i+1$ pelos algoritmos baseados em listas de antecessores inicia com o algoritmo *AncestorsDelete* que processa as tuplas removidas. Em seguida, o algoritmo *AncestorsInsert* realiza as inserções e, por fim, o algoritmo *IncAncestorsTopk* retorna as top-k tuplas no instante $i+1$.

A execução do algoritmo *AncestorsDelete* ocorre da seguinte maneira:

1) Remoção da tupla t_6:
   a) A tupla t_6 é removida do conjunto *Best*;
   b) As demais tuplas permanecem inalteradas por serem incomparáveis a t_6.
2) Remoção da tupla t_3:
   a) A tupla t_1 permanece inalterada devido à poda, $Lev_{\#}(t_1) < Lev_{\#}(t_3)$;
   b) A tupla t_2 permanece inalterada assim como t_1;
   c) A tupla t_4 permanece inalterada assim como t_1 e t_2;
   d) No caso da tupla t_5, $Anc_{\#}(t_5) \leftarrow Anc_{\#}(t_5) - \{t_3\} = \{t_1, t_2\}$ e $Lev_{\#}(t_5) \leftarrow -1$.

Já a execução do algoritmo *AncestorsInsert* acontece do seguinte modo:

1) Inserção da tupla t_7:
   a) O nível e lista de antecessores da tupla são inicializados, $Lev_{\#}(t_7) = -1$ e $Anc_{\#}(t_7) = \{\}$;
   b) Quando comparada com as demais tuplas, apenas o nível e a lista de antecessores de t_5 são alterados ($t_7 \succ t_5$, $Anc_{\#}(t_5) = \{t_1, t_2, t_7\}$, $Lev_{\#}(t_5) = -1$).
2) Inserção da tupla t_8:
   a) O nível e lista de antecessores da tupla são inicializados, $Lev_{\#}(t_8) = -1$ e $Anc_{\#}(t_8) = \{\}$;

b) A tupla t_1 domina t_8, portanto $Anc_{\#}(t_8) = \{t_1\}$;
c) As tuplas t_4 e t_5 são dominadas por t_8, portanto $Anc_{\#}(t_5) = \{t_1, t_2, t_7, t_8\}$, $Lev_{\#}(t_5) = -1$, $Anc_{\#}(t_4) = \{t_1, t_2, t_8\}$ e $Lev_{\#}(t_4) = -1$;
d) Após o processamento das inserções, a lista de atualização é a seguinte $U = [t_5, t_7, t_8, t_4]$.

3) Função *UpdateLevels*:

a) Na primeira iteração do laço externo a tupla t_5 é ignorada porque sua lista de antecessores contém tuplas com nível igual a -1;
b) Ainda na primeira iteração, as tuplas t_7 e t_8 têm seus níveis calculados, $Lev_{\#}(t_7) = 0$ e $Lev_{\#}(t_8) = Lev_{\#}(t_1) + 1 = 1$. O nível da tupla t_4 também pode ser calculado, $Lev_{\#}(t_4) = Lev_{\#}(t_2) + 1 = 2$;
c) Na segunda iteração o nível da tupla t_5 pode ser calculado, $Lev_{\#}(t_5) = Lev_{\#}(t_4)+1 = 3$.

A execução do algoritmo *IncAncestorsTopk* é realizada da seguintes maneira:

1) O conjunto $Best = \{t_1, t_7\}$ é inserido na lista L e o nível l é inicializado com 1;
2) Como a lista possui menos que três tuplas, o algoritmo executa o laço externo. O laço interno acrescenta apenas as tuplas t_2 e t_8 à lista L, apenas estas tuplas possuem nível igual a l. O nível l é incrementado para 2;
3) O laço externo pára uma vez que $|L| > k$;
4) Por fim, o algoritmo retorna as três primeiras tuplas da lista $L = [t_1, t_7, t_2, t_8]$.

Análise de complexidade

O conjunto *Best* e as listas de antecessores podem ser implementados por meio de tabelas *hash*, desta maneira, o custo amortizado das operações de busca e remoção de elementos é de $O(1)$. Sendo assim, a complexidade do algoritmo *AncestorsDelete* é de $O(n\delta^-)$, onde $|R| = n$ e $|\Delta^-| = \delta^-$.

No caso do algoritmo *AncestorsInsert* também é preciso considerar o custo da função *UpdateLevels*. No pior caso, $|U| = |R| = n$ e o nível de preferência máximo é $m = |\Gamma|$. Desta maneira, o custo da função *UpdateLevels* é $O(nm \times |Anc_{\#}(t)|)$. Na prática, os usuários não expressam preferências sobre cada par de tuplas da relação R, isto faz com que as listas de antecessores não tenham um tamanho muito grande. Considerando que o tamanho de $Anc_{\#}(t)$ é limitado por um fator constante, o custo final da função *UpdateLevels* passa a ser $O(nm)$. Logo, a complexidade final do algoritmo *AncestorsInsert* é de $O(nm + nm^m\delta^+) = O(nm^m\delta^+)$, onde $|\Delta^+| = \delta^+$ e o fator m^m é o custo do teste de dominância baseado em busca.

Como já foi explicado, o resultado do operador **BEST** pode ser obtido a qualquer instante a partir do conjunto *Best*, que é atualizado automaticamente durante as operações

de remoção e de inserção. Deste modo, o custo para processar tal operador é de $O(1)$. O operador **TOPK**, por sua vez, é processado pelo algoritmo *IncAncestorsTopk* cujo custo está relacionado com o número de iterações dos laços de repetição. O número de iterações do laço mais externo é igual ao nível de preferência máximo da teoria-pc, já o laço interno é executado uma vez para cada tupla de R. Desta maneira, a complexidade do algoritmo *IncAncestorsTopk* é $O(nm)$.

4.2 Algoritmos baseados em grafos

Os algoritmos baseados em grafos mantêm um BTG com as tuplas da relação em memória que é atualizado a cada instante. Dada uma relação R e uma teoria-pc Γ, o BTG de R é representado pela estrutura $Graph(Anc_{\#}, Succ_{\#}, Best)$. O conjunto $Anc_{\#}(t)$ contém as tuplas que dominam t, o conjunto $Succ_{\#}(t)$ representa as tuplas dominadas por t e *Best* é o conjunto das tuplas dominantes. Esta seção apresenta uma adaptação do algoritmo baseado em grafos proposto em Petit et al. (2012) para que seja considerada a ordem de preferência sobre o conjunto de todas as tuplas do esquema relacional. Esta adaptação se faz necessária para que todos os algoritmos analisados neste capítulo tenham a mesma semântica e possam ser comparados mais adequadamente.

Operação de remoção

O algoritmo *GraphDelete* (Algoritmo 12) é responsável pela tarefa de remoção. Para cada tupla removida $t \in \Delta^-$, o algoritmo verifica se t é uma tupla dominante e a remove do conjunto de tuplas dominantes *Best*. Em seguida, o algoritmo remove as arestas relacionadas à tupla t. Lembrando que as arestas são representadas pelas entradas nas estruturas $Anc_{\#}$ e $Succ_{\#}$.

Algoritmo 12 – *GraphDelete*(Δ^-)

Entrada: Conjunto de tuplas removidas Δ^-
Atualização: Arestas do grafo ($Anc_{\#}$ e $Succ_{\#}$), conjunto de tuplas dominantes *Best*

1: **for all** $t \in \Delta^-$ **do** //Para cada tupla t removida
2: **if** $t \in Best$ **then** //Checa se t é dominante
3: $Best \leftarrow Best - \{t\}$ //Remove t do conjunto *Best*
4: **for all** $t' \in Anc_{\#}(t)$ **do** //Para cada tupla t' que domina t
5: $Succ_{\#}(t') \leftarrow Succ_{\#}(t') - \{t\}$ //Remove t do conjunto de tuplas dominadas por t'
6: **for all** $t'' \in Succ_{\#}(t)$ **do** //Para cada tupla t'' dominada por t
7: $Anc_{\#}(t'') \leftarrow Anc_{\#}(t'') - \{t\}$ //Remove t do conjunto de tuplas que domina t''
8: $Anc_{\#}.del(t)$ //Remove a entrada t de $Anc_{\#}$
9: $Succ_{\#}.del(t)$ //Remove a entrada t de $Succ_{\#}$

Operação de inserção

O algoritmo *GraphInsert* (Algoritmo 13) realiza a tarefa de inserção. O algoritmo compara cada tupla t inserida com cada tupla t' já existente na relação R. Quando t domina t', o algoritmo acrescenta uma aresta de t para t' e remove t' do conjunto *Best*, caso t' seja um tupla dominante. Por outro lado, se t' domina t, o algoritmo adiciona uma aresta de t' para t e marca t como dominada. Após as iterações do laço mais interno, a tupla t é inserida no conjunto *Best*, se a mesma não foi dominada.

A adaptação em relação ao trabalho original de Petit et al. (2012) aconteceu exatamente na tarefa de inserção. No trabalho original o teste de dominância considerava apenas as tuplas existentes na relação. Nesta adaptação o teste de dominância usa a estratégia de busca que considera a ordem de preferência imposta sobre $\mathbf{Tup}(R)$, assim como acontece na abordagem em listas de antecessores.

Algoritmo 13 – *GraphInsert*(Γ, Δ^+)

Entrada: Teoria-pc Γ e conjunto de tuplas inseridas Δ^+
Atualização: Arestas do grafo ($Anc_{\#}$ e $Succ_{\#}$), conjunto de tuplas dominantes *Best*

```
1:  for all t ∈ Δ+ do                                   //Para cada tupla t inserida
2:      dominated ← false                               //Pressupõe que t não será dominada
3:      for all t' ∈ R do                               //Para cada tupla t' já existente
4:          if t ≻Γ t' then                             //Testa se t domina t'
5:              Succ#(t) ← Succ#(t) ∪ {t'}              //Insere t' entre as tuplas dominadas por t
6:              Anc#(t') ← Anc#(t') ∪ {t}               //Insere t entra as tuplas que dominam t'
7:              if t' ∈ Best then                       //Verifica se t' era dominante
8:                  Best ← Best − {t'}                  //Remove t' do conjunto de tuplas dominantes
9:          else if t' ≻Γ t then                        //Testa se t' domina t
10:             dominated ← true                        //Marca t como dominada
11:             Succ#(t') ← Succ#(t') ∪ {t}             //Insere t entre as tuplas dominadas por t'
12:             Anc#(t) ← Anc#(t) ∪ {t'}                //Insere t' entra as tuplas que dominam t
13:     if not dominated then                           //Verifica se t não foi dominada
14:         Best ← Best ∪ {t}                           //Adiciona t entra as tuplas dominantes
```

Obtenção das top-k tuplas

O algoritmo *IncGraphTopk* (Algoritmo 14) realiza o processamento do operador **TOPK** por meio de uma ordenação topológica no grafo (KAHN, 1962). O algoritmo cria cópias das estruturas *Best* e $Anc_{\#}$ para *Best*$'$ e $Anc'_{\#}$, respectivamente. Esta cópia tem o objetivo de preservar as estruturas *Best* e $Anc_{\#}$ que são alteradas durante a ordenação topológica. A lista L é utilizada para ordenar as tuplas, de forma que uma tupla t é adicionada à L somente se todas as tuplas que dominam t já estiverem em L.

Inicialmente, o conjunto *Best*$'$ contém as tuplas com nível de preferência igual a zero, ou seja, as tuplas dominantes. A cada iteração, as tuplas do conjunto *Best*$'$ com nível i são inseridas na lista L e as tuplas de N com nível $i+1$ são movidas para *Best*$'$. Depois

Algoritmo 14 – *IncGraphTopk*(R, k)

Entrada: Relação R e número de top-k tuplas desejadas k
Saída: Top-k tuplas da relação R

1: $Best' \leftarrow Best$ //Copia $Best$ para $Best'$
2: $Anc'_{\#} \leftarrow Anc_{\#}$ //Copia $Anc_{\#}$ para $Anc'_{\#}$
3: $L \leftarrow NewList()$ //Cria lista L para ordenar tuplas
4: **while** $(|L| < k)$ **and** $(|L| < |R|)$ **do** //Enquanto $|L| < k$ e $|L| < |R|$
5: $N \leftarrow \{\}$ //Tuplas a serem processadas na próxima iteração
6: **for all** $t \in Best'$ **do** //Para cada tupla t dominante (na iteração atual)
7: $L.append(t)$ //Adiciona t à lista L
8: **for all** $t' \in Succ_{\#}(t)$ **do** //Para cada tupla t' dominada por t
9: $Anc'_{\#}(t') \leftarrow Anc'_{\#}(t') - \{t\}$ //Remove a aresta de t para t'
10: **if** $|Anc_{\#}(t')| = 0$ **then** //Verifica se t' deixou de ser dominada
11: $N \leftarrow N \cup \{t'\}$ //Insere t' em N
12: $Best' \leftarrow N$ //Move tuplas de N para $Best'$
13: **return** $L.getFirst(k)$ //Retorna as k primeiras tuplas de L

de todas as iterações, a lista L possuirá as tuplas de R ordenadas pelo nível de preferência. Ao final, o algoritmo retorna as k primeiras tuplas de L. O Exemplo 13 apresenta uma possível execução dos algoritmos baseados em grafo.

Exemplo 13 (Execução do algoritmo *IncGraphTopk*)**.** Considere novamente a relação `acoes` e a teoria-pc Γ do Exemplo 12. No instante i, o BTG da relação é representado pelas seguintes estruturas:

- $Anc_{\#}(t_1) = Anc_{\#}(t_6) = \{\}$;
- $Anc_{\#}(t_2) = \{t_1\}$;
- $Anc_{\#}(t_3) = Anc_{\#}(t_4) = \{t_1, t_2\}$;
- $Anc_{\#}(t_5) = \{t_1, t_2, t_3\}$;
- $Succ_{\#}(t_1) = \{t_2, t_3, t_4, t_5\}$;
- $Succ_{\#}(t_2) = \{t_3, t_4, t_5\}$;
- $Succ_{\#}(t_3) = \{t_5\}$
- $Succ_{\#}(t_4) = Succ_{\#}(t_5) = Succ_{\#}(t_6) = \{\}$;
- $Best = \{t_1, t_6\}$.

Considere novamente a remoção das tuplas t_3 e t_6 no instante $i + 1$. A execução do algoritmo *GraphDelete* ocorre da seguinte maneira:

1) Remoção da tupla t_6:

   a) A tupla t_6 é removida do conjunto $Best$;
   b) Nenhuma aresta é alterada pelo fato de t_6 ser incomparável às demais tuplas;
   c) A tupla t_6 é removida das estruturas $Anc_{\#}$ e $Succ_{\#}$.

2) Remoção da tupla t_3:

a) Como as tuplas t_1 e t_2 dominam t_3 ($Anc_{\#}(t_3) = \{t_1, t_2\}$), o algoritmo remove t_3 de $Succ_{\#}(t_1)$ e de $Succ_{\#}(t_2)$. Portanto, $Succ_{\#}(t_1) = \{t_2, t_3, t_4, t_5\} - \{t_3\} = \{t_2, t_4, t_5\}$ e $Succ_{\#}(t_2) = \{t_3, t_5\} - \{t_3\} = \{t_5\}$;
b) Como t_3 domina t_5 ($Succ_{\#}(t_3) = \{t_5\}$), o algoritmo remove t_3 de $Anc_{\#}(t_5)$. Portanto, $Anc_{\#}(t_5) = \{t_1, t_2, t_3\} - \{t_3\} = \{t_1, t_2\}$;
c) A tupla t_3 é removida das estruturas $Anc_{\#}$ e $Succ_{\#}$.

Já a execução do algoritmo *GraphInsert* acontece do seguinte modo:

1) Inserção da tupla t_7:
   a) A tupla t_7 domina a tupla t_5, então $Anc_{\#}(t_5) = \{t_1, t_2\} \cup \{t_7\} = \{t_1, t_2, t_7\}$ e $Succ_{\#}(t_7) = \{t_5\}$;
   b) Como t_7 não é dominada, a mesma é adicionada ao conjunto *Best*.
2) Inserção da tupla t_8:
   a) A tupla t_8 é dominada pela tupla t_1, então $Anc_{\#}(t_8) = \{t_1\}$ e $Succ_{\#}(t_1) = \{t_2, t_4, t_5\} \cup \{t_8\} = \{t_2, t_4, t_5, t_8\}$;
   b) A tupla t_8 domina a tupla t_4, então $Anc_{\#}(t_4) = \{t_1, t_2\} \cup \{t_8\} = \{t_1, t_2, t_8\}$ e $Succ_{\#}(t_8) = \{t_4\}$;
   c) A tupla t_8 domina a tupla t_5, então $Anc_{\#}(t_5) = \{t_1, t_2, t_7\} \cup \{t_8\} = \{t_1, t_2, t_7, t_8\}$ e $Succ_{\#}(t_8) = \{t_4\} \cup \{t_5\} = \{t_4, t_5\}$;
   d) A tupla t_8 não é adicionada ao conjunto *Best* pelo fato de ter sido dominada.

Por fim, a execução do algoritmo *IncGraphTopk* é realizada da seguinte maneira:

1) O algoritmo cria as cópias $Anc'_{\#}$ e $Best'$ para as estruturas $Anc_{\#}$ e *Best*, respectivamente. O algoritmo também cria a lista L;
2) Como L está vazia, o laço inicia as iterações;
3) Na primeira iteração, as tuplas de $Best'$ são inseridas em L. Além disto, as tuplas t_2 e t_8 passam a compor o conjunto $Best'$ na próxima iteração;
4) Na segunda iteração as tuplas t_2 e t_8 são adicionadas à lista L e o conjunto $Best'$ para a próxima iteração contém as tuplas t_4 e t_5;
5) Como L possui mais de três tuplas o laço é interrompido e o algoritmo retorna as três primeiras tuplas da lista $L = [t_1, t_7, t_2, t_8]$.

Análise de complexidade

A complexidade do algoritmo *GraphDelete* é de $O(n\delta^- + n \times |Anc_{\#}(t)| + n \times |Succ_{\#}(t)|)$ onde $n = |R|$ e $\delta^- = |\Delta^-|$. Assumindo também um fator de limitação para as estruturas $|Anc_{\#}(t)|$ e $|Succ_{\#}(t)|$, assim como foi feito para a lista de antecessores, a complexidade do algoritmo *GraphDelete* é de $O(n\delta^-)$. Já a complexidade do algoritmo *GraphInsert* é de $O(nm^m\delta^+)$, onde $\delta^+ = |\Delta^+|$, $n = |R|$ e $m = |\Gamma|$.

A complexidade do algoritmo *IncGraphTopk* depende do número de iterações dos laços. O número de iterações do laço mais externo é igual ao nível de preferência máximo da teoria-pc. No pior caso, $|Best| = |R| = n$, portanto o custo do algoritmo é de $O(mn \times |Succ_{\#}(t)|)$. Assumindo novamente um fator de limitação para $Succ_{\#}(t)$, a complexidade final do algoritmo *IncGraphTopk* é de $O(nm)$. Observe que os algoritmos baseados em grafo possuem a mesma complexidade dos algoritmos baseados em lista de antecessores.

4.3 Algoritmos baseados em particionamento

O Capítulo 3 apresentou a técnica de particionamento para processamento de consultas CPrefSQL em bancos de dados tradicionais. Esta técnica utiliza uma base de conhecimento construída sobre a teoria-pc para agrupar as tuplas em partições. Dentro de uma partição é possível determinar quais tuplas são dominantes ou dominadas sem a necessidade de comparação com as demais tuplas. Ao final, as tuplas que não foram dominadas em nenhuma partição são retornadas como dominantes. Esta seção apresenta os algoritmos relacionados à segunda etapa do trabalho descrito nesta tese (conforme descrito na Seção 1.4). Estes algoritmos funcionam de forma incremental e mantêm em memória uma hierarquia de preferência baseada na técnica de particionamento para realizar o processamento de consultas contínuas contendo preferências condicionais.

A hierarquia de preferência usada pelos algoritmos é representada pelas estruturas $Pref_{\#}$, $NPref_{\#}$ e $Count_{\#}$. A estrutura $Pref_{\#}(p)$ representa a quantidade de tuplas com valores preferidos na partição p. A estrutura $NPref_{\#}(p)$ é o conjunto de tuplas com valores não preferidos em p. Já a estrutura $Count_{\#}(t)$ representa o número de partições onde t foi dominada. Dada uma tupla t e uma comparação b_i, o identificador de partição para t e b_i é a tupla $p = (i, t/W_{b_i})$.

Operação de remoção

O algoritmo *PartitionDelete* (Algoritmo 15) realiza a tarefa de remoção. Para cada tupla t removida e para cada comparação b_i, o algoritmo obtém a partição p correspondente. Em seguida, o algoritmo verifica se t é uma tupla preferida ($t \models F^{+}_{b_i}$) ou uma tupla não preferida ($t \models F^{-}_{b_i}$) em p.

Quando t é uma tupla preferida, o contador $Pref_{\#}(p)$ é decrementado. Se $Pref_{\#}(p)$ atingir zero, então o algoritmo precisa atualizar o contador $Count_{\#}$ das tuplas do conjunto $NPref_{\#}$, uma vez que estas tuplas deixam de ser dominadas em p. Caso o contador $Count_{\#}(t')$ chegue a zero, significa que a tupla t' deixou de ser dominada e a mesma é adicionada ao conjunto de tuplas dominantes *Best*.

Algoritmo 15 – *PartitionDelete*(K_Γ, Δ^-)

Entrada: Base de conhecimento K_Γ e conjunto de tuplas removidas Δ^-
Atualização: Estruturas de partições $Pref_\#$, $NPref_\#$ e $Count_\#$, conjunto de tuplas dominantes *Best*

```
1: for all t ∈ Δ⁻ do                                          //Para cada tupla t inserida
2:     for all b_i ∈ K_Γ do                                    //Para cada comparação b_i de K_Γ
3:         p ← (i, t/W_{b_i})                                  //Obtém o identificador de partição p
4:         if t ⊨ F⁺_{b_i} then                                //Testa se t possui os valores preferidos
5:             Pref_#(p) − −                                   //Decrementa quantidade de tuplas preferidas para p
6:             if Pref_#(p) = 0 then                           //Verifica se não existem mais tuplas preferidas em p
7:                 for all t' ∈ NPref_#(p) do                  //Para cada tupla não preferida t' em p
8:                     Count_#(t') − −                         //Decrementa a quantidade de vezes que t' foi dominada
9:                     if Count_#(t') = 0 then                 //Testa se t' deixou de ser dominada
10:                        Best ← Best ∪ {t'}                  //Adiciona t' ao conjunto de tuplas dominantes
11:        else if t ⊨ F⁻_{b_i} then                           //Verifica se t é uma tupla não preferida
12:            NPref_#(p) ← NPref_#(p) − {t}                   //Remove t do conjunto de tuplas não preferidas de p
```

Operação de inserção

O algoritmo *PartitionInsert* (Algoritmo 16) é responsável pela tarefa de inserção. Para cada tupla inserida $t \in \Delta^+$ e para cada comparação $b_i \in K_\Gamma$, o algoritmo obtém a partição correspondente e verifica se t é preferida ou não preferida em p.

Algoritmo 16 – *PartitionInsert*(K_Γ, Δ^+)

Entrada: Base de conhecimento K_Γ e conjunto de tuplas removidas Δ^+
Atualização: Estruturas de partições $Pref_\#$, $NPref_\#$ e $Count_\#$, conjunto de tuplas dominantes *Best*

```
1: for all t ∈ Δ⁺ do                                          //Para cada tupla t inserida
2:     for all b_i ∈ K_Γ do                                    //Para cada comparação b_i de K_Γ
3:         p ← (i, t/W_{b_i})                                  //Obtém o identificador de partição p
4:         if t ⊨ F⁺_{b_i} then                                //Testa se t possui os valores preferidos
5:             Pref_#(p) + +                                   //Incrementa a quantidade de tuplas preferidas em p
6:             if Pref_#(p) = 1 then                           //Verifica se t é a primeira tupla a ser preferida em p
7:                 for all t' ∈ NPref_#(p) do                  //Para cada tupla t' não preferida em p
8:                     Count_#(t') + +                         //Incrementa a quantidade de vezes que t' foi dominada
9:                     if Count_#(t') = 1 then                 //Testa se t' era dominante
10:                        Best ← Best − {t'}                  //Remove t' do conjunto de tuplas dominantes
11:        else if t ⊨ F⁻_{b_i} then                           //Verifica se t possui os valores não preferidos
12:            NPref_#(p) ← NPref_#(p) ∪ {t}                   //Adiciona t ao conjunto de tuplas não preferidas de p
13:            if Pref_#(p) > 0 then                           //Testa se p possui tuplas dominantes
14:                Count_#(t) + +                              //Incrementa a quantidade de vezes que t foi dominada
15:    if Count_#(t) = 0 then                                  //Verifica se t não foi dominada
16:        Best ← Best ∪ {t}                                   //Acrescenta t ao conjunto de tuplas dominantes
```

Quando t é preferida em p, o algoritmo incrementa o contador $Pref_\#(p)$. Se $Pref_\#(p)$ é igual a um, o algoritmo também incrementa o contador $Count_\#(t')$ de todas as tuplas $t' \in NPref_\#(p)$. Isto é necessário porque quando $Pref_\#(p) = 0$, todas as tuplas são

dominantes na partição p. Caso contrário, se $Pref_{\#}(p)$ for pelo menos um, existem tuplas dominantes em p e as tuplas de $NPref_{\#}(p)$ passam a ser dominadas.

Se t é uma tupla não preferida, o algoritmo adiciona t ao conjunto $NPref_{\#}(p)$. Quando há tuplas dominantes em p, o contador $Count_{\#}(t)$ é incrementado. Após percorrer todas as comparações, o algoritmo testa se t não foi dominada, $Count_{\#}(t) = 0$, neste caso t é inserida no conjunto *Best*.

Obtenção das top-k tuplas

O algoritmo *IncPartitionTopk* (Algoritmo 17) utiliza as estruturas de partições para processar o operador **TOPK** de forma incremental. Primeiro, o algoritmo copia a estrutura $Pref_{\#}$ para $Pref'_{\#}$ e a relação R para S. Isto é necessário para manter os dados originais inalterados após a execução do algoritmo. Em seguida, o algoritmo chama a função *Remove* (Algoritmo 18) para remover as tuplas dominantes de S.

Algoritmo 17 – *IncPartitionTopk*(K_Γ, R, k)

Entrada: Base de conhecimento K_Γ, relação R e número de top-k tuplas desejadas k
Saída: Top-k tuplas da relação R

```
1: Pref'_# ← Pref_#                          //Copia Pref_# para Pref'_#
2: S ← R                                     //Copia R para S
3: Remove(K_Γ, S, Pref'_#, Best)             //Faz a operação S − Best e atualiza Pref'_#
4: L ← NewList(Best)                         //Inicializa lista L com as tuplas de Best
5: while (|L| < k) and (|S| > 0) do          //Enquanto |L| < k e |S| > 0
6:     D ← GetDominant(K_Γ, S, Pref'_#)      //Armazena as tuplas do nível atual em D
7:     L.append(D)                           //Insere as tuplas de D no final de L
8: return L.getFirst(k)                      //Retorna as k primeiras tuplas de L
```

A lista L é usada para ordenar as tuplas processadas de acordo com o nível de preferência. Desta maneira, as tuplas de nível i sempre precedem as tuplas de nível $i + 1$ em L. O algoritmo inicializa a lista L com o conjunto de tuplas dominantes que possuem nível zero. A cada iteração do laço, o algoritmo obtém as tuplas do próximo nível por meio da função *GetDominant* (Algoritmo 19) e as insere em L.

A função *Remove* remove de S as tuplas que estão presentes em $Best'$ ($S = S - Best'$). Além disto, para cada comparação $b_i \in K_\Gamma$, se a tupla $t \in Best'$ for preferida em p, a função decrementa o contador $Pref'_{\#}(p)$, uma vez que t está sendo removida.

A função *GetDominant* varre o conjunto S em busca de tuplas dominantes. Lembrando que uma tupla t é dominante quando a mesma não é dominada em nenhuma partição. Observe que, no final, a função *Remove* é chamada para remover a tuplas dominantes de S. Isto garante a atualização de S e $Pref'_{\#}$ para obter as tuplas do próximo nível. O Exemplo 14 apresenta uma execução do algoritmo *IncPartitionTopk* ao processar o operador **TOPK**.

Algoritmo 18 – *Remove*$(K_\Gamma, S, \mathit{Pref}'_\#, \mathit{Best}')$

Entrada: Base de conhecimento K_Γ e conjunto de tuplas Best'
Atualização: Conjunto de tuplas S, estrutura de partição $\mathit{Pref}'_\#$

```
1: for all t ∈ Best' do                          //Para cada tupla t em Best'
2:     S ← S − {t}                               //Remove t de S
3:     for all b_i ∈ K_Γ do                      //Para cada comparação b_i de K_Γ
4:         p ← (i, t/W_{b_i})                    //Obtém o identificador de partição p
5:         if t ⊨ F^+_{b_i} then                 //Verifica se t é uma tupla preferida em p
6:             Pref'_#(p) − −                    //Decrementa contador de tuplas preferidas em p
```

Algoritmo 19 – *GetDominant*$(K_\Gamma, S, \mathit{Pref}'_\#)$

Entrada: Base de conhecimento K_Γ
Atualização: Conjunto de tuplas S, estrutura de partições $\mathit{Pref}'_\#$
Saída: Tuplas dominantes em S

```
1: Best' ← {}                                    //Cria conjunto para armazenar as tuplas dominantes de S
2: for all t ∈ S do                              //Para cada tupla t de S
3:     dominated ← false                         //Pressupõe que t não será dominada
4:     for all b_i ∈ K_Γ do                      //Para cada comparação b_i
5:         p ← (i, t/W_{b_i})                    //Obtém o identificador de partição p
6:         if (t ⊨ F^−_{b_i}) and (Pref'_#(p) > 0) then   //Verifica se t é dominada em p
7:             dominated ← true                  //Marca t como dominada
8:             break
9:     if not dominated then                     //Testa se t é dominante
10:        Best' ← Best' + {t}                   //Adiciona t ao conjunto de tuplas dominantes
11: Remove(K_Γ, S, Pref'_#, Best')               //Remove as tuplas dominantes de S
12: return Best'                                 //Retorna as tuplas dominantes
```

Exemplo 14 (Execução do algoritmo *IncPartitionTopk*)**.** Considere novamente a relação `acoes` e a teoria-pc Γ do Exemplo 12. A base de conhecimento K_Γ construída sobre a teoria-pc Γ é composta pelas seguintes comparações:

b_1 : $(\texttt{taxa} = A) \wedge (\texttt{pais} = \mathit{Brasil}) \wedge (\texttt{preco} = \mathit{médio}) \succ$
$(\texttt{taxa} = B) \wedge (\texttt{preco} = \mathit{alto})[\texttt{pais}, \texttt{nome}, \texttt{preco}, \texttt{taxa}]$;

b_2 : $(\texttt{taxa} = A) \wedge (\texttt{preco} = \mathit{médio}) \succ$
$(\texttt{taxa} = B) \wedge (\texttt{pais} = \mathit{EUA}) \wedge (\texttt{preco} = \mathit{alto})[\texttt{pais}, \texttt{nome}, \texttt{preco}, \texttt{taxa}]$;

b_3 : $(\texttt{taxa} = A) \wedge (\texttt{preco} = \mathit{médio}) \succ$
$(\texttt{taxa} = B) \wedge (\texttt{pais} = \mathit{Brasil}) \wedge (\texttt{preco} = \mathit{alto})[\texttt{pais}, \texttt{nome}, \texttt{preco}, \texttt{taxa}]$;

b_4 : $(\texttt{preco} = \mathit{médio}) \succ$
$(\texttt{preco} = \mathit{alto})[\texttt{pais}, \texttt{nome}, \texttt{preco}]$;

b_5 : $(\texttt{taxa} = A) \wedge (\texttt{pais} = \mathit{Brasil}) \wedge (\texttt{preco} = \mathit{alto}) \succ$
$(\texttt{taxa} = B) \wedge (\texttt{pais} = \mathit{EUA}) \wedge (\texttt{preco} = \mathit{alto})[\texttt{pais}, \texttt{nome}, \texttt{taxa}]$;

b_6 : $(\texttt{taxa} = A) \wedge (\texttt{pais} = \mathit{Brasil}) \succ$
$(\texttt{taxa} = B) \wedge (\texttt{pais} = \mathit{Brasil})[\texttt{nome}, \texttt{taxa}]$;

b_7 : $(\texttt{pais} = \mathit{Brasil}) \wedge (\texttt{preco} = \mathit{alto}) \succ$
$(\texttt{pais} = \mathit{EUA}) \wedge (\texttt{preco} = \mathit{alto})[\texttt{pais}, \texttt{nome}]$;

Considere também a remoção das tuplas t_3 e t_6 e a inserção das tuplas t_7 e t_8 da Figura 15(a) no instante $i+1$. A execução da operação de remoção pelo algoritmo *PartitionDelete* no instante $i+1$ ocorre da seguinte maneira:

1) Remoção da tupla t_3:

   a) Considerando a comparação b_1, o identificador da partição é $p = (1)$. A tupla t_3 é uma tupla não preferida ($t_3 \models F^-_{b_1}$), portanto $NPref_{\#}(p) = \{t_3, t_5\} - \{t_3\} = \{t_5\}$;
   b) Na comparação b_2, $p = (2)$ e a tupla t_3 é ignorada porque não satisfaz a fórmula $F^+_{b_2}$ nem a fórmula $F^-_{b_2}$;
   c) Na comparação b_3, $p = (3)$ e $t_3 \models F^-_{b_3}$. Portanto $NPref_{\#}(p) = \{t_3\} - \{t_3\} = \{\}$;
   d) No caso da comparação b_4 há duas partições $(4, \texttt{taxa} : A)$ e $(4, \texttt{taxa} : B)$. A tupla t_3 pertence á partição $p = (4, \texttt{taxa} : B)$. Como $t_3 \models F^-_{b_4}$, $NPref_{\#}(p) = \{t_3, t_5\} - \{t_3\} = \{t_5\}$;
   e) Na comparação b_5, t_3 é ignorada;
   f) Na comparação b_6, t_3 pertence à partição $p = (6, \texttt{pais} : \textit{Brasil}, \texttt{preco} : \textit{alto})$. Portanto $NPref_{\#}(p) = \{t_3\} - \{t_3\} = \{\}$;
   g) Na comparação b_7, t_3 pertence à partição $p = (7, \texttt{taxa} : A, \texttt{preco} : \textit{alto})$. Logo $t_3 \models F^+_{b_7}$, $Pref_{\#}(p) = 1 - 1 = 0$. Como $Pref_{\#}(p)$ chegou a zero, o algoritmo decrementa o contador $Count_{\#}$ da tupla t_5 que pertence a $NPref_{\#}(p)$.

2) A tupla t_6 é ignorada em todas as partições.

A execução do algoritmo *PartitionInsert* acontece do seguinte modo:

1) Inserção da tupla t_7:

   a) Nas comparações b_1, b_2 e b_3, t_7 é ignorada;
   b) Na comparação b_4, t_7 pertence á partição $p = (4, \texttt{taxa} : B)$. Portanto $t_7 \models F^+_{b_4}$, $Pref_{\#}(p) = 0 + 1 = 1$. Como $Pref_{\#}(p) = 1$, o algoritmo incrementa o contador $Count_{\#}$ da tupla t_5 que pertence a $NPref_{\#}(p)$;
   c) Nas comparações b_5, b_6 e b_7, a tupla t_7 também é ignorada;
   d) A tupla t_7 é adicionada no conjunto *Best* pelo fato de não ter sido dominada.

2) Inserção da tupla t_8:

   a) Nas comparações b_1, b_2 e b_3, a tupla t_8 é ignorada;
   b) Na comparação b_4, t_8 pertence á partição $p = (4, \texttt{taxa} : A)$. Portanto $t_8 \models F^-_{b_4}$, $NPref_{\#}(p) = \{t_2, t_4\} \cup \{t_8\} = \{t_2, t_4, t_8\}$. Como $Pref_{\#}(p) > 0$, o algoritmo incrementa o contador $Count_{\#}$ da tupla t_8;
   c) Na comparação b_5, t_8 pertence à partição $p = (5, \texttt{preco} : \textit{alto})$. Logo $t_8 \models F^+_{b_5}$ e $Pref_{\#}(p) = 1 + 1 = 2$;
   d) Na comparação b_6, t_8 pertence à partição $p = (6, \texttt{pais} : \textit{Brasil}, \texttt{preco} : \textit{alto})$. Assim $t_8 \models F^+_{b_6}$ e $Pref_{\#}(p) = 1 + 1 = 2$;

e) Na comparação b_7, t_8 pertence à partição $p = (7, \texttt{taxa} : A, \texttt{preco} : \textit{alto})$. Como $t_8 \models F^+_{b_7}$, $\textit{Pref}_{\#}(p) = 1 + 1 = 2$.

A Figura 16 apresenta as estruturas de partições $\textit{Pref}_{\#}$ e $\textit{NPref}_{\#}$ construídas pelos algoritmos nos instantes i e $i + 1$.

Partição (p)	**Instante** i		**Instante** $i+1$	
	$\textit{Pref}_{\#}$	$\textit{NPref}_{\#}$	$\textit{Pref}_{\#}$	$\textit{NPref}_{\#}$
(1)	1	$\{t_3, t_5\}$	1	$\{t_5\}$
(2)	1	$\{t_5\}$	1	$\{t_5\}$
(3)	1	$\{t_3\}$	1	$\{\}$
$(4, \texttt{taxa} : A)$	1	$\{t_2, t_4\}$	1	$\{t_2, t_4, t_8\}$
$(4, \texttt{taxa} : B)$	0	$\{t_3, t_5\}$	1	$\{t_5\}$
$(5, \texttt{preco} : \textit{alto})$	1	$\{t_5\}$	2	$\{t_5\}$
$(6, \texttt{pais} : \textit{Brasil}, \texttt{preco} : \textit{alto})$	1	$\{t_3\}$	2	$\{\}$
$(6, \texttt{pais} : \textit{Brasil}, \texttt{preco} : \textit{médio})$	1	$\{\}$	1	$\{\}$
$(7, \texttt{taxa} : A, \texttt{preco} : \textit{alto})$	1	$\{t_4\}$	2	$\{t_4\}$
$(7, \texttt{taxa} : B, \texttt{preco} : \textit{alto})$	1	$\{t_5\}$	0	$\{t_5\}$

Figura 16 – Estruturas de partição $\textit{Pref}_{\#}$ e $\textit{NPref}_{\#}$ construídas pelos algoritmos baseados em particionamento

A execução do algoritmo *IncPartitionTopk* para processar a operação $\textbf{TOPK}_{\Gamma,3}(\texttt{acoes})$ no instante $i + 1$ ocorre da seguintes maneira:

1) O algoritmo cria as cópias $\textit{Pref}'_{\#}$ e S a partir de $\textit{Pref}_{\#}$ e R, respectivamente e chama a função *Remove*;
2) Execução da função $\textit{Remove}(S, \textit{Pref}'_{\#}, \textit{Best})$:
   a) As tuplas t_1 e t_7 pertencentes ao conjunto *Best* são removidas de S;
   b) Nas comparações b_1, b_2, b_3, b_4 e b_6, a tupla t_1 atende aos valores preferidos das partições (1), (2), (3), $(4, \texttt{taxa} : A)$ e $(6, \texttt{pais} : \textit{Brasil}, \texttt{preco} : \textit{médio})$, respectivamente. Portanto os contadores $\textit{Pref}'_{\#}$ destas partições são decrementados;
   c) A tupla t_7 pertence apenas à partição $p = (4, \texttt{taxa} : B)$ na partição b_4, assim $\textit{Pref}'_{\#}(p) = 1 - 1 = 0$;
3) As tuplas t_1 e t_7 são inseridas à lista L e o algoritmo *IncPartitionTopk* inicia as iterações do laço;
4) Na primeira iteração, o algoritmo chama a função *GetDominant*.
5) Execução da função $\textit{GetDominant}(S, \textit{Pref}'_{\#})$:
   a) A função inicializa o conjunto $\textit{Best}'$ e varre as tuplas de $S = \{t_2, t_4, t_5, t_8\}$;
   b) As tuplas t_2 e t_8 não são dominadas em nenhuma partição e são adicionadas ao conjunto *Best*;
   c) No final a função *Remove* é chamada para remover as tuplas t_2 e t_8 de S e atualizar a estrutura $\textit{Pref}'_{\#}$.
6) O algoritmo *IncPartitionTopk* recebe as tuplas t_2 e t_8 da função *GetDominant* e as adiciona à lista L

7) Neste momento $|L| > 3$ e o algoritmo interrompe o laço de repetição;
8) Ao final, o algoritmo retornas as tuplas t_1, t_7 e t_2 que ocupam as três primeiras posições da lista $L = [t_1, t_7, t_2, t_8]$.

Análise de complexidade

O algoritmo *PartitionDelete* varre o conjunto Δ^- e, para cada tupla $t \in \delta^-$, todas as comparações da base de conhecimento K_Γ são analisadas. Como foi explicado no Capítulo 3, o tamanho da base de conhecimento é $O(m^{4l})$. Além disto, há um terceiro laço de repetição aninhado que percorre as tuplas da estrutura $NPref_\#$. Assim, a complexidade do algoritmo *PartitionDelete* é de $O(m^{4l} \times \delta^- \times |NPref_\#(p)|)$, onde $m = |\Gamma|$ é o número de regras, l é o número de atributos da relação R e $\delta^- = |\Delta^-|$ é o número de tuplas removidas. Considerando um fator de limitação para o tamanho de $NPref_\#(p)$ assim como foi feito para lista de antecessores, a complexidade do algoritmo *PartitionDelete* é de $O(m^{4l}\delta^-)$.

O custo do algoritmo *PartitionInsert* é análogo ao custo do algoritmo *PartitionDelete*. Exitem três laços de repetição aninhados percorrendo as tuplas de Δ^+, as comparações de K_Γ e as tuplas de $NPref_\#$. Desta maneira, a complexidade do algoritmo *PartitionInsert* é de $O(|NPref_\#(p)| \times m^{4l}\delta^+)$. Novamente, considerando um fator de limitação para o tamanho do conjunto $NPref_\#(p)$, a complexidade final do algoritmo é de $O(m^{4l}\delta^+)$.

Para analisar a complexidade do algoritmo *IncPartitionTopk* é preciso considerar o custo das funções *Remove* e *GetDominant*. Na função *Remove*, para cada tupla $t \in Best'$, é preciso remover t de S e varrer as comparações de K_Γ para atualizar a estrutura $Pref'_\#$. O conjunto S pode ser implementado por meio de uma tabela *hash*, isto possibilita que a operação $S.remove(t)$ seja feita com custo amortizado de $O(1)$. No pior caso, $|Best'| = |R| = n$, portanto o custo da função *Remove* é de $O(n \times |K_\Gamma|)$. A função *GetDominant* realiza uma varredura em S e, para cada tupla $t \in S$, percorre as comparações de K_Γ. Assim, o custo da função *GetDominant* é de $O(n \times |K_\Gamma|)$.

O número de chamadas à função *GetDominant* pelo algoritmo *IncPartitionTopk* está relacionado com o número de iterações do laço. No pior caso, este número é $|\Gamma| = m$. Logo, a complexidade do algoritmo *IncPartitionTopk* é de $O(nm^{4l+1})$, onde $n = |R|$ e $m^{4l} = |K_\Gamma|$.

A construção da base de conhecimento não é uma tarefa trivial e pode consumir um tempo considerável conforme mostrado no Capítulo 8. Uma otimização interessante para os algoritmos baseados em particionamento é a persistência da base de conhecimento para processamento de consultas no futuro. Desta maneira, os usuários podem utilizar bases de conhecimento já computadas para processamento de consultas no futuro.

4.4 Considerações finais

A principal contribuição apresentada neste capítulo foi o algoritmo incremental *IncPartitionBest* que utiliza uma hierarquia de preferência baseada em partições. O algoritmo *IncPartitionBest* trabalha de forma incremental e mantém uma hierarquia de preferência baseada na técnica de particionamento descrita no Capítulo 3. Este capítulo abordou também os algoritmos estado-da-arte para processamento de consultas contínuas contendo preferências condicionais. O capítulo analisou os algoritmos incrementais *IncAncestorsBest* proposto em Amo e Bueno (2011) e *IncGraphBest* proposto em Petit et al. (2012).

O presente capítulo apresentou uma análise de complexidade dos algoritmos incrementais considerando as tarefas de remoção, de inserção e de obtenção das top-k tuplas. O próximo capítulo apresentará o modelo de preferência que será usado nos operadores de preferência temporal da linguagem StreamPref. As técnicas de processamento de consultas usadas no capítulo atual serviram de base para o desenvolvimento de algoritmos eficientes no processamento de consultas contínuas contendo preferências temporais. Mais adiante, o Capítulo 8 apresenta experimentos comparativos entre os algoritmos *IncAncestorsBest*, *IncGraphBest* e *IncPartitionBest*. Os resultados dos experimentos mostram que o algoritmo *IncPartitionBest* possui melhor performance em relação aos demais algoritmos.

Capítulo 5

Preferências temporais

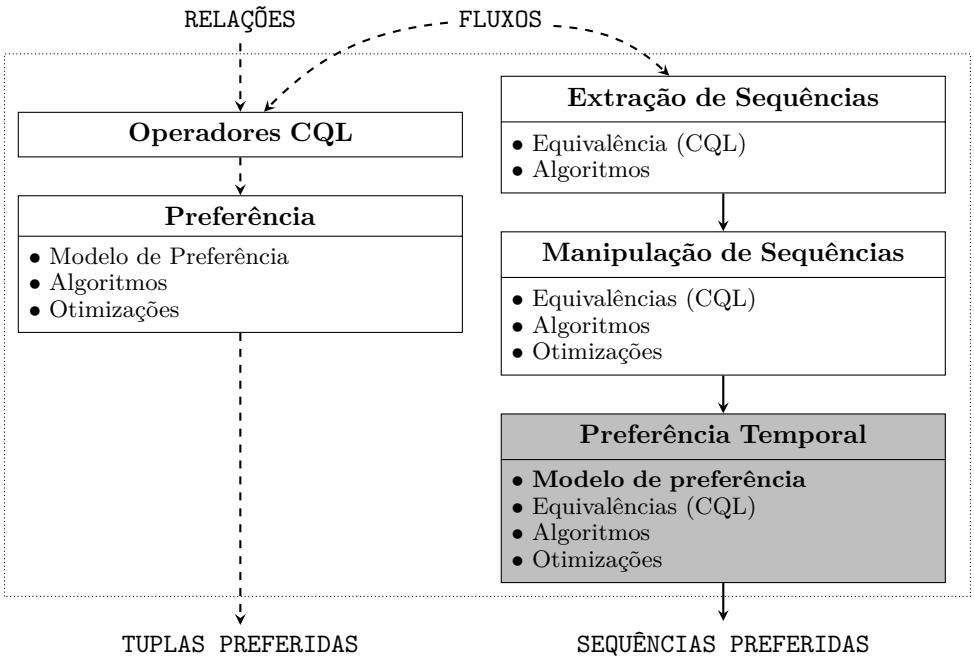


Figura 17 – Visão geral da linguagem StreamPref destacando modelo de preferência temporal

A terceira etapa do trabalho descrito nesta tese está relacionada com o especificação de novos operadores para processamento de consultas contínuas contendo preferências condicionais temporais. Dentre estes operadores estão os operadores de preferência que devem ser capazes de selecionar as *melhores sequências* de acordo com as preferências do usuário. Para selecionar as melhores sequências, os operadores de preferência precisam comparar sequências obedecendo a um certo modelo de preferência. Este modelo de preferência especifica como as preferências devem ser expressas e como estas preferências são usadas para comparar sequências. O capítulo atual apresenta o modelo de preferência temporal adotado pela linguagem StreamPref conforme destacado (em cinza) na Figura 17.

Este capítulo está organizado como se segue. A Seção 5.1 define a estrutura de dados de sequência que será usada pelas preferências temporais condicionais. A Seção 5.2 apresenta o modelo de preferência utilizado pela linguagem StreamPref, neste modelo as preferências temporais são expressas por um conjunto de regras denominado *teoria de preferências condicionais temporais*, ou simplesmente teoria-pct. Já a Seção 5.3 mostra como as teorias-pct são usadas na comparação de sequências. A Seção 5.4 propõe um teste para verificar a consistência de teorias-pct. A Seção 5.5 realiza uma análise comparativa entre o modelo de preferência da linguagem StreamPref e os trabalhos correlatos. Por fim, a Seção 5.6 apresenta as considerações finais sobre o capítulo.

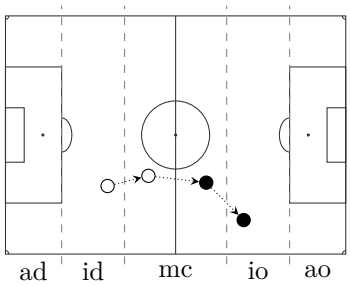
5.1 Sequências

Uma característica marcante dos ambientes de dados em fluxo é o formato sequencial dos dados, ou seja, os itens de dados possuem informações que permitem saber se um elemento ocorreu antes ou depois de outro. Tirando vantagem desta característica, a linguagem StreamPref possibilita mapear elementos de um fluxo de dados para sequências de tuplas (Definição 18).

Definição 18 (Sequência)**.** Seja $R(A_1, ..., A_l)$ um esquema relacional. Uma sequência $s = \langle t_1, ..., t_z \rangle$ sobre R é um conjunto ordenado de tuplas tal que $t_i \in \mathbf{Tup}(R)$ para todo $i \in \{1, ..., z\}$ (RIBEIRO et al., 2017a).

O comprimento de uma sequência s é denotado por $|s|$ e a posição i de s é denotada por $s[i]$. O conjunto de todas as sequências possíveis sobre R é denotado por $\mathbf{Seq}(R)$. Dadas duas sequências $s = \langle t_1, ..., t_z \rangle$ e $s' = \langle t'_1, ..., t'_{z'} \rangle$, a concatenação destas sequências, denotada por $s + s'$, é $s'' = \langle t_1, ..., t_z, t'_1, ..., t'_{z'} \rangle$. A notação $s[i, j]$ denota a subsequência $s = \langle t_i, ..., t_j \rangle$ contida dentro da sequência $s = \langle t_1, ..., t_z \rangle$ tal que $1 \leq i \leq j$ e $j \leq z$. O Exemplo 15 demonstra como elementos de dados podem ser armazenados em uma sequência.

Exemplo 15 (Sequência)**.** Dentro do contexto da aplicação de futebol descrita anteriormente nos Capítulo 1 e 2. Considere o esquema relacional `posicionamento(local, jbola)` tal que `local` é a posição do jogador em campo e `jbola` indica se o jogador tem a posse de bola. O domínio do atributo `local` é composto pelos valores descritos na Figura 1 do Capítulo 1. Já o atributo `jbola` tem o valor 1 quando o jogador possui posse de bola e 0 quando o jogador não tem a posse de bola. O comportamento dos jogadores em relação ao posicionamento e posse de bola podem ser armazenados através de sequências. Como exemplo, considere os quatro posicionamentos do jogador 1 mostrado na Figura 18(a). Estes posicionamentos são representados, instante por instante, pelas sequências da Figura 18(b).



(a) Posicionamentos do jogador 1

Instante 0: $s_1 = \langle(id, 0)\rangle$
Instante 1: $s_1 = \langle(id, 0), (mc, 0)\rangle$
Instante 2: $s_1 = \langle(id, 0), (mc, 0), (mc, 1)\rangle$
Instante 3: $s_1 = \langle(id, 0), (mc, 0), (mc, 1), (io, 1)\rangle$

(b) Sequências

Figura 18 – Posicionamentos de um jogador e as sequências que os representam

No que diz respeito ao modelo de preferência, na linguagem StreamPref as preferências do usuário são expressas por meio de regras, seguindo as linhas do formalismo lógico TPref proposto por Amo e Giacometti (2007). Basicamente, as regras de preferência possuem o formato "*se* determinada condição for verdadeira *então* tenho esta preferência". Como exemplo, um treinador de futebol pode ter a seguinte preferência: "*se* a jogada anterior foi uma recepção *então* prefiro que o jogador passe a bola a driblar". Observe que a preferência do treinador apresenta características temporais pois os valores desejados em um dado momento são afetadas por acontecimentos passados. Estas características possibilitam aplicar as preferências temporais na comparação de sequências de forma que os valores desejados são considerados em uma posição i e os acontecimentos passados são as posições anteriores a i.

5.2 Teorias de preferências condicionais temporais

A especificação das preferências temporais usadas na linguagem StreamPref utilizam fórmulas baseadas na Lógica Temporal Proposicional (LTP) (PRIOR, 1967). Na LTP as fórmulas básicas são variáveis proposicionais. No caso da StreamPref, as fórmulas básicas ou proposições são predicados obedecendo a um dos seguintes formatos:

1) $A\theta a$, onde $a \in \mathbf{Dom}(A)$ e $\theta \in \{<, \leq, =, \neq, \geq, >\}$;
2) $a\theta A\theta a'$, onde $a, a' \in \mathbf{Dom}(A)$, $a < a'$ e $\theta \in \{<, \leq\}$.

Dada uma proposição $Q(A)$, a notação $S_{Q(A)} = \{a \in \mathbf{Dom}(A) \mid a \models Q(A)\}$ denota o conjunto de valores do atributo A representado por $Q(A)$. Os novos formatos de predicados são interessantes para expressar preferências contendo faixas de valores como por exemplo "prefiro contratar jogadores com 11 a 20 gols a jogadores com 5 a 10 gols". A lógica usada pela linguagem StreamPref mantém as mesmas propriedades da LTP, mas usam o novo formato de proposição. A Definição 19 formaliza o conceito de fórmula StreamPref.

Definição 19 (Fórmula StreamPref)**.** Uma *fórmula StreamPref* é definida como se segue:

1) **true** e **false** são fórmulas StreamPref;
2) Se $Q(A)$ é uma proposição sobre o atributo A, então $Q(A)$ é uma fórmula StreamPref;
3) Se F e G são fórmulas StreamPref, então $F \wedge G$, $F \vee G$, $\neg G$ e $\neg F$ são fórmulas StreamPref.
4) Se F e G são fórmulas StreamPref, então F **since** G é uma fórmula StreamPref (RIBEIRO et al., 2017a).

Quanto à semântica, as fórmulas StreamPref são avaliadas de acordo com uma posição de uma sequência. Mais precisamente, uma fórmula StreamPref F é satisfeita por uma sequência $s = \langle t_1, ..., t_z \rangle$ em uma posição $i \in \{1, ..., z\}$, denotado por $(s, i) \models F$, quando:

1) $(s, i) \models Q(A)$ se e somente se existe $s[i].A \models Q(A)$;
2) $(s, i) \models F \wedge G$ se e somente se $(s, i) \models F$ e $(s, i) \models G$;
3) $(s, i) \models F \vee G$ se e somente se $(s, i) \models F$ ou $(s, i) \models G$;
4) $(s, i) \models \neg F$ se e somente se $s[i] \not\models F$;
5) $(s, i) \models F$ **since** G se e somente se existe j onde $1 \leq j < i$ e $(s, j) \models G$ e $(s, z) \models F$ para todo z tal que $j < z \leq i$.

A fórmula **true** sempre é satisfeita e a fórmula **false** nunca é satisfeita.

Para facilitar a especificação de preferências condicionais temporais por parte do usuário, a linguagem StreamPref também possui fórmulas derivadas. Dada uma proposição $Q(A)$, as fórmulas StreamPref derivadas são as seguintes:

Prev $Q(A)$**:** Equivalente a (**false since** $Q(A)$). Esta fórmula é satisfeita em uma posição i, se $Q(A)$ for satisfeita na posição imediatamente anterior a i, ou seja, $(s, i) \models$ **Prev** $Q(A)$ se e somente se $i > 1$ e $(s, i-1) \models Q(A)$;

SomePrev $Q(A)$**:** Equivalente a (**true since** $Q(A)$). Esta fórmula é satisfeita em uma posição i, se $Q(A)$ for satisfeita em qualquer posição anterior a i, ou seja, $(s, i) \models$ **Prev** $Q(A)$ se e somente se existe j tal que $1 \leq j < i$ e $(s, j) \models Q(A)$;

AllPrev $Q(A)$**:** Equivalente a ($\neg$ **SomePrev**$(\neg Q(A))$). Esta fórmula é satisfeita em uma posição i, se $Q(A)$ for satisfeita em todas as posições anteriores a i, ou seja, $(s, i) \models$ **AllPrev** $Q(A)$ se e somente se $i > 1$ e $(s, j) \models Q(A)$ para todo $1 \leq j < i$;

First: Equivalente a ($\neg$ **Prev true**). Esta fórmula é satisfeita somente na primeira posição de uma sequência, ou seja, $(s, i) \models$ **First** se e somente se $i = 1$.

Na linguagem StreamPref as proposições são chamadas de *fórmulas de presente* e as fórmulas derivadas são chamadas de *fórmulas de passado*. Além disto, tanto as fórmulas de passado quanto as fórmulas de presente (chamadas de *fórmulas átomo*) são usadas para compor uma *condição temporal* (Definição 20).

Definição 20 (Condição temporal)**.** Uma condição temporal é uma fórmula StreamPref $C = F_1 \wedge ... \wedge F_p$, onde os termos $F_1, ..., F_p$ são fórmulas átomo. A notação $C^{\leftarrow}$ é a

conjunção das fórmulas de passado que aparecem em C. Já a notação $C^{\bullet}$ é a conjunção das fórmulas de presente que aparecem em C (RIBEIRO et al., 2017a).

Exemplo 16 (Condições temporais)**.** Considere o esquema relacional `posicionamento(local, bola, direcao)` tal que $\mathbf{Dom}(\texttt{bola}) = \{0, 1\}$ e $\mathbf{Dom}(\texttt{direcao}) = \{\textit{frente}, \textit{tras}, \textit{lateral}\}$. O domínio do atributo `local` é composto pelos valores descritos na Figura 1 do Capítulo 1. Os atributos `bola` e `direcao` indicam a posse de bola do time e a direção de movimento do jogador, respectivamente. O valor 1 indica que o time ou jogador tem a posse de bola e o valor 0 indica o contrário. Suponha que um treinador de futebol tenha as seguintes preferências:

[P1] *Se*, em um dado momento, o time tem a posse de bola e, imediatamente antes deste momento, o jogador estava na intermediária defensiva, *então* eu prefiro que este jogador vá para o meio de campo e não permaneça na mesma posição, independentemente da direção de movimento do jogador;

[P2] *Se*, em um dado momento, o time não tem a posse de bola e, imediatamente antes deste momento, o jogador estava na ofensiva intermediária e, em todos os momentos anteriores o time tinha a posse de bola, *então* eu prefiro que o jogador volte para o meio de campo e não fique na intermediária ofensiva;

[P3] Movimentos laterais são melhores do que movimentos para frente.

A condição da preferência **P1** é expressa pela condição temporal $C_1 : (\texttt{bola} = 1) \wedge \mathbf{Prev}(\texttt{local} = id)$, já a condição da preferência **P2** é expressa pela condição temporal $C_2 : (\texttt{bola} = 0) \wedge \mathbf{Prev}(\texttt{local} = io) \wedge \mathbf{AllPrev}(\texttt{bola} = 1)$. A preferência **P3** é uma preferência sem condição ou com condição vazia. Analisando os componentes de C_2, é possível observar que $C_2^{\leftarrow} : \mathbf{Prev}(\texttt{local} = io) \wedge \mathbf{AllPrev}(\texttt{bola} = 1)$ e $C_2^{\bullet} : (\texttt{bola} = 0)$.

Na linguagem StreamPref as preferências são representadas através de *regras de preferências condicionais temporais* (Definição 21) que por sua vez são combinadas para formar *teorias de preferências condicionais temporais* (Definição 22). O Exemplo 17 mostra como as preferências de um usuário são representadas por meio de uma teoria de preferências condicionais temporais.

Definição 21 (Regra de preferências condicionais temporais)**.** Seja R um esquema relacional. Uma regra de preferências condicionais temporais, ou simplesmente regra-pct, sobre R é uma expressão no formato $\varphi : C_\varphi \rightarrow Q_\varphi^+ \succ Q_\varphi^-[W_\varphi]$, onde:

1) As proposições Q_φ^+ e Q_φ^- representam os valores preferidos e os valores não preferidos para o atributo de preferência A_φ, respectivamente, tais que $S_{Q_\varphi^+} \cap S_{Q_\varphi^-} = \{\}$;
2) C_φ é uma condição temporal tal que $\mathbf{Att}(C) \in R$ e $\mathbf{Att}(C_\varphi^{\bullet}) \cap \{A_\varphi\} \cap W_\varphi = \{\}$;
3) W_φ é um subconjunto de R representando os atributos indiferentes tal que $A_\varphi \notin W_\varphi$ (RIBEIRO et al., 2017a).

Definição 22 (Teoria de preferências condicionais temporais)**.** Seja R um esquema relacional. Uma teoria de preferências condicionais temporais, ou simplesmente teoria-pct, sobre R é um conjunto finito de regras-pct sobre R (RIBEIRO et al., 2017a).

Exemplo 17 (Teoria-pct)**.** Considere novamente as preferências **P1**, **P2** e **P3** do Exemplo 16. Estas preferências podem ser representadas pela teoria-pct $\Phi = \{\varphi_1, \varphi_2, \varphi_3\}$ onde:

φ_1 : $(\texttt{bola} = 1) \wedge \mathbf{Prev}(\texttt{local} = id) \rightarrow (\texttt{local} = mc) \succ (\texttt{local} = id)[\texttt{direcao}]$;
φ_2 : $(\texttt{bola} = 0) \wedge \mathbf{Prev}(\texttt{local} = io) \wedge \mathbf{AllPrev}(\texttt{bola} = 1) \rightarrow (\texttt{local} = mc) \succ (\texttt{local} = io)$;
φ_3 : $(\texttt{direcao} = \textit{lateral}) \succ (\texttt{direcao} = \textit{frente})$.

5.3 Comparação de sequências

A comparação de sequências na linguagem StreamPref é feita de maneira análoga à comparação lexicográfica de palavras. O primeiro passo é buscar a primeira posição diferente nas duas sequências. Depois disto, o processo se resume a comparar esta posição diferente. No caso de sequências, a comparação pode ser feita de forma direta com o uso de uma única regra-pct ou por transitividade. Na comparação direta, dadas duas sequências s e s' e uma regra-pct φ, a sequência s é preferida a s', denotado por $s \succ_\varphi s'$, se existe uma posição i tal que:

1) Todas as posições anteriores a i são iguais em ambas sequências, $s[j] = s'[j]$ para todo $j \in \{1, ..., i-1\}$;
2) A condição da regra-pct φ é satisfeita por ambas sequências na posição i, $(s, i) \models C_\varphi$ e $(s', i) \models C_\varphi$;
3) A posição $s[i]$ satisfaz o predicado de valores preferidos, $(s, i) \models Q^+_\varphi$, e a posição $s'[i]$ satisfaz o predicado de valores não preferidos, $(s', i) \models Q^-_\varphi$;
4) As sequências s e s' têm valores coincidentes em todos os atributos da posição i, exceto no atributo de preferência e nos atributos indiferentes, $s[i].A' = s'[i].A'$ para todo $A' \notin (\{A_\varphi\} \cup W_\varphi)$.

Através das comparações diretas é possível observar que uma regra-pct φ sobre R induz a relação de ordem $\succ_\varphi \in \mathbf{Seq}(R) \times \mathbf{Seq}(R)$ tal que $(s, s') \in \succ_\varphi$ se s é preferida a s' de acordo com φ. No caso da comparação por transitividade é necessário que exista um encadeamento de comparações diretas, conforme estabelecido pelo Teorema 5.

Teorema 5. Seja Φ uma teoria-pct sobre um esquema relacional R. Sejam s, s' duas sequências em $\mathbf{Seq}(R)$. A sequência s é preferida a sequência s' de acordo com Φ, denotado por $s \succ_\Phi s'$, se e somente se existem as sequências $s_1, ..., s_{m+1} \in \mathbf{Seq}(R)$ e as regras-pct $\varphi_1, ..., \varphi_m \in \Phi$ tais que $s_1 \succ_{\varphi_1} ... \succ_{\varphi_m} s_{m+1}$, onde $s = s_1$ e $s' = s_{m+1}$.

Em outras palavras, o Teorema 5 diz que, dada uma teoria-pct Φ sobre um esquema relacional R, a relação de ordem induzida por Φ sobre $\mathbf{Seq}(R)$, denotada por $\succ_\Phi$ e chamada de *ordem de preferência temporal*, é o fecho transitivo de $\bigcup_{\varphi\in\Phi} \succ_\varphi$. O Exemplo 18 mostra algumas comparações de sequências considerando uma teoria-pct.

Exemplo 18 (Comparação de sequências)**.** Considere a teoria-pct Φ do Exemplo 17 representando as preferências do treinador. Considere também três jogadores de futebol 1, 2 e 3 cujos posicionamentos foram monitorados durante uma partida de futebol. As sequências a seguir representam o comportamento destes jogadores nos últimos quatro segundos:

$s_1 = \langle(id, 1, lateral), (mc, 1, lateral), (io, 1, lateral), (io, 0, tras)\rangle$;
$s_2 = \langle(id, 1, lateral), (mc, 1, lateral), (io, 1, lateral), (mc, 0, tras)\rangle$;
$s_3 = \langle(id, 1, lateral), (id, 1, tras), (mc, 0, lateral), (mc, 0, tras)\rangle$.

Ao comparar estas sequências usando as regras de Φ, é possível constatar que $s_1 \succ_{\varphi_1} s_3$ (posição 2) e $s_2 \succ_{\varphi_2} s_1$ (posição 4). Portanto, $s_2 \succ_\Phi s_1 \succ_\Phi s_3$. Isto significa que o jogador 2 possui o melhor comportamento de posicionamento considerando as preferências do treinador.

5.4 Teste de consistência

Quando as ordens impostas pelas regras-pct são combinadas pelo fecho transitivo, a ordem de preferência temporal gerada pode ser uma ordem reflexiva. Quando isto acontece a teoria-pct é *inconsistente*. Isto pode levar a situações indesejáveis no contexto de banco de dados, como uma sequência ser preferida a ela mesma. O Exemplo 19 mostra como pode surgir este tipo de inconsistência.

Exemplo 19 (Teoria-pct inconsistente)**.** Considere a teoria-pct $\Phi = \{\varphi_1, \varphi_2, \varphi_3\}$, onde:

φ_1 : $\mathbf{Prev}(\texttt{local} = id) \rightarrow (\texttt{local} = mc) \succ (\texttt{local} = id)$;
φ_2 : $(\texttt{bola} = 1) \rightarrow (\texttt{local} = io) \succ (\texttt{local} = mc)$;
φ_3 : $\mathbf{Prev}(\texttt{bola} = 0) \rightarrow (\texttt{local} = id) \succ (\texttt{local} = io)$.

Considere também as seguintes sequências:

$s_1 = \langle(id, 0, tras), (mc, 0, tras)\rangle$;
$s_2 = \langle(id, 0, tras), (id, 0, tras)\rangle$;
$s_3 = \langle(id, 0, tras), (io, 0, tras)\rangle$.

Com a teoria-pc Φ é possível realizar as comparações $s_1 \succ_{\varphi_1} s_2$, $s_2 \succ_{\varphi_3} s_3$ e $s_3 \succ_{\varphi_2} s_1$, portanto $s_1 \succ_\Phi s_1$. Neste caso, a ordem de preferência temporal $\succ_\Phi$ é reflexiva e a teoria-pct Φ é inconsistente.

Como foi mencionado no início deste capítulo, o modelo de preferência da StreamPref é baseado no formalismo TPref. O formalismo TPref permite a composição de regras-pct usando fórmulas arbitrárias, porém seu teste de consistência é viável considerando apenas um fragmento da linguagem. A restrição a este fragmento foi feita devido a alta complexidade para lidar com fórmulas arbitrárias (AMO; GIACOMETTI, 2007).

No caso da linguagem StreamPref, as regras-pct possuem um formato específico de condições temporais (conjunções de fórmulas átomo). Por outro lado, o teste de consistência especificado no presente trabalho é válido para qualquer teoria-pct da StreamPref. Este teste leva em consideração a compatibilidade temporal entre as fórmulas átomo (Definição 23).

Definição 23 (Fórmulas átomo temporalmente compatíveis)**.** Duas fórmulas átomo F e G são temporalmente compatíveis quando ao menos uma das seguintes condições for verdadeira:

1) Uma das fórmulas é uma proposição;
2) As fórmulas F e G são idênticas;
3) As fórmulas são $F = \mathbf{SomePrev}\, Q(A)$ e $G = \mathbf{SomePrev}\, Q'(A')$;
4) As fórmulas são $F = \mathbf{Prev}\, Q(A)$, $G = \mathbf{Prev}\, Q'(A')$ ou $G = \mathbf{AllPrev}\, Q'(A')$ tais que: (a) $A \neq A'$ ou (b) $A = A'$ e $S_{Q(A)} \cap S_{Q'(A')} \neq \{\}$;
5) As fórmulas são $F = \mathbf{AllPrev}\, Q(A)$, $G = \mathbf{Prev}\, Q'(A')$ ou $G = \mathbf{SomePrev}\, Q'(A')$ ou $G = \mathbf{AllPrev}\, Q'(A')$ tais que: (a) $A \neq A'$ ou (b) $A = A'$ e $S_{Q(A)} \cap S_{Q'(A')} \neq \{\}$ (RIBEIRO et al., 2017a).

A Tabela 1 apresenta a relação de compatibilidade temporal entre as fórmulas átomos, onde $Q(A)$ e $Q'(A')$ são proposições. uma das seguintes condições for satisfeita: O símbolo * indica que as fórmulas são temporalmente incompatíveis apenas se $A = A'$ e $s_{Q(A)} \cap S_{Q'(A)} = \{\}$.

Tabela 1 – Compatibilidade temporal entre fórmulas átomo

	$Q(A)$	**First**	**Prev** $Q(A)$	**SomePrev** $Q(A)$	**AllPrev** $Q(A)$
$Q'(A')$	✓	✓	✓	✓	✓
First	✓	✓	×	×	×
Prev $Q'(A')$	✓	×	*	✓	*
SomePrev $Q'(A')$	✓	×	✓	✓	*
AllPrev $Q'(A')$	✓	×	*	*	*

Através da compatibilidade temporal entre fórmulas átomo é possível estabelecer a compatibilidade temporal entre regras-pct (Definição 24). Dada uma teoria-pct Φ e uma regra-pct $\varphi \in \Phi$, a compatibilidade temporal é usada para extrair o subconjunto de regras-pct temporalmente compatíveis com φ, denotado por Φ_φ.

Definição 24 (Regras-pct temporalmente compatíveis)**.** Sejam φ e φ' duas regras-pct tais que $C_\varphi : F_1 \wedge ... \wedge F_p$ e $C_{\varphi'} : F'_1 \wedge ... \wedge F'_q$. As regras-pct φ e φ' são temporalmente compatíveis se as fórmulas átomo F_i e F'_j são temporalmente compatíveis para todo $i \in \{1, ..., p\}$ e para todo $j \in \{1, ..., q\}$. Uma regra-pct com condição vazia é temporalmente compatível com qualquer outra regra-pct (RIBEIRO et al., 2017a).

Quando duas regras-pct φ e φ' sobre o mesmo esquema relacional R são temporalmente compatíveis, significa que existe uma sequência $s \in \mathbf{Seq}(R)$ contendo uma posição i que satisfaz as condições de φ e φ' ($(s, i) \models C_\varphi$ $(s, i) \models C_{\varphi'}$). Cada conjunto de regras-pct temporalmente compatíveis Φ_φ pode ser mapeado para uma *teoria de preferências condicionais*, ou simplesmente *teoria-pc*. As teorias-pc são compostas por *regras de preferências condicionais*, ou *regras-pc*, que não possuem fórmulas de passado. Como existe um teste de consistência viável desenvolvido para teorias-pc, a ideia é reduzir o problema do teste de consistência de teorias-pct para um certo número de testes de consistências de teorias-pc.

O mapeamento de uma regra-pct para uma regra-pc consiste na remoção dos componentes temporais (fórmulas de passado). Dada uma regra-pct $\varphi : C_\varphi \rightarrow Q^+_\varphi \succ Q^-_\varphi[W_\varphi]$, a notação $\varphi^\bullet$ denota a regra-pc obtida a partir de φ, definida como $\varphi^\bullet : C^\bullet_\varphi \rightarrow Q^+_\varphi \succ Q^-_\varphi[W_\varphi]$. Já a teoria-pc resultante do mapeamento de um conjunto de regras-pct compatíveis Φ_φ é definida como $\Gamma(\Phi_\varphi) = \{\varphi'^\bullet \mid \varphi' \in \Phi_\varphi\}$. O Lema 3 e o Teorema 6 estabelecem as condições suficientes para verificar se uma teoria-pct é consistente.

Lema 3. Sejam φ, φ' duas regras-pct e s uma sequência. Se existe uma posição i em s tal que $(s, i) \models C_\varphi$ e $(s, i) \models C_{\varphi'}$, então φ e φ' são temporalmente compatíveis (RIBEIRO et al., 2017a).

Teorema 6. Seja Φ uma teoria-pct sobre um esquema relacional R. Se $\Gamma(\Phi_\varphi)$ é consistente para toda regra-pct $\varphi \in \Phi$, então Φ é consistente (RIBEIRO et al., 2017a).

Com base no Teorema 6, o algoritmo *IsConsistent* (Algoritmo 20) implementa o teste de consistência para uma teoria-pct. O algoritmo retorna **true** se a teoria-pct recebida é consistente ou **false** quando contrário. Na linha 2 é executado o teste de consistência para teorias-pc proposto em Pereira (2011). Este teste tem um custo de $O(m^3 + l^2)$, onde $m = |\Gamma(\Phi_\varphi)|$ é o número de regras-pc e l é o número de atributos que aparecem na teoria-pc $\Gamma(\Phi_\varphi)$. No pior caso, todas as regras-pct são temporalmente compatíveis e o número de regras-pc de cada teoria-pc será igual ao número total de regras-pct ($|\Gamma(\Phi_\varphi)| = |\Phi| = m$). Deste modo, a complexidade do algoritmo *IsConsistent* é de $O(m \times (m^3 + l^2)) = O(m^4 + ml^2)$, onde m é o número de regras-pct.

5.5 Análise comparativa dos modelos de preferência

A Tabela 2 apresenta uma análise comparativa entre os modelos de preferência da linguagem CPrefSQL, do formalismo TPref e da linguagem StreamPref. Como foi descrito

Algoritmo 20 – *IsConsistent*(Φ)

Entrada: Teoria-pct Φ
Saída: **true** quando Φ é consistente, caso contrário, **false**

```
1: for all φ ∈ Φ do                                  //Para cada regra-pct φ de Φ
2:     if Γ(Φ_φ) is not consistent then              //Teste de consistência para teorias-pc
3:         return false
4: return true
```

nas seções anteriores o modelo de preferência da CPrefSQL permite expressar preferências condicionais enquanto o formalismo TPref permite expressar preferências condicionais temporais. Já a linguagem StreamPref possui operadores capazes de processar consultas tanto com preferências condicionais quanto com preferências condicionais temporais.

Tabela 2 – Análise comparativa entre o modelo de preferência da StreamPref e trabalhos correlatos

		CPrefSQL	TPref	StreamPref
Preferências simples		✓	✓	✓
Preferências temporais			✓	✓
Teste de consistência viável		✓		✓
Predicados	igualdade	✓	✓	✓
	desigualdade	✓		✓
	intervalos			✓
Condições temporais com fórmulas arbitrárias			✓	
Atributos Indiferentes		✓		✓
Domínio temporal	presente	✓	✓	✓
	passado		✓	✓
	futuro		✓	

Quanto ao teste de consistência, as linguagens CPrefSQL e a StreamPref levam vantagem sobre o formalismo TPref. Os testes de consistência das linguagens CPrefSQL e StreamPref são viáveis para testar qualquer conjunto de preferências dentro das respectivas linguagens, já o formalismo TPref possui um teste de consistência viável apenas para um fragmento de sua linguagem.

No que diz respeito ao poder de expressão, os predicados da StreamPref são mais expressivos pois permitem expressar faixas de valores como $a < A < a'$. As fórmulas derivadas e, consequentemente, as condições temporais do formalismo TPref podem usar fórmulas arbitrárias, já na StreamPref as condições temporais são compostas por fórmulas mais simples. No entanto, pela propriedade da separação, é possível converter algumas teorias-pct do formalismo TPref para o modelo de preferência da linguagem StreamPref. Outra vantagem da StreamPref em relação ao TPref é a possibilidade de expressar preferências com atributos indiferentes. Deste modo é possível comparar posições de sequências que diferem em mais de um atributo, assim como ocorre na comparação de tuplas da CPrefSQL.

O modelo de preferência da StreamPref não possui o domínio temporal de futuro (fórmulas derivadas do operador temporal **until**). Todavia, no contexto de dados em fluxo este domínio temporal não é muito adequado, pois os usuários, em geral, expressam suas preferências com base nos dados dos instantes anteriores e do instante atual.

5.6 Considerações finais

Este capítulo apresentou o modelo de preferência utilizado pela linguagem StreamPref que teve como base o formalismo TPref. Este modelo de preferência permite expressar preferências condicionais temporais através de regras. O diferencial do modelo de preferência da linguagem StreamPref está em seu poder de expressão, já que, além da possibilidade de especificar atributos indiferentes, as regras são compostas por predicados capazes de representar faixas de valores.

Foi realizada uma comparação entre os modelos de preferência da linguagem CPrefSQL, do formalismo TPref e da linguagem StreamPref. Apesar de ser baseado no formalismo TPref, o modelo de preferência da linguagem StreamPref possui vantagens significativas como a definição de um teste de consistência viável.

O próximo capítulo apresentará os operadores da linguagem StreamPref. No caso dos operadores de preferência, o modelo de preferência apresentado no capítulo atual será utilizado para comparar sequências durante a execução de consultas contendo preferências temporais.

Capítulo 6

A linguagem StreamPref

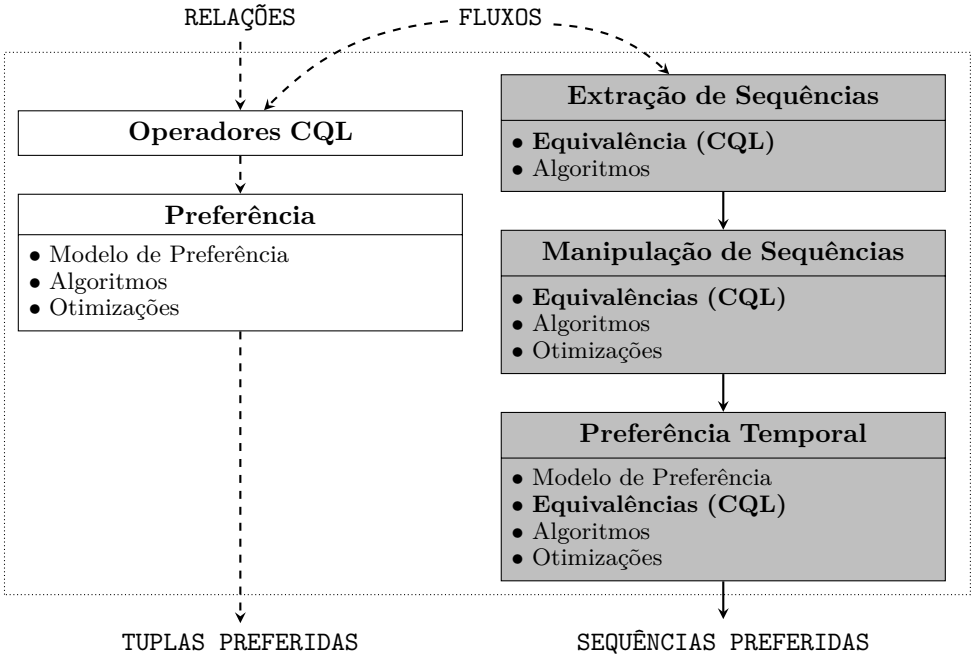


Figura 19 – Visão geral da linguagem StreamPref destacando os novos operadores

Como já foi explicado nos capítulos anteriores, a linguagem StreamPref é uma extensão da linguagem CQL. Conforme descrito no Capítulo 2, a linguagem CQL foi projetada para manipular tuplas provenientes de fluxos ou relações. Por outro lado, como foi mostrado no Capítulo 5, o modelo de preferência StreamPref é utilizado para comparar sequências de tuplas. Desta maneira, a linguagem StreamPref possui, além dos operadores CQL, novos operadores adequados para lidar com sequências. A Seção 6.6 mostra que existem equivalências para os operadores StreamPref usando apenas os operadores da CQL. Contudo, estas equivalências não são triviais e, como será mostrado nos capítulos posteriores, os operadores StreamPref realizam um processamento de consultas muito mais eficiente do que suas contrapartes em CQL. Além disto, a escrita de uma consulta com preferências usando as equivalências CQL envolve diversas operações complexas que

dificultam sua escrita pelo usuário.

A Figura 19 apresenta novamente uma visão geral da linguagem com os novos operadores destacados em cinza. O foco deste capítulo é a definição dos novos operadores e suas equivalências na linguagem CQL (RIBEIRO et al., 2017c). Contudo, é importante ressaltar que, além dos novos operadores descritos no capítulo atual, a linguagem StreamPref incorpora também os operadores **BEST** e **TOPK** explicados no Capítulo 3. A Figura 20 apresenta as categorias de operadores da linguagem StreamPref. As arestas tracejadas indicam as categorias de operadores já existentes na linguagem CQL. Os operadores **BEST** e **TOPK** se enquadram na categoria relação-para-relação. Observe que a StreamPref introduz duas novas categorias de operadores: fluxo-para-sequência e sequência-para-sequência. A categoria fluxo-para-sequência possui um operador que extrai sequências a partir de fluxos. Dentro da categoria sequência-para-sequência estão os os operadores de preferência e operadores para manipular sequências.

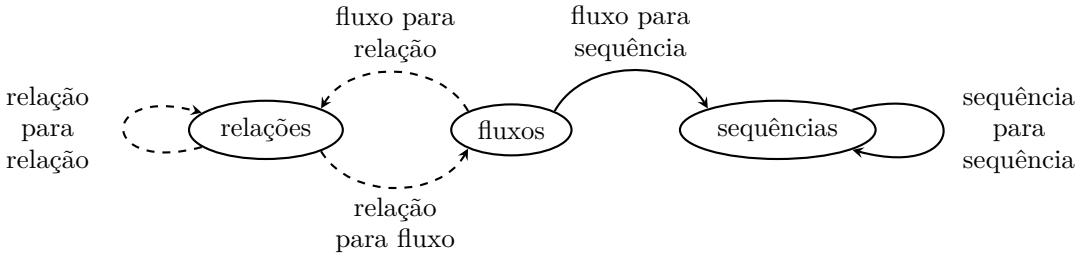


Figura 20 – Categorias de operadores da linguagem StreamPref

Este capítulo está organizado como se segue. A Seção 6.1 apresenta o operador para extração de sequências a partir de fluxos de dados. Os operadores de preferências são explicados na Seção 6.2. A Seção 6.3 descreve os operadores para obtenção de subsequências e os operadores para filtragem de sequências por comprimento. Já a Seção 6.4 discute sobre o plano de execução de consultas contendo preferências temporais. A Seção 6.5 apresenta a sintaxe de consultas na linguagem StreamPref. Em seguida, a Seção 6.6 aborda as equivalências CQL dos novos operadores. A Seção 6.7 faz uma análise comparativa entre a linguagem StreamPref e os trabalhos correlatos sobre processamento de consultas contínuas com preferências. Por fim, a Seção 6.8 apresenta as considerações finais sobre o capítulo.

6.1 Extração de sequências

Antes que as preferências temporais possam ser processadas em uma consulta StreamPref, é necessário converter os dados das relações e fluxos envolvidos para o formato de sequências. O operador **SEQ** extrai *sequências identificadas* (Definição 25) a partir de um fluxo de dados. Quando existem relações na consulta, as mesmas podem ser convertidas para fluxos usando os operadores relação-para-fluxo da CQL.

Definição 25 (Sequências identificadas)**.** Seja S um fluxo, sejam X e Y dois conjuntos disjuntos de atributos tais que $X \cup Y = \mathbf{Att}(S)$. Uma sequência identificada $s_x = \langle t_1, ..., t_n \rangle$ sobre S é uma sequência contendo um identificador $x \in \mathbf{Tup}(X)$ sendo que $t_i \in \mathbf{Tup}(Y)$ para todo $i \in \{1, ..., n\}$ (RIBEIRO et al., 2017c).

O operador **SEQ** possui três parâmetros: o conjunto de atributos identificadores das sequências (X), a abrangência temporal (n) e o intervalo de deslocamento (d). Os parâmetros de abrangência temporal (n) e intervalo de deslocamento (d) servem para delimitar uma porção de tuplas do fluxo de dados assim como ocorre com os operadores de janela deslizante baseadas em tempo explicadas no Capítulo 2. O conjunto de atributos identificadores X é usado para construir o identificador das sequências e posicionar as tuplas nas sequências corretas. É importante salientar que, dado o conjunto de tuplas de um instante, os atributos de X devem ser *atributos chaves* para este conjunto de tuplas, ou seja, em um mesmo instante não deve existir mais de uma tupla com os mesmos valores para os atributos de X. Como exemplo, considere um treinador que deseja analisar o comportamento de jogadores de seu time, o atributo identificador pode ser o número da camisa do jogador, assim cada sequência será identificada por um número único.

A definição formal do operador **SEQ** é dada pela Equação (1). Os termos $\texttt{START}_d$ e $\texttt{END}_{n,d}$ da Equação (1a) representam respectivamente os limites inferior e superior usados para selecionar as tuplas assim como ocorre nos operadores de janelas deslizantes explicados no Capítulo 2. A Equação (1b) seleciona apenas as tuplas com identificador igual a x. Em seguida, a Equação (1c) constrói sequências identificadas por x usando as tuplas da relação T_x. A Equação (1d) garante que as tuplas das sequências da relação S_x estão posicionadas corretamente de acordo com seus instantes. Finalmente, a Equação (1e) retorna todas as sequências identificadas provenientes da operação de concatenação. O Exemplo 20 demonstra como o operador **SEQ** pode ser utilizado para extrair as sequências de um fluxo de dados.

$$T = \{t \in S \mid \texttt{START}_d(t) \leq \tau_{now} \leq \texttt{END}_{n,d}(t)\} \tag{1a}$$

$$T_x = \{t/X \mid t \in T \textbf{ and } \pi_X(t_i) = x\} \tag{1b}$$

$$S'_x = \{\langle t_1, ..., t_n \rangle \mid t_1, ..., t_n \in T_x\} \tag{1c}$$

$$S_x = \{s_x \mid s_x \in S'_x \textbf{ and } \texttt{TS}(t_1) < \texttt{TS}(t_2) \textbf{ and } ... \textbf{ and } \texttt{TS}(t_{n-1}) < \texttt{TS}(t_n)\} \tag{1d}$$

$$\mathbf{SEQ}_{X,n,d}(S) = \bigcup_{x \,\in\, \mathbf{Tup}(X)} S_x \tag{1e}$$

Exemplo 20 (Operador **SEQ**)**.** Considere o fluxo `posicionamento(jid, local, bola, direcao)` mostrado na Figura 21(a). Considere também que um treinador deseje monitorar o posicionamento dos jogadores a cada dois segundos (intervalo de deslocamento) considerando os últimos quatro segundos (abrangência temporal). Usando o atributo `jid` como identificador, o monitoramento desejado pelo treinador pode ser realizado pela operação $\mathbf{SEQ}_{\{\texttt{jid}\},4,2}(\texttt{posicionamento})$.

TS()	jid	local	bola	direcao
0	1	id	1	*lateral*
0	2	mc	1	*lateral*
0	3	id	1	*frente*
0	4	mc	1	*lateral*
0	5	mc	1	*lateral*
1	1	mc	1	*lateral*
1	2	mc	1	*lateral*
1	3	id	1	*frente*
1	4	mc	1	*lateral*
1	5	mc	1	*lateral*
2	1	mc	1	*lateral*
2	2	io	1	*lateral*
2	3	mc	1	*frente*
2	4	io	1	*lateral*
2	5	io	1	*lateral*
3	1	io	0	*lateral*
3	2	io	0	*lateral*
3	3	mc	0	*lateral*
3	4	mc	0	*lateral*
3	5	io	0	*frente*
4	1	mc	1	*lateral*
4	2	io	1	*tras*
4	3	id	1	*lateral*
4	4	mc	1	*tras*
4	5	io	1	*tras*
5	1	io	0	*tras*
5	2	io	0	*tras*
5	3	mc	0	*lateral*
5	4	io	0	*tras*
5	5	mc	0	*tras*

(a) Fluxo `posicionamento`

Instante 0: $\mathbf{SEQ}_{\{\texttt{jid}\},4,2}(\texttt{posicionamento})$

$s_1 = \langle(id, 1, lateral)\rangle$
$s_2 = \langle(mc, 1, lateral)\rangle$
$s_3 = \langle(id, 1, frente)\rangle$
$s_4 = \langle(mc, 1, lateral)\rangle$
$s_5 = \langle(mc, 1, lateral)\rangle$

Instante 1: $\mathbf{SEQ}_{\{\texttt{jid}\},4,2}(\texttt{posicionamento})$

$s_1 = \langle(id, 1, lateral), (mc, 1, lateral)\rangle$
$s_2 = \langle(mc, 1, lateral), (mc, 1, lateral)\rangle$
$s_3 = \langle(id, 1, frente), (id, 1, frente)\rangle$
$s_4 = \langle(mc, 1, lateral), (mc, 1, lateral)\rangle$
$s_5 = \langle(mc, 1, lateral), (mc, 1, lateral)\rangle$

Instante 2: $\mathbf{SEQ}_{\{\texttt{jid}\},4,2}(\texttt{posicionamento})$

$s_1 = \langle(id, 1, lateral), (mc, 1, lateral), (mc, 1, lateral)\rangle$
$s_2 = \langle(mc, 1, lateral), (mc, 1, lateral), (io, 1, lateral)\rangle$
$s_3 = \langle(id, 1, frente), (id, 1, frente), (mc, 1, frente)\rangle$
$s_4 = \langle(mc, 1, lateral), (mc, 1, lateral), (io, 1, lateral)\rangle$
$s_5 = \langle(mc, 1, lateral), (mc, 1, lateral), (io, 1, lateral)\rangle$

Instante 3: $\mathbf{SEQ}_{\{\texttt{jid}\},4,2}(\texttt{posicionamento})$

$s_1 = \langle(id, 1, lateral), (mc, 1, lateral), (mc, 1, lateral), (io, 0, lateral)\rangle$
$s_2 = \langle(mc, 1, lateral), (mc, 1, lateral), (io, 1, lateral), (io, 0, lateral)\rangle$
$s_3 = \langle(id, 1, frente), (id, 1, frente), (mc, 1, frente), (mc, 0, lateral)\rangle$
$s_4 = \langle(mc, 1, lateral), (mc, 1, lateral), (io, 1, lateral), (mc, 0, lateral)\rangle$
$s_5 = \langle(mc, 1, lateral), (mc, 1, lateral), (io, 1, lateral), (io, 0, frente)\rangle$

Instante 4: $\mathbf{SEQ}_{\{\texttt{jid}\},4,2}(\texttt{posicionamento})$

$s_1 = \langle(mc, 1, lateral), (io, 0, lateral), (mc, 1, lateral)\rangle$
$s_2 = \langle(io, 1, lateral), (io, 0, lateral), (io, 1, tras)\rangle$
$s_3 = \langle(mc, 1, frente), (mc, 0, lateral), (id, 1, lateral)\rangle$
$s_4 = \langle(io, 1, lateral), (mc, 0, lateral), (mc, 1, tras)\rangle$
$s_5 = \langle(io, 1, lateral), (io, 0, frente), (io, 1, tras)\rangle$

Instante 5: $\mathbf{SEQ}_{\{\texttt{jid}\},4,2}(\texttt{posicionamento})$

$s_1 = \langle(mc, 1, lateral), (io, 0, lateral), (mc, 1, lateral), (io, 0, tras)\rangle$
$s_2 = \langle(io, 1, lateral), (io, 0, lateral), (io, 1, tras), (io, 0, tras)\rangle$
$s_3 = \langle(mc, 1, frente), (mc, 0, lateral), (id, 1, lateral), (mc, 0, lateral)\rangle$
$s_4 = \langle(io, 1, lateral), (mc, 0, lateral), (mc, 1, tras), (io, 0, tras)\rangle$
$s_5 = \langle(io, 1, lateral), (io, 0, frente), (io, 1, tras), (mc, 0, tras)\rangle$

(b) Execução do operador **SEQ**

Figura 21 – Fluxo `posicionamento` e execução do operador **SEQ**

O resultado da extração de sequências é apresentado, instante por instante, na Figura 21(b). A extração de sequências inicia no instante 0, as tuplas deste instante possuem $\texttt{START}_d = 0$ e $\texttt{END}_{n,d} = 3$. No instante 1, os limites permanecem os mesmos e as tuplas são acrescentadas às sequências correspondentes. As tuplas dos instantes 2 e 3 possuem $\texttt{START}_d = 2$ e $\texttt{END}_{n,d} = 5$ e também são acrescentadas às sequências. No instante 4, as tuplas dos instantes 0 e 1 expiram e são removidas (o $\texttt{END}_{n,d}$ destas tuplas é menor do que o instante atual). As tuplas dos instantes 4 e 5 com $\texttt{START}_d = 4$ e $\texttt{END}_{n,d} = 7$ são apenas acrescentadas às sequências. O Código 2 mostra como a operação $\mathbf{SEQ}_{\{\texttt{jid}\},4,2}(\texttt{posicionamento})$ é escrita usando a linguagem StreamPref. A sintaxe completa da linguagem StreamPref é descrita na Seção 6.5.

Código 2 – Consulta para extração de sequências na linguagem StreamPref

```
SELECT SEQUENCE IDENTIFIED BY jid [RANGE 4 SECOND, SLIDE 2 SECOND]
FROM posicionamento;
```

6.2 Operadores de preferência temporal

A linguagem StreamPref possui dois operadores de preferência temporal, são eles **BESTSEQ** e **TOPKSEQ**. O operador **BESTSEQ** é apresentado na Seção 6.2.1 e o operador **TOPKSEQ** é explicado na Seção 6.2.2.

6.2.1 O operador BESTSEQ

O operador **BESTSEQ** recebe um conjunto de sequências Z e, utilizando o modelo de preferência temporal explicado no Capítulo 5, retorna as sequências dominantes em Z de acordo com uma teoria-pct Φ, isto é, $\mathbf{BESTSEQ}_{\Phi}(Z) = \{s \in Z \mid \nexists s' \in Z \textbf{ and } s' \succ_{\Phi} s\}$. O Exemplo 21 mostra uma aplicação prática do operador **BESTSEQ**.

Exemplo 21 (Operador **BESTSEQ**)**.** Considere novamente a teoria-pct $\Phi = \{\varphi_1, \varphi_2, \varphi_3\}$ do Exemplo 17 onde:

φ_1 : $(\texttt{bola} = 1) \wedge \mathbf{Prev}(\texttt{local} = id) \rightarrow (\texttt{local} = mc) \succ (\texttt{local} = id)[\texttt{direcao}]$;
φ_2 : $(\texttt{bola} = 0) \wedge \mathbf{Prev}(\texttt{local} = io) \wedge \mathbf{AllPrev}(\texttt{bola} = 1) \rightarrow (\texttt{local} = mc) \succ (\texttt{local} = io)$;
φ_3 : $(\texttt{direcao} = \textit{lateral}) \succ (\texttt{direcao} = \textit{frente})$.

Considere também que o treinador deseje realizar a seguinte consulta sobre o fluxo `posicionamento` da Figura 21(a):

[Q1] "A cada dois segundos, me informe os jogadores que melhor atendem às minhas preferências sobre posicionamento levando em consideração os quatro últimos instantes".

O Código 3 mostra como a consulta **Q1** é escrita na linguagem StreamPref. Esta consulta pode ser resolvida usando a operação $\mathbf{BESTSEQ}_{\Phi}(\mathbf{SEQ}_{\{\texttt{jid}\},4,2}(\texttt{posicionamento}))$. O processamento desta operação é realizado da seguinte maneira:

- No instante 0, é possível comparar as sequências s_1 e s_3, sendo que $s_1 \succ_{\varphi_3} s_3$. Portanto, o resultado para este instante é $\{s_1, s_2, s_4, s_5\}$;
- No instante 1, o resultado é $\{s_1, s_2, s_4, s_5\}$ ($s_1 \succ_{\varphi_1} s_3$);
- O instante 2 possui o mesmo resultado do instante 1;
- No instante 3, as sequências s_2, s_3 e s_5 são dominadas ($s_2 \succ_{\varphi_3} s_5$, $s_4 \succ_{\varphi_2} s_2$ e $s_1 \succ_{\varphi_3} s_3$). Assim o resultado deste instante é $\{s_1, s_4\}$;
- No instante 4, o início das sequências é alterado. Porém, as mesmas comparações são possíveis.
- O instante 5 possui o mesmo resultado dos instantes 3 e 4.

Código 3 – Consulta **Q1** na linguagem StreamPref

```
SELECT SEQUENCE IDENTIFIED BY jid [RANGE 4 SECOND, SLIDE 2 SECOND]
FROM posicionamento
ACCORDING TO TEMPORAL PREFERENCES
  IF (bola = 1) AND PREVIOUS (local = 'id')
    THEN (local = 'mc') BETTER (local = 'id')[direcao] AND
  IF (bola = 0) AND PREVIOUS (local = 'io') AND ALL PREVIOUS (bola = 1)
    THEN (local = 'mc') BETTER (local = 'io') AND
  (direcao = 'lateral') BETTER (direcao = 'frente');
```

6.2.2 O operador TOPKSEQ

O operador **TOPKSEQ** recebe como entrada um conjunto de sequências Z e retorna as *top-k sequências* de Z de acordo com uma teoria-pct Φ. As top-k sequências de Z são as k sequências com menor nível de preferência temporal (Definição 26) segundo Φ. O nível de preferência temporal é análogo ao nível de preferência imposto pelas teorias-pc sobre tuplas.

Definição 26 (Nível de preferência temporal)**.** Seja Φ uma teoria-pct. Seja Z um conjunto de sequências e seja $s \in Z$ uma sequência. O nível de preferência temporal de s de acordo com Φ, denotado por $level(s)$, é definido de forma indutiva como se segue:

1) Se $\nexists s' \in Z$ tal que $s' \succ_\Phi s$, então $level(s) = 0$;
2) Caso contrário, $level(s) = \max\{level(s') \mid s' \in Z \textbf{ and } s' \succ_\Phi s\} + 1$.

O operador **TOPKSEQ** é útil em situações onde o operador **BESTSEQ** retorna poucas sequências. Nestes casos, o **TOPKSEQ** pode ser usado para complementar a resposta usando sequências com níveis maiores que zero como é mostrado no Exemplo 22.

Exemplo 22 (Operador **TOPKSEQ**)**.** Considere novamente a teoria-pct do Exemplo 21. Suponha agora que o treinador deseje realizar a seguinte consulta:

[Q2] "A cada dois segundos, me informe os quatro jogadores que melhor atendem às minhas preferências sobre posicionamento levando em consideração os quatro últimos instantes".

O Código 4 mostra como a consulta **Q2** é escrita na linguagem StreamPref. Esta consulta pode ser resolvida usando a operação $\textbf{TOPKSEQ}_{\Phi,4}(\textbf{SEQ}_{\{\texttt{jid}\},4,2}(\texttt{posicionamento}))$. O processamento desta operação é realizado da seguinte maneira:

- No instante 0, apenas a sequência s_3 é dominada. Portanto, o resultado para este instante é $\{s_1, s_2, s_4, s_5\}$;
- No instante 1, o resultado permanece o mesmo $\{s_1, s_2, s_4, s_5\}$;
- O instante 2 possui o mesmo resultado do instante 1;
- No instante 3, as sequências s_2, s_3 e s_5 são dominadas ($s_2 \succ_{\varphi_3} s_5$, $s_4 \succ_{\varphi_2} s_2$, $s_1 \succ_{\varphi_3} s_3$ e $s_4 \succ_{\varphi_2} ... \succ_{\varphi_3} s_5$). A Figura 22 mostra o BTG das sequências neste instante. Como

o usuário deseja as 4 melhores sequências, o resultado é complementado com as sequências s_2 e s_3 de nível 1. Assim o resultado deste instante é $\{s_1, s_2, s_3, s_4\}$;

- No instante 4, o início das sequências é alterado. Porém, as mesmas comparações são possíveis.
- O instante 5 possui o mesmo resultado dos instantes 3 e 4.

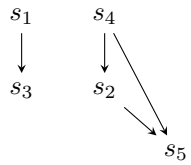


Figura 22 – BTG das sequências extraídas do fluxo `posicionamento` no instante 3

Código 4 – Consulta **Q2** na linguagem StreamPref

```
SELECT TOP(4) SEQUENCE IDENTIFIED BY jid [RANGE 4 SECOND, SLIDE 2 SECOND]
FROM posicionamento
ACCORDING TO TEMPORAL PREFERENCES
  IF (bola = 1) AND PREVIOUS (local = 'id')
    THEN (local = 'mc') BETTER (local = 'id')[direcao] AND
  IF (bola = 0) AND PREVIOUS (local = 'io') AND ALL PREVIOUS (bola = 1)
    THEN (local = 'mc') BETTER (local = 'io') AND
  (direcao = 'lateral') BETTER (direcao = 'frente');
```

O resultado do operador **TOPKSEQ** pode ser obtido em função do operador **BESTSEQ** conforme mostrado na Equação (2). O termo m é o nível de preferência temporal máximo imposto por Φ.

$$L_0 \leftarrow \pi'_{\mathtt{X}}(\mathbf{BESTSEQ}_{\Phi}(Z)) \tag{2a}$$

$$L_1 \leftarrow \pi'_{\mathtt{X}}(\mathbf{BESTSEQ}_{\Phi}(Z - L_0)) \tag{2b}$$

$$L_2 \leftarrow \pi'_{\mathtt{X}}(\mathbf{BESTSEQ}_{\Phi}(Z - L_1 - L_0)) \tag{2c}$$

$$\vdots$$

$$L_m \leftarrow \pi'_{\mathtt{X}}(\mathbf{BESTSEQ}_{\Phi}(Z - L_{m-1} - ... - L_0)) \tag{2d}$$

$$L \leftarrow \cup_{0 \leq i \leq m} (L_i \times \rho_{(\mathtt{level})}(\{i\})) \tag{2e}$$

No pior caso, o maior nível será igual ao número de regras de Φ. Cada relação L_i possui os identificadores das sequências de Z com nível igual a i. O símbolo π' representa o operador de projeção com eliminação de duplicatas, necessário para garantir que um identificador de sequência seja considerado uma única vez. A relação L possui todos os identificadores de sequências associados a seus respectivos níveis.

O conjunto de identificadores das top-k sequências pode então ser obtido através das operações de ordenação (`ORDER BY`) e contagem (`LIMIT` k). Posteriormente, a seleção das top-k sequências é feita usando este conjunto de identificadores.

6.3 Obtenção de subsequências e filtragem por comprimento

Em determinadas situações, os operadores **SEQ**, **BESTSEQ** e **TOPKSEQ** podem não ser suficientes para retornar o resultado desejado pelo usuário. Para lidar com estes casos, a linguagem StreamPref também possui operadores de obtenção de subsequências, descritos nas Seções 6.3.1 e 6.3.2, e operadores de filtragem por comprimento, explicados na Seção 6.3.3.

6.3.1 O operador CONSEQ

O Exemplo 21 mostrou que é possível resolver a consulta **Q1** do treinador utilizando os operadores **SEQ** e **BESTSEQ**. Todavia, estes operadores não são suficientes quando as consultas do usuário precisam extrair subsequências maximais contendo apenas posições com instantes consecutivos, chamadas de *subsequências-tc*. O Exemplo 23 apresenta uma consulta onde esta situação ocorre.

Exemplo 23 (Consulta por subsequências-tc)**.** Considere o fluxo `evento(jid, jogada, local)` com informações sobre as jogadas realizadas pelos jogadores com posse de bola. A descrição dos atributos do fluxo é a seguinte:

`jid`: identificação do jogador;
`jogada`: jogada realizada pelo jogador;
`local`: divisão do campo usada nos exemplos anteriores.

As possíveis jogadas realizadas pelos jogadores são: condução (*cond*), passe completo (*pass*), passe incompleto (*bpas*), drible (*drib*), perda de bola (*lbal*) e recepção (*rec*). Considere o fluxo `evento` mostrado na Figura 23(a). Considere também as seguintes preferências de um treinador a respeito das jogadas feitas pelos jogadores:

[P4] Para a primeira jogada, uma recepção é melhor do que perda de bola;
[P5] Se a jogada anterior foi uma condução, então prefiro um drible a um passe, independente do local;
[P6] Passes completos são melhores do que passes incompletos, independente do local;
[P7] Se todas as jogadas anteriores foram na intermediária ofensiva, então prefiro uma jogada neste mesmo local a uma jogada no meio de campo.

As preferências do treinador são representadas pela teoria-pct $\Phi' = \{\varphi_4, \varphi_5, \varphi_6, \varphi_7\}$, onde:

φ_4 : **First** $\rightarrow$ (`jogada` $= rec$) $\succ$ (`jogada` $= lbal$);
φ_5 : **Prev**(`jogada` $= cond$) $\rightarrow$ (`jogada` $= drib$) $\succ$ (`jogada` $= pass$)[`local`];
φ_6 : $\rightarrow$ (`jogada` $= pass$) $\succ$ (`jogada` $= bpas$)[`local`];
φ_7 : **AllPrev**(`local` $= io$) $\rightarrow$ (`local` $= io$) $\succ$ (`local` $= mc$).

Suponha que o treinador deseje uma resposta para a seguinte consulta:

[Q3] "A cada instante, me informe as jogadas consecutivas de cada jogador que melhor atendam às minhas preferências, considerando os últimos 20 segundos".

Utilizando apenas os operadores **SEQ** e **BESTSEQ**, a consulta **Q3** é escrita na linguagem StreamPref como mostrado no Código 5.

Código 5 – Consulta **Q3** na linguagem StreamPref

```
SELECT SEQUENCE IDENTIFIED BY jid [RANGE 20 SECOND]
FROM evento
ACCORDING TO TEMPORAL PREFERENCES
  IF FIRST THEN jogada = 'rec' BETTER jogada = 'lbal' AND
  IF PREVIOUS (jogada = 'cond')
    THEN (jogada = 'drib') BETTER (jogada = 'pass')[local] AND
  (jogada = 'pass') BETTER (jogada = 'bpas')[local] AND
  IF ALL PREVIOUS (local = 'io')
    THEN (local = 'io') BETTER (local = 'mc');
```

O primeiro passo para resolver a consulta é extrair as sequências representando as jogadas consecutivas do fluxo `evento`. Entretanto, somente o operador **SEQ** não é suficiente para realizar esta tarefa. Observe o resultado da operação $\mathbf{SEQ}_{\{\texttt{jid}\},20,1}(\texttt{evento})$ apresentado na Figura 23(b). Note que, na sequência s_1, as posições $s_1[2]$ e $s_1[3]$ não possuem instantes consecutivos, $\texttt{TS}(s_1[2]) = 2$ e $\texttt{TS}(s_1[3]) = 12$. Como a abrangência temporal da consulta é de 20 segundos, pode acontecer de um jogador ter a posse de bola, passar a bola ou perdê-la e, mais adiante, ter a posse de bola novamente. Utilizando apenas o operador **SEQ**, cada sequência será composta por todas as jogadas de cada jogador nos últimos 20 segundos, podendo existir jogadas não consecutivas nestas sequências.

TS()	jid	local	jogada
1	1	mc	pass
2	1	mc	lbal
6	3	io	rec
7	3	io	cond
8	3	io	bpas
12	1	mc	rec
13	1	io	cond
14	1	io	drib
15	2	mc	lbal
16	2	io	cond
17	2	io	pass
18	4	io	rec
19	4	mc	cond
20	4	ao	cond

(a) Fluxo `evento`

$s_1 = \langle \underset{1}{(mc, pass)}, \underset{2}{(mc, lbal)}, \underset{12}{(mc, rec)}, \underset{13}{(io, cond)}, \underset{14}{(io, drib)} \rangle$

$s_2 = \langle \underset{15}{(mc, lbal)}, \underset{16}{(io, cond)}, \underset{17}{(io, pass)} \rangle$

$s_3 = \langle \underset{6}{(io, rec)}, \underset{7}{(io, cond)}, \underset{8}{(io, bpas)} \rangle$

$s_4 = \langle \underset{18}{(io, rec)}, \underset{19}{(mc, cond)}, \underset{20}{(ao, cond)} \rangle$

(b) Resultado da operação $\mathbf{SEQ}_{\{\texttt{jid}\},20,1}(\texttt{evento})$ no instante 20

Figura 23 – Fluxo `evento` e resultado da operação $\mathbf{SEQ}_{\{\texttt{jid}\},20,1}(\texttt{evento})$

No Exemplo 23 foi mostrado que o operador **SEQ** não é suficiente para obter as subsequências-tc necessárias para resolver a consulta **Q3**. Para lidar com este tipo de situação, foi definido o operador **CONSEQ**. Dado um conjunto de sequências Z, para cada sequência $s \in Z$, o operador **CONSEQ** extrai as subsequências-tc de s.

A definição do operador **CONSEQ** é dada pela Equação (3). A função $start(s, i)$, Equação (3a), verifica se a posição i de s é o início de uma subsequência-tc. Já a função $end(s, j)$, Equação (3b), checa se a posição $s[j]$ é o fim de uma subsequência-tc. A função $isconseq(s)$ da Equação (3c) é usada para garantir que uma subsequência s contenha apenas posições consecutivas. Na Equação(3d), para cada sequência do conjunto Z, são obtidas todas as subsequências que começam e terminam em uma posição não consecutiva. Por fim, a Equação (3e) obtém as subsequências-tc usando a função *isconseq*. O Exemplo 24 demonstra como a consulta **Q3** pode ser resolvida usando o operador **CONSEQ**.

$$start(s,i) = \begin{cases} \textbf{true}, \textbf{if } i = 1 \textbf{ or } \texttt{TS}(s[i]) \neq \texttt{TS}(s[i-1]) + 1 \\ \textbf{false}, \textbf{else} \end{cases} \tag{3a}$$

$$end(s,j) = \begin{cases} \textbf{true}, \textbf{if } j = |s| \textbf{ or } \texttt{TS}(s[j]) \neq \texttt{TS}(s[j+1]) - 1 \\ \textbf{false}, \textbf{else} \end{cases} \tag{3b}$$

$$isconseq(s) = \begin{cases} \textbf{true}, \textbf{if } \nexists k \text{ tal que } 1 < k < |s| \textbf{ and } start(s,k) \\ \textbf{false}, \textbf{else} \end{cases} \tag{3c}$$

$$Z' = \{s[i,j] \mid s \in Z \textbf{ and } 1 \leq i \leq |s| \textbf{ and } i \leq j \leq |s| \textbf{ and } start(s,i) \textbf{ and } end(s,j)\} \tag{3d}$$

$$\textbf{CONSEQ}(Z) = \{s \in Z' \mid isconseq(s)\} \tag{3e}$$

Exemplo 24 (Operador **CONSEQ**)**.** Considere novamente a consulta **Q3** do Exemplo 23. O processamento desta consulta utilizando o operador **CONSEQ** é realizado pela operação $\textbf{BESTSEQ}_{\Phi'}(\textbf{CONSEQ}(\textbf{SEQ}_{\{\texttt{jid}\},20,1}(\texttt{evento})))$. A consulta StreamPref referente a esta operação é mostrada no código 6. As sequências extraídas pelo operador **SEQ** são quebradas em subsequências-tc pelo operador **CONSEQ**. A Figura 24 mostra estas subsequências. Em seguida, o operador **BESTSEQ** compara as subsequências-tc. Agora é possível fazer as comparações $s_{1_b} \succ_{\varphi_4} s_2$ e $s_3 \succ_{\varphi_7} s_4$ fazendo com que o resultado da consulta no instante 20 seja $\{s_{1_a}, s_{1_b}, s_3\}$.

$$\begin{array}{l} s_{1_a} = \langle (mc, pass)_{1}, (mc, lbal)_{2} \rangle \\ s_{1_b} = \langle (mc, rec)_{12}, (io, cond)_{13}, (io, drib)_{14} \rangle \\ s_2 = \langle (mc, lbal)_{15}, (io, cond)_{16}, (io, pass)_{17} \rangle \\ s_3 = \langle (io, rec)_{6}, (io, cond)_{7}, (io, bpas)_{8} \rangle \\ s_4 = \langle (io, rec)_{18}, (mc, cond)_{19}, (ao, cond)_{20} \rangle \end{array}$$

Figura 24 – Resultado da operação $\textbf{CONSEQ}(\textbf{SEQ}_{\{\texttt{jid}\},20,1}(\texttt{evento}))$ no instante 20

Código 6 – Consulta **Q3** usando o operador **CONSEQ**

```
SELECT SUBSEQUENCE CONSECUTIVE TUPLES FROM
SEQUENCE IDENTIFIED BY jid [RANGE 20 SECOND]
FROM evento
ACCORDING TO TEMPORAL PREFERENCES
  IF FIRST THEN jogada = 'rec' BETTER jogada = 'lbal' AND
  IF PREVIOUS (jogada = 'cond')
    THEN (jogada = 'drib') BETTER (jogada = 'pass')[local] AND
  (jogada = 'pass') BETTER (jogada = 'bpas')[local] AND
  IF ALL PREVIOUS (local = 'io')
    THEN (local = 'io') BETTER (local = 'mc');
```

6.3.2 O operador ENDSEQ

Considere novamente o Exemplo 24, observe que apenas duas comparações são possíveis, uma vez que o operador **BESTSEQ** deve comparar duas sequências na primeira posição não correspondente entre elas. Em aplicações práticas, consultas-pct contínuas com um baixo número de comparações podem retornar uma quantidade muito grande de sequências, fazendo com que a resposta da consulta não tenha grande utilidade para o usuário. Considere agora as sequências $s_{1_b} = \langle (mc, rec), \textbf{(io, cond), (io, drib)} \rangle$ e $s_3 = \langle (io, rec), \textbf{(io, cond), (io, bpas)} \rangle$ presentes no resultado da consulta do Exemplo 24. Estas sequências não podem ser comparadas, mas suas duas últimas posições (em negrito) podem ser comparadas de forma transitiva pelas regras φ_4 e φ_5.

Uma alternativa para evitar respostas com muitas sequências é obter subsequências menores que tenham mais chances de serem comparadas. O operador **ENDSEQ** foi criado para realizar este tipo de operação. Dado um conjunto de sequências Z, para cada sequência $s \in Z$, o operador **ENDSEQ** obtém todas as subsequências de s contendo sua última posição. Estas subsequências são chamadas de *subsequências-up*. A definição do operador **ENDSEQ** é dada pela Equação (4).

$$\textbf{ENDSEQ}(Z) = \{s[i,j] \mid s \in Z \textbf{ and } j = |s| \textbf{ and } 1 \leq i \leq j\} \tag{4}$$

Para cada sequência s, o operador **ENDSEQ** obtém n subsequências tal que $n = |s|$. As subsequências de s têm a mesma posição final, mas todas iniciam com posições distintas o que aumenta as possibilidades de comparação. Isso acontece porque haverão mais sequências com início diferente e, portanto, mais chances de comparação. O Exemplo 25 mostra como processar a consulta **Q3** usando o operador **ENDSEQ**.

Exemplo 25 (Operador **ENDSEQ**)**.** Considere novamente a consulta **Q3**. Com a utilização do operador **ENDSEQ**, esta consulta pode ser resolvida pela operação $\textbf{BESTSEQ}_{\Phi'}(\textbf{ENDSEQ}(\textbf{CONSEQ}(\textbf{SEQ}_{\{\texttt{jid}\},20,1}(\texttt{evento}))))$. O Código 7 mostra como esta operação é escrita na linguagem StreamPref. O processamento da consulta inicia com a extração de sequências feita pelo operador **SEQ**. Em seguida, o operador **CONSEQ** obtém as subsequências-tc. O operador **ENDSEQ** obtém as subsequências-up sobre o

resultado do operador **CONSEQ**. A Figura 25 exibe as subsequências retornadas pelo operador **ENDSEQ**.

Código 7 – Consulta **Q3** usando o operador **ENDSEQ**

```
SELECT SUBSEQUENCE END POSITION
SUBSEQUENCE CONSECUTIVE TUPLES FROM
SEQUENCE IDENTIFIED BY jid [RANGE 20 SECOND]
FROM evento
ACCORDING TO TEMPORAL PREFERENCES
  IF FIRST THEN jogada = 'rec' BETTER jogada = 'lbal' AND
  IF PREVIOUS (jogada = 'cond')
    THEN (jogada = 'drib') BETTER (jogada = 'pass')[local] AND
  (jogada = 'pass') BETTER (jogada = 'bpas')[local] AND
  IF ALL PREVIOUS (local = 'io')
    THEN (local = 'io') BETTER (local = 'mc');
```

O último passo da consulta é o processamento do operador **BESTSEQ**. As sequências s''_{1_a}, s'_{1_b}, s''_4 e s'_4 são incomparáveis e, portanto, dominantes. Além disto, as seguintes comparações são realizadas: $s'''_{1_b} \succ_{\Phi'} s'_{1_a}$, $s'''_{1_b} \succ_{\Phi'} s'''_2$, $s''_{1_b} \succ_{\Phi'} s''_2$, $s''_{1_b} \succ_{\Phi'} s''_3$, $s'_2 \succ_{\Phi'} s'_3$, $s''_2 \succ_{\Phi'} s''_3$ e $s'''_3 \succ_{\Phi'} s'''_4$. Portanto, o resultado da consulta é composto pelas sequências s''_{1_a}, s'_{1_b}, s''_{1_b}, s'''_{1_b}, s'_2, s'''_3, s'_4 e s''_4 (em negrito na Figura 25).

$s'_{1_a} = (mc, lbal)\rangle$
$\mathbf{s''_{1_a}} = (\mathbf{mc}, \mathbf{pass}), (\mathbf{mc}, \mathbf{lbal})\rangle$
$\mathbf{s'_{1_b}} = (\mathbf{io}, \mathbf{drib})\rangle$
$\mathbf{s''_{1_b}} = (\mathbf{io}, \mathbf{cond}), (\mathbf{io}, \mathbf{drib})\rangle$
$\mathbf{s'''_{1_b}} = (\mathbf{mc}, \mathbf{rec}), (\mathbf{io}, \mathbf{cond}), (\mathbf{io}, \mathbf{drib})\rangle$
$\mathbf{s'_2} = (\mathbf{io}, \mathbf{pass})\rangle$
$s''_2 = (io, cond), (io, pass)\rangle$
$s'''_2 = (mc, lbal), (io, cond), (io, pass)\rangle$
$s'_3 = (io, bpas)\rangle$
$s''_3 = (io, cond), (io, bpas)\rangle$
$\mathbf{s'''_3} = (\mathbf{io}, \mathbf{rec}), (\mathbf{io}, \mathbf{cond}), (\mathbf{io}, \mathbf{bpas})\rangle$
$\mathbf{s'_4} = (\mathbf{ao}, \mathbf{cond})\rangle$
$\mathbf{s''_4} = (\mathbf{mc}, \mathbf{cond}), (\mathbf{ao}, \mathbf{cond})\rangle$
$s'''_4 = (io, rec), (mc, cond), (ao, cond)\rangle$

Figura 25 – Resultado da consulta **Q3** usando o operador **ENDSEQ** no instante 20

6.3.3 O operadores MINSEQ e MAXSEQ

O resultado da consulta **Q3** usando o operador **ENDSEQ** possui sequências de comprimento um a quatro. O treinador poderia refinar mais ainda este resultado dizendo que deseja considerar apenas um certo número de jogadas consecutivas. A linguagem StreamPref possui os operadores **MINSEQ** e **MAXSEQ** que permitem filtrar as sequências pelo comprimento. O operador **MINSEQ**, definido pela Equação (5a), descarta as sequências com comprimento menor do que um parâmetro α. Já o operador

MAXSEQ, definido pela Equação (5b), elimina as sequências com comprimento maior que um limite β. O Exemplo 26 demonstra a utilização do operador **MINSEQ** para atender às restrições do treinador.

$$\mathbf{MINSEQ}_{\alpha}(Z) = \{s \in Z \mid \alpha \leq |s|\} \tag{5a}$$

$$\mathbf{MAXSEQ}_{\beta}(Z) = \{s \in Z \mid \beta \geq |s|\} \tag{5b}$$

Exemplo 26 (Operador **MINSEQ**)**.** Considere o resultado do operador **ENDSEQ** mostrado na Figura 25. Suponha que o treinador deseje apenas as melhores sequências de jogadas consecutivas com pelo menos 2 jogadas. Tais sequências podem ser obtidas por meio da consulta apresentada no Código 8 que, por sua vez, corresponde à operação $\mathbf{BESTSEQ}_{\Phi'}(\mathbf{MINSEQ}_3(\mathbf{ENDSEQ}(\mathbf{CONSEQ}(\mathbf{SEQ}_{\{\texttt{jid}\},20,1}(\texttt{evento})))))$. As sequências retornadas após o processamento do operador **MINSEQ** são exibidas na Figura 26. O operador **BESTSEQ** realiza as seguintes comparações: $s'''_{1_b} \succ_{\Phi'} s'''_2$, $s''_{1_b} \succ_{\Phi'} s''_2$, $s''_{1_b} \succ_{\Phi'} s''_3$, $s''_2 \succ_{\Phi'} s''_3$ e $s'''_3 \succ_{\Phi'} s'''_4$. Assim, o resultado da consulta é composto pelas sequências s''_{1_a}, s''_{1_b}, s'''_{1_b}, s'''_3, s''_4 (em negrito na Figura 26).

Código 8 – Consulta **Q3** usando os operadores **CONSEQ**, **ENDSEQ** e **MINSEQ**

```
SELECT
  SUBSEQUENCE END POSITION FROM
  SUBSEQUENCE CONSECUTIVE TUPLES FROM
  SEQUENCE IDENTIFIED BY jid [RANGE 20 SECOND]
FROM evento
WHERE MINIMUM LENGTH IS 2
ACCORDING TO TEMPORAL PREFERENCES
  IF FIRST THEN jogada = 'rec' BETTER jogada = 'lbal' AND
  IF PREVIOUS (jogada = 'cond')
    THEN (jogada = 'drib') BETTER (jogada = 'pass')[local] AND
  (jogada = 'pass') BETTER (jogada = 'bpas')[local] AND
  IF ALL PREVIOUS (local = 'io')
    THEN (local = 'io') BETTER (local = 'mc');
```

$\mathbf{s''_{1_a}} = \langle(\mathbf{mc}, \mathbf{pass}), (\mathbf{mc}, \mathbf{lbal})\rangle$
$\mathbf{s''_{1_b}} = \langle(\mathbf{io}, \mathbf{cond}), (\mathbf{io}, \mathbf{drib})\rangle$
$\mathbf{s'''_{1_b}} = \langle(\mathbf{mc}, \mathbf{rec}), (\mathbf{io}, \mathbf{cond}), (\mathbf{io}, \mathbf{drib})\rangle$
$s''_2 = \langle(io, cond), (io, pass)\rangle$
$s'''_2 = \langle(mc, lbal), (io, cond), (io, pass)\rangle$
$s''_3 = \langle(io, cond), (io, bpas)\rangle$
$\mathbf{s''_3} = \langle(\mathbf{io}, \mathbf{rec}), (\mathbf{io}, \mathbf{cond}), (\mathbf{io}, \mathbf{bpas})\rangle$
$\mathbf{s''_4} = \langle(\mathbf{mc}, \mathbf{cond}), (\mathbf{ao}, \mathbf{cond})\rangle$
$s'''_4 = \langle(io, rec), (mc, cond), (ao, cond)\rangle$

Figura 26 – Resultado da consulta **Q3** usando o operador **MINSEQ** no instante 20

6.4 Plano de execução

A Figura 27 apresenta o plano de execução completo para consultas na linguagem StreamPref contendo preferências temporais. Primeiro, o operador **SEQ** extrai as sequências do fluxo. Os operadores **CONSEQ**, **ENDSEQ**, **MINSEQ** e **MAXSEQ** são opcionais, podendo ser usados conforme a necessidade do usuário. O operador **CONSEQ**, é executado logo após o **SEQ** para obter as subsequências-tc. Em seguida, o operador **ENDSEQ** é usado para gerar as subsequências-up. Depois disto, os operadores **MINSEQ** e **MAXSEQ** são utilizados para filtrar as sequências pelo comprimento. E, por fim, os operadores **BESTSEQ** ou **TOPKSEQ** são processados para selecionar as sequências que melhor atendem às preferências temporais da consulta.

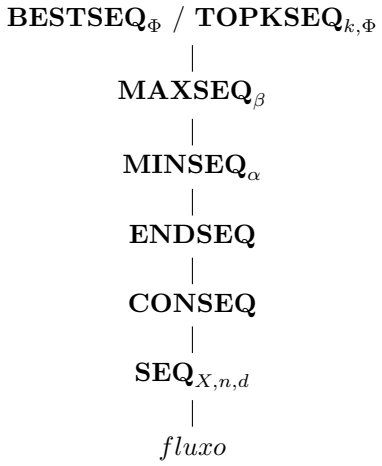


Figura 27 – Plano de execução de consultas StreamPref contendo preferências temporais

6.5 Sintaxe de consultas na linguagem StreamPref

A estrutura do comando `SELECT` para consultas por sequências na linguagem StreamPref é mostrada na Figura 28. Quando a consulta possui a opção `TOP` (linha 2), o operador **TOPKSEQ** é usado para retornar as top-k sequências, sendo que o parâmetro k usado pelo operador será o número inteiro positivo informado logo após a palavra-chave `TOP`.

O bloco `SEQUENCE IDENTIFIED BY...`, nas linhas 5 e 6, é obrigatório e especifica os parâmetros para o operador **SEQ**. Os atributos especificados após `BY` são usados como identificadores, a abrangência temporal é informada após `RANGE` e o deslocamento temporal é informado após `SLIDE`. O parâmetro `SLIDE` é opcional, quando o mesmo é omitido, o operador **SEQ** usa o deslocamento padrão de 1 segundo. Tanto a abrangência quanto o deslocamento devem ser especificados por um número inteiro positivo seguido de uma unidade de tempo. As unidades de tempo suportadas são `SECOND`, `MINUTE`, `HOUR` ou `DAY`.

```
1:  SELECT
2:    TOP(<inteiro>)                                          TOPKSEQ
3:    SUBSEQUENCE END POSITION FROM                           ENDSEQ
4:    SUBSEQUENCE CONSECUTIVE TUPLES FROM                     CONSEQ
5:    SEQUENCE IDENTIFIED BY <atributos>                      SEQ
6:      [RANGE <abrangencia>, SLIDE <deslocamento>]
7:  FROM <fluxo> | <operacao-fluxo> FROM <tabela>
8:  WHERE
9:    MINIMUM LENGTH IS <inteiro>                             MINSEQ
10: AND
11:   MAXIMUM LENGTH IS <inteiro>                             MAXSEQ
12: ACCORDING TO TEMPORAL PREFERENCES <regras-pct>;           BESTSEQ/TOPKSEQ
```

Figura 28 – Estrutura de uma consulta por sequências da linguagem StreamPref

O bloco `SUBSEQUENCE CONSECUTIVE TUPLES...` da linha 4 informa que a consulta deve obter as subsequências-tc com o uso do operador **CONSEQ**. Já o bloco `SUBSEQUENCE END POSITION...` da linha 3 é usado para que a consulta considere as subsequências-up por meio do operador **ENDSEQ**. Quando as duas cláusulas são informadas, os operadores são processados obedecendo a ordem do plano de execução explicado na Seção 6.4, ou seja, primeiro são obtidas as subsequências-tc e sobre estas são computadas as subsequências-up.

A cláusula opcional `WHERE` permite filtrar as sequências por comprimento. O bloco `MINIMUM LENGTH IS...` especifica o comprimento mínimo permitido e o bloco `MAXIMUM LENGTH IS...` estabelece o comprimento máximo permitido. Ambos blocos recebem como parâmetro um número inteiro positivo para informar o comprimento.

A cláusula `ACCORDING TO TEMPORAL PREFERENCES...` especifica as regras de preferências do usuário a serem usadas pelos operadores **BESTSEQ** ou **TOPKSEQ**. As regras devem ser especificadas no formato `IF <condição> THEN preferência` e são separadas pelo conector `AND`. Na linguagem CPrefSQL para banco de dados tradicionais as preferências são criadas separadamente com o comando `CREATE PREFERENCES` para depois serem usadas em uma consulta. No caso da linguagem StreamPref as preferências são submetidas juntamente com a consulta porque, geralmente, em ambientes de dados em fluxos, as consultas são registradas uma única vez e passam a reportar os resultados continuamente. O Apêndice A apresenta a gramática completa da linguagem StreamPref.

6.6 Equivalências CQL

Como foi explicado no Capítulo 1 uma das questões relacionadas à hipótese do trabalho descrito nesta tese é verificar se os novos operadores da linguagem StreamPref aumentam o poder de expressão da linguagem CQL. Esta seção apresenta as equivalências CQL para os operadores StreamPref, mostrando que a StreamPref não aumenta o poder de expressão da CQL. Entretanto, as equivalências não são triviais e envolvem o processamento de diversas operações intermediárias para se atingir o mesmo resultado dos operadores da

StreamPref. Os próximos capítulos mostram que os algoritmos específicos desenvolvidos para os operadores da StreamPref são capazes de realizar um processamento muito mais eficiente do que as equivalências CQL. Além disto, as equivalências CQL podem ser analisadas por trabalhos futuros em busca de otimizações de planos de consulta.

jid	POS	local	jogada
1	1	*id*	*pass*
1	2	*id*	*lbal*
1	3	*id*	*rec*
1	4	*mc*	*cond*
1	5	*mc*	*drib*
2	1	*id*	*lbal*
2	2	*mc*	*cond*
2	3	*mc*	*pass*

jid	POS	local	jogada
3	1	*mc*	*rec*
3	2	*mc*	*cond*
3	3	*mc*	*bpas*
4	1	*mc*	*rec*
4	2	*id*	*cond*
4	3	*io*	*cond*

Figura 29 – Sequências representadas por tuplas

A linguagem CQL não possui estruturas de dados específicas para representar sequências de tuplas, portanto as equivalências a serem apresentadas precisam lidar com tal situação. Apesar das sequências apresentarem características específicas, é possível representá-las usando relações. Para fazer isto, a relação deve possuir, além dos mesmos atributos da sequência, o atributo `POS` para identificar a posição da tupla. Como exemplo, considere as sequências da Figura 23(b). A representação destas sequências em uma relação é mostrada na Figura 29. Neste exemplo, o atributo `jid` é usado para identificar a sequência e o atributo `POS` indica a posição que a tupla ocupa dentro da sequência.

Todas as equivalências demonstradas consideram um fluxo $S(A_1, ...A_l)$ e o conjunto de atributos identificadores de sequências é $X = \{A_1\}$. Porém, pode ser usado qualquer subconjunto de $\{A_1, ..., A_l\}$ como identificador sem perda de generalidade. As subseções a seguir apresentam as equivalências CQL para os operadores da linguagem StreamPref.

6.6.1 Equivalência CQL do operador SEQ

A equivalência CQL para o operador **SEQ** é estabelecida pela Equação (6). O primeiro passo é obter todas as tuplas que serão usadas para construir as sequências, conforme a Equação (6a). O instante da tupla, denotado por $\texttt{TS}(t)$, dá origem ao atributo `POS` para ser usado posteriormente no cálculo da posição correta da tupla dentro da sequência. Em seguida, o operador de janela deslizante da CQL ($\boxplus$) é usado para selecionar as tuplas considerando a abrangência temporal (n) e o intervalo de deslocamento (d). Os símbolos π, $-$, γ, $\bowtie$, σ e $\cup$ representam as operações equivalentes aos operadores de projeção, diferença de conjunto, função de agregação, junção, seleção e união, respectivamente. A notação

$A \mapsto A'$ denota a renomeação do atributo A para A'.

$$W_0 = \pi_{\texttt{POS},A_1}(\boxplus_{n,d}(\textbf{RSTREAM}(\pi_{\texttt{TS}(t)\mapsto\texttt{POS},A_1,...,A_l}(\boxplus_{1,1}(S))))) \tag{6a}$$

$$P_0 = \{\} \tag{6b}$$

$$W_i = W_{i-1} - P_{i-1} \tag{6c}$$

$$P_i = \gamma_{A_1,\min(\texttt{POS})\mapsto\texttt{POS}}(W_i) \bowtie_{A_1,\texttt{POS}} W_0 \tag{6d}$$

$$\textbf{SEQ}_{\{A_1\},n,d}(S) = \pi_{1\mapsto\texttt{POS},A_1,...A_l}(P_1) \cup ... \cup \pi_{n\mapsto\texttt{POS},A_1,...A_l}(P_n) \tag{6e}$$

Depois de obter todas as tuplas necessárias, é preciso computar cada posição das sequências usando as relações W_i e P_i para $i \in \{1, ..., n\}$. A relação W_i contém os identificadores que estão na posição i ou posterior. Já a relação P_i possui somente as tuplas com os identificadores da posição i. Estas relações são necessárias porque pode acontecer de uma sequência não ter tuplas em todos os instantes. Por exemplo, se uma sequência possui uma tupla no segundo instante, mas não tem no primeiro, a posição desta tupla será 1 (um) e não 2 (dois). Por fim, na Equação (6e), as relações P_i são combinadas para criar as sequências completas.

6.6.2 Equivalência CQL do operador CONSEQ

As operações CQL equivalentes ao operador **CONSEQ** são mostradas na Equação 7. A Equação 7 utiliza o operador **SEQ** na equivalência do operador **CONSEQ** porque a equivalência CQL do operador **SEQ** já foi demonstrada na Seção 6.6.1. A relação Z da Equação (7a) possui os mesmos atributos do fluxo S acrescidos do atributo `OTS` que representa o instante original de cada tupla. A Equação (7b) cria a relação Z' com um acréscimo de 1 nos atributos `POS` e `OTS` que também são renomeados para `POS'` e `OTS'`, respectivamente. Já as Equações (7c) e (7d) calculam as posições iniciais e finais das subsequências-tc. Em seguida as posições calculadas por estas equações são usadas pelas Equações (7e) e (7f) para selecionar todas as posições de cada subsequência-tc. Por fim, a Equação (7g) recalcula o atributo `POS` para obter o resultado final contendo as subsequências-tc.

$$Z = \textbf{SEQ}_{\{A_1\},n,d}(\textbf{RSTREAM}(\pi_{\texttt{TS}(t)\mapsto\texttt{OTS},A_1,...,A_l}(\boxplus_{1,1}(S))) \tag{7a}$$

$$Z' = \pi_{(\texttt{POS}+1)\mapsto\texttt{POS},(\texttt{OTS}+1)\mapsto\texttt{OTS'},A_1}(Z) \tag{7b}$$

$$P_s = \pi_{\texttt{POS}\mapsto\texttt{START},A_1}(\sigma_{\texttt{POS}=1}(Z)) \cup \pi_{\texttt{POS}\mapsto\texttt{START},A_1}(\sigma_{\texttt{OTS}\neq\texttt{OTS'}}(Z \bowtie_{\texttt{POS},A_1} Z')) \tag{7c}$$

$$P_e = \pi_{(\texttt{START}-1)\mapsto\texttt{END},A_1}(\sigma_{\texttt{START} > 1}(P_s)) \cup \gamma_{\max(\texttt{POS})\mapsto\texttt{END},A_1}(Z) \tag{7d}$$

$$P_{se} = \gamma_{\texttt{START},\min(\texttt{END})\mapsto\texttt{END},A_1}(\sigma_{\texttt{START}\leq\texttt{END}}(P_s \bowtie_{A_1} P_e)) \tag{7e}$$

$$P = \sigma_{\texttt{POS}\geq\texttt{START}\wedge\texttt{POS}\leq\texttt{END}}(Z \bowtie_{A_1} P_{se}) \tag{7f}$$

$$\textbf{CONSEQ}(\textbf{SEQ}_{\{A_1\},n,d}(S)) = \pi_{(\texttt{POS}-\texttt{START}+1)\mapsto\texttt{POS},A_1,...,A_l}(P) \tag{7g}$$

6.6.3 Equivalência CQL do operador ENDSEQ

A equivalência entre o operador **ENDSEQ** e a linguagem CQL é dada pela Equação 8. Cada relação Z_i da Equação (8a) possui as subsequências-up de comprimento i tal que $i \in \{1, ..., n\}$ onde n é a abrangência temporal usada pela extração de sequências. A Equação (8b) faz a união entre as subsequências das relações $Z_1, ..., Z_n$ para obter todas as subsequências-up.

$$Z_i = \pi_{(\texttt{POS}-i+1)\mapsto\texttt{POS},A_1,...,A_l}(\sigma_{\texttt{POS}\geq i}(Z)) \tag{8a}$$

$$\textbf{ENDSEQ}(Z) = Z_1 \cup ... \cup Z_n \tag{8b}$$

6.6.4 Equivalências CQL dos operadores MINSEQ e MAXSEQ

As operações equivalentes aos operadores **MINSEQ** e **MAXSEQ** em CQL se resumem a selecionar as tuplas de uma relação filtrando pelo atributo `POS` conforme mostrado na Equação (9). A Equação (9a) representa a operação equivalente ao operador **MINSEQ** enquanto a Equação (9b) estabelece a equivalência do operador **MAXSEQ**.

$$\textbf{MINSEQ}_{\alpha}(Z) = \pi'_{A_1}(\sigma_{\texttt{POS}\geq\alpha}(Z)) \ltimes_{A_1} Z \tag{9a}$$

$$\textbf{MAXSEQ}_{\beta}(Z) = (\pi'_{A_1}(Z) - \pi_{A_1}(\sigma_{\texttt{POS}>\beta}(Z))) \ltimes_{A_1} Z \tag{9b}$$

6.6.5 Equivalências CQL dos operadores de preferência

As operações em CQL equivalentes aos operadores de preferência temporal levam em consideração uma relação Z contendo as sequências extraídas pelo operador **SEQ**. Na demonstração de equivalência, a relação Z é obtida pela operação $\textbf{SEQ}_{X,n,d}(S)$ onde $S(A_1, ..., A_l)$. Deste modo, a relação Z possui os mesmos atributos do fluxo S e também o atributo `POS` para identificar a posição das tuplas dentro das sequências.

O primeiro passo da equivalência CQL para o operador **BESTSEQ** é computar a posição onde cada par de sequências deve ser comparado conforme mostrado na Equação (10). A relação Z' é usada apenas para renomear os atributos de Z. Em seguida, esta relação é usada na Equação (10b) para obter todas as posições não correspondentes entre os pares de sequências. Por fim, a Equação (10c) obtém apenas a primeira posição onde as sequências devem ser comparadas.

$$Z' = \pi_{\texttt{POS},A_1\mapsto B',A_2\mapsto A'_2,...,A_l\mapsto A'_l}(Z) \tag{10a}$$

$$P_{nc} = \sigma_{A_2\neq A'_2\vee...\vee A_l\neq A'_l}(\pi_{\texttt{POS},A_1\mapsto B,A_2,...,A_l}(Z) \ltimes_{\texttt{POS}} Z') \tag{10b}$$

$$P = \gamma_{B,B',\min(\texttt{POS})\mapsto\texttt{POS}}(P_{nc}) \tag{10c}$$

O próximo passo é identificar as posições que satisfazem os componentes temporais das condições de cada regra-pct. A Equação (11) estabelece as equivalências para cada

uma das fórmulas derivadas.

$$P^{\mathbf{First}} = \pi_{\mathtt{POS},B}(\sigma_{\mathtt{POS}=1}(P)) \tag{11a}$$

$$P^{\mathbf{Prev}}_{Q(A)} = \pi_{\mathtt{POS},B}(P \bowtie_{\mathtt{POS},B} (\pi_{(\mathtt{POS}+1)\mapsto\mathtt{POS},A_1\mapsto B}(\sigma_{Q(A)}(Z)))) \tag{11b}$$

$$P^{\mathbf{SomePrev}}_{Q(A)} = \pi_{\mathtt{POS},B}(\sigma_{\mathtt{POS}>\mathtt{POS}'}(P \bowtie_B (\gamma_{A_1\mapsto B,\min(\mathtt{POS})\mapsto\mathtt{POS}'}(\sigma_{Q(A)}(Z))))) \tag{11c}$$

$$P^{\max} = \gamma_{A_1\mapsto B,\max(\mathtt{POS})\mapsto\mathtt{POS}}(P) \tag{11d}$$

$$P'_{\neg Q(A)} = \gamma_{B,\min(\mathtt{POS})\mapsto\mathtt{POS}'}(\pi_{\mathtt{POS},A_1\mapsto B}(\sigma_{\neg Q(A)}(Z)) \cup P^{\max}) \tag{11e}$$

$$P^{\mathbf{AllPrev}}_{Q(A)} = \pi_{\mathtt{POS},B}(\sigma_{\mathtt{POS}\leq\mathtt{POS}',\wedge\mathtt{POS}>1}(P \bowtie_B (P'_{\neg Q(A)}))) \tag{11f}$$

A seguir, a Equação (12) computa a relação R_i contendo as posições que satisfazem a condição $C^{\leftarrow}_{\varphi_i} = F_1 \wedge ... \wedge F_p$ para cada regra-pct $\varphi_i \in \Phi$.

$$P_j = \begin{cases} P^{\mathbf{First}}, \textbf{if } F_j = \textbf{First} \\ P^{\mathbf{Prev}}_{Q(A)}, \textbf{if } F_j = \textbf{Prev}(Q(A)) \\ P^{\mathbf{SomePrev}}_{Q(A)}, \textbf{if } F_j = \textbf{SomePrev}(Q(A)) \\ P^{\mathbf{AllPrev}}_{Q(A)}, \textbf{if } F_j = \textbf{AllPrev}(Q(A)) \end{cases} \tag{12a}$$

$$R_i = (P_1) \bowtie_{\mathtt{POS},B} ... \bowtie_{\mathtt{POS},B} (P_p) \tag{12b}$$

A Equação (13) realiza as comparações diretas para cada regra-pct $\varphi_i \in \Phi$. Observe que a Equação (13) considera também as tuplas da relação $\mathbf{Tup}(S)$. Tais tuplas serão necessárias posteriormente para a computação do fecho transitivo. As relações D^+_i e D^-_i representam as tuplas que satisfazem respectivamente os valores preferidos (tuplas dominantes) e os valores não preferidos (tuplas dominadas) de φ_i. As tuplas provenientes da relação Z são chamadas de *tuplas originais* e as tuplas provenientes da relação $\mathbf{Tup}(S)$ são chamadas de *tuplas temporárias*. O atributo A_t é usado para diferenciar estes dois tipos de tuplas de forma que as tuplas originais possuem $A_t = 1$ e as tuplas temporárias possuem $A_t = 0$. Observe também que a Equação (13e) aplica o filtro $E_{\varphi_i} : (A_{i_1} = A'_{i_1}) \wedge ... \wedge (A_{i_j} = A'_{i_j})$ tal que $\{A_{i_1}, ..., A_{i_j}\} = (\{A_1, ..., A_l\} - \{A_{\varphi_i}, B, B'\} - W_{\varphi_i})$. Este filtro é necessário para que a semântica *ceteris paribus* seja obedecida.

$$Z^+_i = \pi_{\mathtt{POS},B}(R_i) \bowtie_{\mathtt{POS},B} (\sigma_{C^{\bullet}_{\varphi_i}\wedge Q^+_{\varphi_i}}(\pi_{\mathtt{POS},A_1\mapsto B,...,A_l,1\mapsto A_t}(Z))) \tag{13a}$$

$$D^+_i = Z^+_i \cup (\pi_{\mathtt{POS},B}(R_i) \bowtie_B (\pi_{A_1\mapsto B,...,A_l,0\mapsto A_t}(\sigma_{C^{\bullet}_{\varphi_i}\wedge Q^+_{\varphi_i}}(\mathbf{Tup}(S))))) \tag{13b}$$

$$Z^-_i = \pi_{\mathtt{POS},B'}(R_i) \bowtie_{\mathtt{POS},B'} (\sigma_{C^{\bullet}_{\varphi_i}\wedge Q^-_{\varphi_i}}(\pi_{\mathtt{POS},A_1\mapsto B',...,A_l,1\mapsto A_t}(Z))) \tag{13c}$$

$$D^-_i = Z^-_i \cup (\pi_{\mathtt{POS},B'}(R_i) \bowtie_{B'} (\pi_{A_1\mapsto B',...,A_l,0\mapsto A_t}(\sigma_{C^{\bullet}_{\varphi_i}\wedge Q^-_{\varphi_i}}(\mathbf{Tup}(S))))) \tag{13d}$$

$$D_i = \sigma_{E_{\varphi_i}}(P \bowtie_{\mathtt{POS},B,B'} (D^+_i \bowtie_{\mathtt{POS}} (\pi_{\mathtt{POS},B',A_2\mapsto A'_2,...,A_l\mapsto A'_l,A_t\mapsto A'_t}(D^-_i)))) \tag{13e}$$

Após computar as comparações diretas, é preciso calcular o fecho transitivo, conforme mostrado na Equação (14). A relação T_1 contém todas as comparações diretas. As relações T'_i, T''_i e T_i são computadas para $i \in \{2, ..., m\}$ onde $m = |\Phi|$ é o número de regras

de Φ. A relação T_m contém todas as comparações impostas por Φ.

$$T_1 = D_1 \cup ... \cup D_m \tag{14a}$$

$$T'_i = \pi_{(\texttt{POS},B,B',A_2,...,A_l,A_t,A'_2 \mapsto A''_2,...,A'_l \mapsto A''_l,A'_t \mapsto A''_t)}(T_{i-1}) \tag{14b}$$

$$T''_i = \pi_{(\texttt{POS},B,B',A_2 \mapsto A''_2,...,A_l \mapsto A''_l,A_t \mapsto A''_t,A'_2,...,A'_l,A'_t)}(T_{i-1}) \tag{14c}$$

$$T_i = \pi_{\texttt{POS},B,B',A_2,...,A_l,A_t,A'_2,...,A'_l,A'_t}(T'_i \bowtie_{\texttt{POS},B,B',A''_2,...,A''_l} T''_i) \cup T_{i-1} \tag{14d}$$

Após o cálculo do fecho transitivo, é possível selecionar as posições de sequências que foram dominadas conforme a Equação (15). Note que são considerados apenas os casos em que uma tupla original domina outra tupla original ($A_t = 1 \wedge A'_t = 1$). Esta equação utiliza a projeção com eliminação de duplicatas (π') para que cada identificador de sequência seja considerado uma única vez.

$$\textbf{BESTSEQ}_\Phi(Z) = Z \bowtie_{A_1} (\pi'_{A_1}(Z) - \pi'_{B' \mapsto A_1}(\sigma_{A_t=1 \wedge A'_t=1}(T_m))) \tag{15}$$

Como foi discutido na Seção 6.2.2, o operador **TOPKSEQ** pode ser obtido em função do operador **BESTSEQ**, portanto a obtenção das top-k sequências também possui operação equivalente em CQL.

6.7 Comparação com trabalhos correlatos

A Tabela 3 apresenta uma análise comparativa entre a linguagem StreamPref e os demais trabalhos correlatos. Os critérios de comparação considerados na Tabela 3 são os seguintes:

- Modelo de preferência: *skyline*, função *score*, regras de preferências condicionais (regras-pc) e regras de preferências condicionais temporais (regras-pct);
- Tipos de atributos considerados: numéricos (Num.) e categóricos (Cat.);
- Tipo de consulta: *skyline* (S), *top-k* (TK) e *top-k* dominante (TKD);
- Tipo de resultado retornado: tuplas (Tup.) e sequência de tuplas (Seq.).

A principal contribuição do trabalho descrito nesta tese é a especificação da linguagem de consulta StreamPref destacada em cinza na Tabela 3. Além das vantagens relativas ao modelo de preferência descritas no Capítulo 5, a linguagem StreamPref permite a realização consultas contínuas contendo tanto preferências condicionais quanto preferências condicionais temporais.

6.8 Considerações finais

Este capítulo abordou a linguagem de consulta StreamPref capaz de lidar com sequências e preferências temporais. Foram detalhados a semântica dos operadores

Tabela 3 – Análise comparativa dos trabalhos relacionados ao processamento de consultas contínuas com suporte a preferências

Trabalho	Modelo de Preferências	Atributos		Tipo de Consulta			Resultado	
		Num.	Cat.	S	TK	TKD	Tup.	Seq.
(ZHU et al., 2017)	*score*	✓			✓		✓	
(SANTOSO; CHIU, 2014)	*skyline*	✓	✓			✓	✓	
(LEE; LEE; KIM, 2013)	*skyline*	✓		✓			✓	
(SHEN et al., 2012)	*score*	✓			✓		✓	
(PETIT et al., 2012)	regras-pc	✓	✓			✓	✓	
(KONTAKI; PAPADOPOULOS; MANOLOPOULOS, 2012)	*skyline*	✓				✓	✓	
(AMO; BUENO, 2011)	regras-pc	✓	✓			✓	✓	
(SARKAS et al., 2008)	*skyline*		✓	✓			✓	
(MORSE; PATEL; GROSKY, 2007)	*skyline*	✓		✓			✓	
(MOURATIDIS; BAKIRAS; PAPADIAS, 2006)	*score*	✓			✓		✓	
(TAO; PAPADIAS, 2006)	*skyline*	✓		✓			✓	
(LIN et al., 2005)	*skyline*	✓		✓			✓	
StreamPref	regras-pc e regras-pct	✓	✓			✓	✓	✓

StreamPref, o plano de execução das consultas, a sintaxe da linguagem e as equivalências CQL dos operadores.

Os operadores da linguagem StreamPref não aumentam o poder de expressão da linguagem CQL, ou seja, todos os operadores especificados podem ser escritos por meio de operações equivalentes na linguagem CQL. Entretanto, a utilização dos novos operadores possibilita o desenvolvimento de algoritmos específicos mais eficientes para a execução das consultas do que suas equivalências em CQL. Além disto, a sintaxe da linguagem StreamPref permite a escrita de consultas contendo preferências temporais de uma forma mais intuitiva e compacta. Como foi mostrado através das equivalências, na linguagem CQL, é preciso encadear diversas operações complexas e não intuitivas para se ter o mesmo resultado de uma consulta StreamPref.

Existe a possibilidade de criar outros operadores de subsequência para a linguagem StreamPref que poderiam quebrar as sequências de outras maneiras. Uma possibilidade seria por exemplo obter todas as subsequências possíveis de uma sequência s. Entretanto, isto causaria uma explosão no número de subsequências. Além disto, a comparação de sequências pelos operadores de preferências consideram a primeira posição distinta. Considerando o operador **ENDSEQ**, as subsequências-up retornadas já apresentam todas as primeiras posições distintas o que devem proporcionar as mesmas chances de comparação deste possível operador.

O próximo capítulo tratará dos algoritmos desenvolvidos para processar os operadores da linguagem StreamPref. A grande maioria destes algoritmos funciona de forma incremental para que as consultas da linguagem StreamPref possam ser processadas de forma eficiente.

Capítulo 7

Algoritmos dos operadores StreamPref

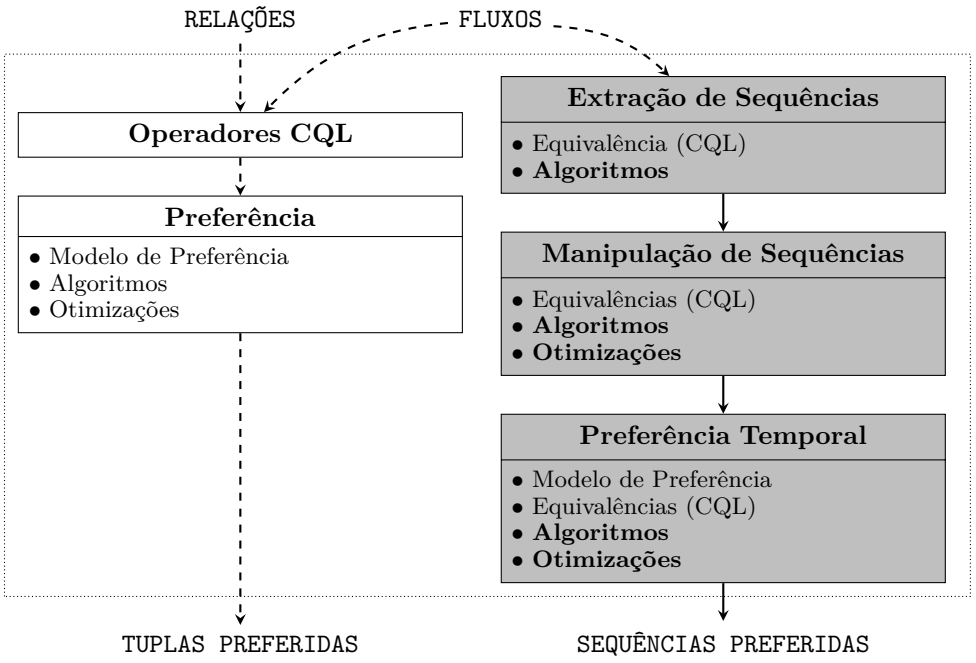


Figura 30 – Visão geral da linguagem StreamPref destacando os algoritmos e otimizações

O presente capítulo trata do processamento dos operadores da linguagem StreamPref conforme destacado (em cinza) na Figura 30. As seções a seguir apresentam os algoritmos para cada um dos operadores (RIBEIRO et al., 2017a; RIBEIRO et al., 2017c). A Seção 7.1 descreve o algoritmo do operador **SEQ**. A Seção 7.2 apresenta os dois tipos de algoritmos (ingênuo e incremental) para processar os operadores **CONSEQ** e **ENDSEQ**. Já a Seção 7.3 explica como funcionam os algoritmos dos operadores **MINSEQ** e **MAXSEQ**. A Seção 7.4 apresenta os algoritmos para realizar o teste de dominância entre sequências e processar os operadores de preferência **BESTSEQ** e **TOPKSEQ**. Por fim, a Seção 7.5 apresenta as considerações finais sobre o capítulo.

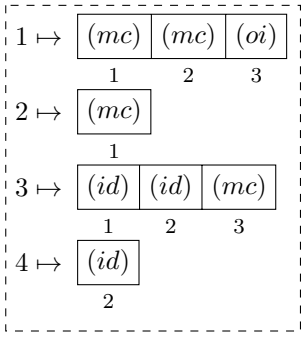
7.1 Algoritmo do operador SEQ

O processamento do operador **SEQ** é feito de forma incremental e utiliza uma tabela *hash* $Z_{\#} : (x \mapsto s_x)$ que mapeia cada identificador x para sua sequência identificada correspondente. A cada instante de tempo, as novas tuplas são inseridas em suas sequências correspondentes e as tuplas que *expiraram* são removidas.

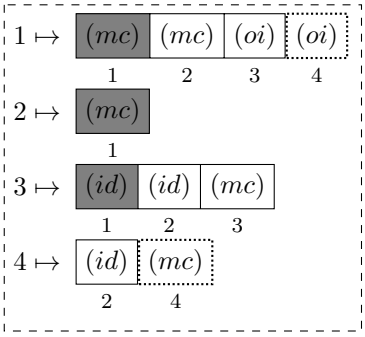
Como exemplo, considere o fluxo `posicionamento(jid, local)` da Figura 31(a) e a operação $\mathbf{SEQ}_{3,1}$(`posicionamento`). A Figura 31(b) apresenta a tabela *hash* $Z_{\#}$ no instante 3 e a Figura 31(c) mostra a tabela *hash* $Z_{\#}$ no instante 4. No instante 4, as tuplas do instante 1 (em cinza) expiram e são removidas das sequências. Já as novas tuplas (em pontilhado) são anexadas no final das sequências correspondentes.

TS()	jid	local
1	1	mc
1	2	mc
1	3	id
2	1	mc
2	3	id
2	4	id
3	1	oi
3	3	mc
4	1	oi
4	4	mc

(a) Fluxo `posicionamento`



(b) $Z_{\#}$ no instante 3



(c) $Z_{\#}$ no instante 4

Figura 31 – Funcionamento do algoritmo *IncSeq*

Algoritmo 21 – *IncSeq*(S, X, n, d)

Entrada: Fluxo S, conjunto de atributos identificadores X, abrangência temporal n e intervalo de deslocamento d
Atualização: Tabela *hash* de sequências $Z_{\#}$
Saída: Conjunto de sequências identificadas do instante atual

```
1:  RemoveExpired(n, d)                          //Remove as posições expiradas
2:  for all t ∈ S do                             //Para cada tupla t de S
3:      if START_d(t) ≤ τ_now ≤ END_{n,d}(t) then   //Verifica se o instante de t é valido
4:          x ← π_X(t)                           //Obtém o identificador de sequência x
5:          if x ∈ Z_# then                      //Testa já existe x em Z_#
6:              s_x ← Z_#(x) + ⟨t/X⟩             //Acrescenta t à sequência existente s_x
7:          else
8:              s_x ← ⟨t/X⟩                      //Cria uma nova sequência s_x
9:          Z_#(x) ← s_x                         //Atualiza a tabela hash Z_#
10: return {Z_#.values()}                        //Retorna as sequências de Z_#
```

O processamento do operador **SEQ** é feito por meio do algoritmo *IncSeq* (Algoritmo 21). Como o algoritmo *IncSeq* trabalha de forma incremental, sua execução ocorre a cada instante. A tabela *hash* $Z_{\#}$ é criada antes da primeira execução e passa a ser

atualizada a cada execução do algoritmo. O primeiro passo do algoritmo é chamar a função *RemoveExpired* (Algoritmo 22) para remover as tuplas expiradas das sequências existentes.

Após a remoção das tuplas expiradas, o algoritmo *IncSeq* processa todas as tuplas do instante atual ($t \in S$). Primeiro o algoritmo verifica se cada tupla t deve ser considerada no instante atual de acordo com as funções $\texttt{START}_d(t)$ e $\texttt{END}_{n,d}(t)$. Conforme explicado no Capítulo 2, estas funções delimitam a porção de tuplas válidas do fluxo considerando a abrangência temporal n e o intervalo de deslocamento d. Caso a tupla t seja válida, o algoritmo obtém o identificador de sequência x associado a t.

O identificador x é usado para verificar se a sequência s_x já se encontra armazenada na tabela *hash* $Z_\#$. Em caso afirmativo, a tupla t é acrescentada na posição final da sequência existente. Quando a entrada x não existe em $Z_\#$, o algoritmo cria a nova sequência $s_x \leftarrow \langle t/X \rangle$ e insere s_x em $Z_\#$. Observe que a tupla t tem os atributos identificadores X removidos antes de ser inserida na sequência s_x. Ao final, o algoritmo retorna todas as sequências armazenadas em $Z_\#$. O Apêndice B apresenta um exemplo de execução completo do algoritmo *IncSeq*.

A função *RemoveExpired* percorre todas as sequências armazenadas em $Z_\#$ e remove as posições expiradas. No final, as sequências que tiveram todas as suas posições removidas são removidas de $Z_\#$.

Algoritmo 22 – *RemoveExpired*(n, d)

Entrada: Abrangência temporal n e intervalo de deslocamento d
Atualização: Tabela *hash* $Z_\#$

```
1: for all x ∈ Z_# do                                   //Para cada entrada x de Z_#
2:     s_x ← Z_#(x)                                     //Obtém a sequência s_x associada a x
3:     while |s_x| ≥ 1 and not START_d(s_x[1]) ≤ τ_now ≤ END_{n,d}(s_x[1]) do //Testa a posição 1 de s_x
4:         s_x ← s_x[2, |s_x|])                         //Remove a primeira posição
5:     if |s_x| > 0 then                                //Verifica se s_x não é vazia
6:         Z_#(x) ← s_x                                 //Atualiza tabela hash
7:     else
8:         Z_#.del(x)                                   //Remove a entrada x de Z_#
```

Análise de Complexidade

A complexidade do algoritmo *IncSeq* é calculada em função do custo da função *RemoveExpired* e do número de tuplas do instante atual. Assumindo um tempo contante de $O(1)$ para as operações com tabelas *hash*, a função *RemoveExpired* tem um custo de $O(z \times |s_x|)$, onde z é o número total de identificadores existentes. No pior caso $|s_x| = n$, desta maneira a complexidade da função *RemoveExpired* é $O(zn)$. Após a chamada da função *RemoveExpired*, o algoritmo *IncSeq* se resume a percorrer as tuplas do instante atual e as sequências da tabela *hash* $Z_\#$. À medida que o algoritmo é processado por mais

iterações, o número de entradas de $Z_{\#}$ e o número de tuplas de cada instante são, no pior caso, $O(z)$. Portanto, a complexidade do algoritmo *IncSeq* é de $O(zn + z + z) = O(zn)$.

7.2 Algoritmos dos operadores CONSEQ e ENDSEQ

Para os operadores **CONSEQ** e **ENDSEQ** foram desenvolvidos algoritmos usando duas abordagens diferentes: uma abordagem *ingênua* e uma abordagem *incremental*. No cenário de dados em fluxos as abordagens incrementais tendem a ser mais eficientes do que as abordagens ingênuas. Com a utilização das duas abordagens, esta tendência pode ser melhor analisada por meio de experimentos comparativos.

A abordagem ingênua segue as linhas do algoritmo *NaiveSubseq* (Algoritmo 23) de forma que, a cada instante, todas as subsequências precisam ser reconstruídas. Observe que o algoritmo *NaiveSubseq* é executado a cada instante e todas as posições de todas as sequências do instante atual precisam ser varridas para a construção das subsequências.

Algoritmo 23 – *NaiveSubseq*(Z)

Entrada: Conjunto de sequências Z
Saída: Conjunto de subsequências sobre Z
1: $Z' \leftarrow \{\}$ //Conjunto de subsequências a ser retornado
2: **for all** $s \in Z$ **do** //Para cada sequência s em Z
3: $Z' \leftarrow Z' \cup \{\text{subsequências de } s\}$ //Percorre todas as posições de s para obter as subsequências
4: **return** Z' //Retorna o conjunto de subsequências obtidas

No caso da abordagem incremental, os algoritmos consideram apenas as remoções e inserções do instante anterior para atualizar as subsequências já construídas. Para realizar este processo, usam os seguintes atributos das sequências:

key: Valor inteiro único gerado na criação da sequência;
deleted: Número de remoções no instante anterior;
inserted: Número de inserções no instante anterior.

Observe que o atributo *key* é diferente do identificador da sequência. Dada uma sequência s_x identificada por x, todas as subsequências de s_x tem o mesmo identificador x, mas o valor de *key* é único em cada sequência. Os atributos *deleted* e *inserted* são atualizados automaticamente quando as sequências sofrem alguma inserção ou remoção.

O funcionamento das abordagens incrementais depende da alteração automática dos atributos *inserted* e *deleted* por todos os algoritmos envolvidos. Antes de cada iteração, os algoritmos devem reiniciar estes atributos em todas as sequências. Contudo isto não afeta a complexidade dos algoritmos uma vez que eles precisam acessar cada sequência pelo menos uma vez.

7.2.1 Algoritmo incremental do operador CONSEQ

O algoritmo incremental para processar o operador **CONSEQ** utiliza uma lista L associada a cada sequência s para obter de forma eficiente as subsequências-tc de s. A Figura 32(a) mostra uma sequência s com sua respectiva lista de subsequências L. Os números abaixo das posições representam seus instantes originais. A cada iteração o algoritmo remove as tuplas removidas do início da lista e acrescenta as novas tuplas no final da lista. Como exemplo, considere as alterações feitas na sequência s da Figura 32(b). As duas primeiras posições (em cinza) expiraram e uma nova (em pontilhado) foi acrescentada. Com a utilização da lista, apenas a primeira subsequência é descartada e apenas a última subsequência sofre alterações.

(a) Sequência s e sua lista de subsequências L (b) Atualização da lista L

Figura 32 – Sequência e sua lista de subsequências-tc

O algoritmo *IncConseq* (Algoritmo 24) emprega a abordagem incremental para processar o operador **CONSEQ**. A tabela *hash* $L_{\#}$ é usada para associar o atributo *key* de cada sequência s com sua respectiva lista de subsequências L.

Algoritmo 24 – *IncConseq*(Z)

Entrada: Conjunto de sequências Z
Atualização: Tabela *hash* $L_{\#}$
Saída: Conjunto de subsequências-tc sobre Z

```
1:  Z' ← {}                                  //Conjunto de subsequências a ser retornado
2:  N# ← NewHashTable()                      //Nova tabela hash vazia
3:  for all s ∈ Z do                         //Para cada sequência s de Z
4:      if s.key ∈ L# then                   //Verifica se existe lista para s
5:          L ← L#(s.key)                    //Obtém a lista L associada a s
6:          DeleteConseq(L, s)               //Remove as posições expiradas das subsequências
7:          InsertConseq(L, s)               //Insere as novas posições
8:      else
9:          L ← NewConseq(s)                 //Constrói uma nova lista de subsequências
10:     Z' ← Z' ∪ L                          //Acrescenta as subsequências de L em Z'
11:     N#(s.key) ← L                        //Armazena a lista L associada a s em N#
12: L# ← N#                                  //Atualiza tabela hash L# usando N#
13: return Z'                                //Retorna as subsequências
```

O primeiro passo do algoritmo *IncConseq* é criar o conjunto Z' para guardar as subsequências a serem retornadas. O algoritmo também cria uma nova tabela *hash* $N_{\#}$ para armazenar apenas as listas das sequências já processadas. Para cada sequência s

processada, o algoritmo verifica se já existe uma lista associada a s. Se existir, a lista existente é apenas atualizada. Caso contrário, uma nova lista é criada. As subsequências de cada lista L são adicionadas ao conjunto Z' que, por sua vez, é retornado no final do algoritmo. Antes de retornar o conjunto Z', o algoritmo atualiza a tabela *hash* $L_{\#}$ para ser usada na próxima iteração.

A criação de uma lista contendo as subsequências de uma sequência é feita pela função *NewConseq*. Esta função percorre todas as posições de uma sequência para obter as subsequências. A atualização de uma lista existente é realizada pelas funções *DeleteConseq* (Algoritmo 25) e *InsertConseq* (Algoritmo 26). O Apêndice B apresenta um exemplo de execução completo do algoritmo *IncConseq*.

Algoritmo 25 – *DeleteConseq*(L, s)

Entrada: Lista L e sequência s
Atualização: Lista L

```
1: count ← 0                                          //Contador de posições removidas count
2: while count < s.deleted do                         //Enquanto count for menor do que o número de remoções
3:     ss ← L.popFirst()                              //Remove a primeira subsequência de L
4:     count ← count + |ss|                           //Considera as posições da subsequência removida
5: if count > s.deleted then                          //Checa se houve remoções em excesso
6:     count ← s.deleted − (count − |ss|)             //Número posições a remover em ss
7:     ss ← ss[count + 1, |ss|]                       //Remove as posições necessárias de ss
8:     L.pushFirst(ss)                                //Devolve ss para o início de L
```

A função *DeleteConseq* considera as remoções da sequência s e atualiza sua lista de subsequências L. O processo de atualização envolve a remoção das subsequências do início de L até que o número de remoções seja atingido. À medida que as subsequências são removidas, a função contabiliza o número de posições já removidas usando o contador *count*. Se o contador for maior do que o número de remoções, significa que apenas algumas posições da última sequência ss precisam ser consideradas. Então, a função remove somente as posições necessárias de ss e devolve esta subsequência para o início da lista L.

Algoritmo 26 – *InsertConseq*(L, s)

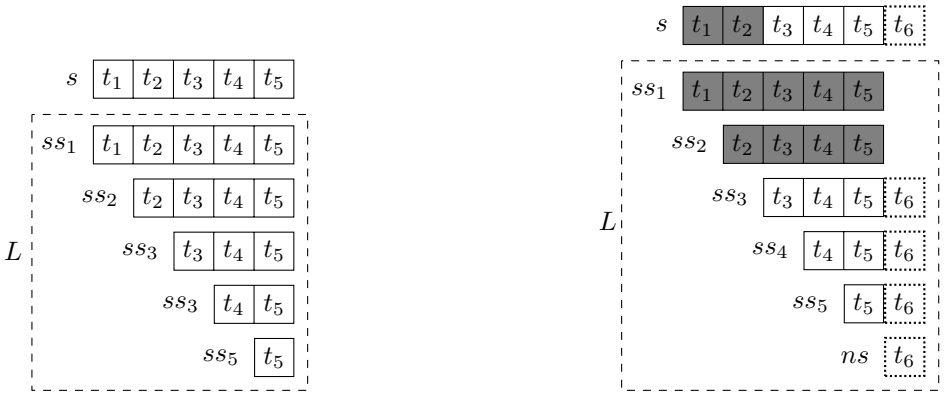
Entrada: Lista L e sequência s
Atualização: Lista L

```
1: if s.inserted > 0 then                             //Verifica se houve inserções em s
2:     L' ← NewConseq(s[|s| − s.inserted + 1, |s|])   //Cria a lista L' com as novas posições
3:     ss ← L.popLast()                               //ss é última subsequência de L
4:     ns ← L'.popFirst()                             //ns é a primeira subsequência de L'
5:     if TS(ss[|ss|]) + 1 = TS(ns[1]) then           //Verifica se ss e ns podem ser concatenadas
6:         ss ← ss + ns                               //Concatena ss e ns
7:     else
8:         L'.pushFirst(ns)                           //Devolve ns para o início de L'
9:     L.pushLast[ss]                                 //Devolve ss para o final de L
10:    L ← L + L'                                     //Concatena as listas L e L'
```

O primeiro passo da função *InsertConseq* é verificar se houveram inserções na sequência s. Se existirem novas posições inseridas, a função cria uma lista L' contendo as subsequências com estas novas posições. Em seguida, a função retira as subsequências ss (última de L) e ns (primeira de L') para verificar se o instante da última posição de ss e o instante da primeira posição de L' são consecutivos. Se forem, estas duas subsequências são concatenadas, senão ns é devolvida para sua posição original. Por fim, a função devolve a subsequência ss para L e acrescenta todas as sequências de L' em L.

7.2.2 Algoritmo incremental do operador ENDSEQ

O algoritmo incremental do operador **ENDSEQ** associa a cada sequência s uma lista L. Esta lista contém as subsequências-up de s ordenadas de forma decrescente pelo comprimento, como mostrado na Figura 33(a). A lista de subsequências L é alterada de acordo com as inserções e remoções na sequência s. Todas as subsequências com posições removidas são consideradas inválidas.



(a) Sequência s e sua lista de subsequências L (b) Atualização da lista L

Figura 33 – Sequência e sua lista de subsequências-up

Uma subsequência-up de s se torna inválida quando seu comprimento é maior do que o comprimento de s no instante anterior subtraído pelo número de remoções. Como exemplo, considere a sequência s com comprimento 5 mostrada na Figura 33(a). Na Figura 33(b) é mostrado o instante seguinte, quando s tem 2 posições removidas (em cinza). Deste modo, as subsequências ss_1 e ss_2 (em cinza) são inválidas pois possuem comprimento maior que 3 $(5 - 2)$.

Após a remoção das sequências inválidas, as posições inseridas são usadas para complementar as subsequências remanescentes e criar novas subsequências. Na Figura 33(b), a tupla t_6 (em pontilhado) foi inserida. Esta tupla é acrescentada às subsequências ss_3, ss_4 e ss_5 (que não foram removidas) e a nova sequência $ns = \langle t_6 \rangle$ é criada.

O algoritmo *IncEndseq* (Algoritmo 27) utiliza a abordagem incremental para realizar o processamento do operador **ENDSEQ**. Este algoritmo é muito similar ao algoritmo

IncConseq, a diferença está nas chamadas às rotinas específicas de cada algoritmo. Para cada sequência s, o algoritmo verifica se já existe a lista de subsequências associada a s. Se já existe uma lista L associada a s, o algoritmo atualiza L usando as funções *DeleteEndseq* e *InsertEndseq*. Em caso negativo, esta lista é criada usando a função *NewEndseq*. No final, o algoritmo retorna as subsequências contidas em Z'. O Apêndice B apresenta um exemplo de execução completo do algoritmo *IncEndseq*.

Algoritmo 27 – *IncEndseq*(Z)

Entrada: Conjunto de sequências Z
Atualização: Tabela *hash* $L_{\#}$
Saída: Conjunto de subsequências-up de Z

```
1:  Z' ← {}                                  //Conjunto de subsequências a ser retornado
2:  N_# ← NewHashTable()                     //Nova tabela hash vazia
3:  for all s ∈ Z do                         //Para cada sequência s de Z
4:      if s.key ∈ L_# then                  //Verifica se existe lista associada a s
5:          L ← L_#(s.key)                   //Obtém a lista L associada a s
6:          DeleteEndseq(L, s)               //Remove as subsequências inválidas
7:          InsertEndseq(L, s)               //Insere as novas posições
8:      else
9:          L ← NewEndseq(s)                 //Constrói uma nova lista de subsequências
10:     Z' ← Z' ∪ L                          //Acrescenta subsequências de L em Z'
11:     N_#(s.key) ← L                       //Armazena a lista L associada a s em N_#
12: L_# ← N_#                                //Atualiza tabela hash L_# usando N_#
13: return Z'                                //Retorna as subsequências
```

A função *DeleteEndseq* (Algoritmo 28) recebe uma sequência s juntamente com sua lista de subsequências L e remove as subsequências inválidas contidas em L. Para um melhor desempenho, as sequências são mantidas em ordem decrescente pelo tamanho dentro da lista.

Algoritmo 28 – *DeleteEndseq*(L, s)

Entrada: Lista L e sequência s
Atualização: Lista L

```
1: if s.deleted > 0 then                     //Verifica se s possui remoções
2:     maxlen ← |s| − s.inserted             //Obtém o tamanho máximo válido
3:     while |L| > 0 do                      //Enquanto houverem sequências em L
4:         ss ← L.popFirst()                 //Pega a primeira subsequência ss de L
5:         if |ss| ≤ maxlen then             //Verifica se ss possui tamanho válido
6:             L.pushFirst(ss)               //Insere ss no início de L
7:             break                         //Interrompe o laço de repetição
```

A função *DeleteEndseq* começa verificando se existem remoções em s. Quando há remoções, a função calcula o comprimento máximo das subsequências válidas (*maxlen*). As sub-sequências do início de L (aquelas com tamanhos maiores) são removidas e a função testa se o maior tamanho válido já foi atingido. A função termina ao encontrar uma sequência ss com tamanho válido.

A função *InsertEndseq* (Algoritmo 29) é responsável por acrescentar as inserções da sequência s às subsequências da lista L e também criar novas subsequências com estas posições. Quando existem inserções em s ($s.inserted > 0$), a função cria uma subsequência ns contendo as inserções e concatena esta subsequência no final de todas as subsequências já existentes. Em seguida, todas as subsequências de ns contendo a última posição de ns também são acrescentadas à lista L.

Algoritmo 29 – *InsertEndseq*(L, s)

Entrada: Lista L e sequência s
Atualização: Lista L

```
1: if s.inserted > 0 then                          //Verifica se s possui inserções
2:     ns ← s[|s| − s.inserted + 1, |s|]           //Obtém a sequência ns com as posições inseridas
3:     for ss ∈ L do                               //Para cada subsequência ss ∈ L
4:         ss ← ss + ns                            //Concatena ss com ns
5:     L ← L ∪ NewConseq(ns)                       //Adicionas as subsequências de ns em L
```

7.2.3 Análise de complexidade

A complexidade do algoritmo ingênuo *NaiveSubseq* é $O(zn)$ para o operador **CONSEQ** e $O(zn^2)$ para o operador **ENDSEQ**, onde $z = |Z|$ é o número de sequências recebidas e n é o comprimento da maior sequência recebida. O custo do *NaiveSubseq* para o operador **ENDSEQ** é maior porque o algoritmo precisa processar uma sequência múltiplas vezes para obter todas as suas subsequências-up.

A complexidade do algoritmo *IncConseq* depende do custo das funções *DeleteConseq*, *InsertConseq* e *NewConseq*. As funções *DeleteConseq* e *InsertConseq* têm os custos $O(d)$ e $O(i)$, respectivamente, onde d é o maior número de remoções e i é o maior número de inserções. Já a função *NewConseq* tem o custo de $O(i^2)$. Desta maneira, a complexidade do algoritmo *IncConseq* é $O(zd + zi + zi^2)$. Como foi descrito no Capítulo 6, o operador **CONSEQ** obtém subsequências a partir das sequências extraídas pelo operador **SEQ**. Na prática, o número de inserções nunca é maior do que um porque a cada instante a operação de extração de sequências (operador **SEQ**) vai receber no máximo uma nova tupla. Isto faz com que a complexidade do algoritmo *IncConseq* seja reduzida para $O(zd)$.

A complexidade do algoritmo *IncEndseq* está relacionada com o custo das funções *DeleteEndseq*, *InsertEndseq* e *NewEndseq*. Com a ordenação da lista L, o custo da função *DeleteEndseq* é $O(d)$. A função *InsertEndseq* tem o custo de $O(n - d)$ para atualizar as sequências existentes uma vez que o algoritmo *DeleteConseq* já removeu d das n subsequências existentes. Já a construção das novas subsequências pela função *NewEndseq* tem o custo de $O(i^2)$. Assim, o custo final da função *InsertEndseq* é de $O(n - d + i^2)$. Logo, a complexidade do algoritmo *IncEndseq* é $O(z \times (d + n - d + i^2)) = O(zn + zi^2)$. Considerando novamente que o número máximo de inserções é um, esta complexidade passa a ser $O(zn)$.

7.3 Algoritmos dos operadores MINSEQ e MAXSEQ

O processamento dos operadores **MINSEQ** e **MAXSEQ** é feito conforme mostrado no algoritmo *FilterByLength* (Algoritmo 30). O algoritmo consiste em varrer todas as sequências recebidas e verificar o comprimento das mesmas. As sequências que não possuem comprimento válido de acordo com os parâmetros α e β são descartadas. A implementação do algoritmo usa um atributo especial para armazenar o comprimento da sequência, evitando assim a varredura de todas as posições da sequência para obter seu comprimento. Portanto, a complexidade do algoritmo *FilterByLength* é $O(z)$, onde $z = |Z|$ é o número de sequências recebidas.

Algoritmo 30 – $FilterByLength(Z)$

```
1: Z' ← {}                          //Cria o conjunto Z' para armazenar as sequências a serem retornadas
2: for all s ∈ Z do                 //Para cada sequência s ∈ Z
3:     if |s| é válido then         //Verifica se s possui um comprimento válido
4:         Z' ← Z' ∪ {s}            //Adiciona s em Z'
5: return Z'                        //Retorna as sequências com tamanho válido
```

7.4 Algoritmos dos operadores BESTSEQ e TOPKSEQ

No processamento dos operadores de preferência **BESTSEQ** e **TOPKSEQ**, assim como no processamento dos operadores de subsequência, foram consideradas duas abordagens: ingênua e incremental. A Seção 7.4.1 descreve a algoritmo ingênuo. Em seguida, o algoritmo incremental é explicado na Seção 7.4.2. Depois, a Seção 7.4.3 apresenta a análise de complexidade dos algoritmos.

7.4.1 Algoritmo ingênuo

O processamento ingênuo do operador **BESTSEQ** é feita por meio do algoritmo *GetBestSeq* (Algoritmo 31). Primeiro, o algoritmo copia as sequências recebidas para o conjunto Z'. Em seguida, para cada par de sequências s e s' de Z', o algoritmo testa se $s \succ_\Phi s'$ ou $s' \succ_\Phi s$ usando a função *Dominates* (Algoritmo 32). Ao final, as sequências dominadas são descartadas e somente aquelas que são dominantes permanecem em Z'. O Apêndice B apresenta um exemplo de execução completo do algoritmo *GetBestSeq*.

A função *Dominates* realiza o teste de dominância entre duas sequências s e s' de acordo com uma teoria-pct Φ. Primeiro, o menor tamanho entre as duas sequências recebidas é calculado e armazenado em j. Em seguida, a função varre as posições de 1 a j das sequências em busca da primeira posição não correspondente. Quando a posição

Algoritmo 31 – *GetBestSeq*(Φ, Z)

Entrada: Conjunto de sequências Z e teoria-pct Φ
Saída: Sequências dominantes de Z

```
1: Z' ← Z                                          //Copia as sequências recebidas para Z'
2: for all s, s' ∈ Z' do                           //Para cada par de sequências s, s' em Z'
3:     if Dominates(Φ, s, s') then                 //Testa se s ≻_Φ s'
4:         Z' ← Z' − {s'}                          //Remove a sequência dominada s' de Z'
5:     else if Dominates(Φ, s', s) then            //Testa se s' ≻_Φ s
6:         Z' ← Z' − {s}                           //Remove a sequência dominada s de Z'
7: return Z'                                       //Retorna somente as sequências dominantes
```

não corresponde i é encontrada, todas as regras-pct de Φ válidas em $s[i]$ e $s'[i]$ são usadas para criar a teoria-pc Γ.

A obtenção de uma regra-pc a partir de uma regra-pct φ consiste em remover os componentes temporais de φ conforme explicado no Capítulo 5. A teoria-pc Γ construída é usada então para comparar as tuplas $s[i]$ e $s'[i]$ usando o algoritmo *SearchDom* (Algoritmo 1). O algoritmo *SearchDom*, apresentado no Capítulo 3, usa a técnica de busca em profundidade para comparar duas tuplas de acordo com uma teoria-pc. Desta maneira, o resultado da comparação das sequências s e s' é o resultado da comparação das tuplas $s[i]$ e $s'[i]$. O algoritmo *Dominates* retorna **false**, quando a posição de comparação não é encontrada.

Algoritmo 32 – *Dominates*(Φ, s, s')

Entrada: Teoria-pct Φ e duas sequências s e s'
Saída: **true**, se s domina s'. Caso contrário, **false**

```
1: j ← min{|s|, |s'|}                              //Obtém o tamanho j da menor sequência
2: for all i ∈ {1, ..., j} do                      //Percorre as posições de 1 a j das sequências
3:     if s[i] ≠ s'[i] then                        //Testa se posição atual é diferente nas sequências
4:         Γ ← {}                                  //Cria uma teoria-pc Γ vazia
5:         for all φ ∈ Φ do                        //Para cada regra-pct φ de Φ
6:             if ((s, i) ⊨ C_φ) and ((s', i) ⊨ C_φ) then   //Testa se a posição i de s e s' satisfaz C_φ
7:                 Γ ← Γ ∪ {φ•}                    //Adiciona a regra-pc φ• em Γ
8:         return SearchDom(Γ, s[i], s'[i])        //Testa se s[i] ≻_Γ s'[i]
9: return false
```

O algoritmo *GetTopkSeq* (Algoritmo 33) realiza a processamento ingênua do operador **TOPKSEQ**. Inicialmente, todas as sequências são movidas para o conjunto Z^+. A cada iteração do laço de repetição, as sequências são separadas em dominantes (Z^+) e dominadas (Z^-) por meio da função *SeparateSequences* (Algoritmo 34). As sequências dominantes são inseridas no final da lista L e as sequências dominadas são movidas para Z^+ para que possam ser separadas novamente na próxima iteração.

A lista L armazena as sequências ordenadas de forma crescente pelo nível de preferência. Na primeira iteração, as sequências com nível zero são acrescentadas a lista, na segunda iteração, são acrescentadas as sequências com nível um e assim por diante.

Algoritmo 33 – *GetTopkSeq*(Φ, Z, k)

Entrada: Conjunto de sequências Z, teoria-pct Φ e inteiro k
Saída: Top-k sequências de Z

```
1: Z+ ← Z                                          //Copia as sequências recebidas para Z+
2: Z− ← {}                                         //Conjunto para armazenar as sequências dominadas
3: L ← NewList()                                   //Lista para guardar as sequências ordenadas pelo nível
4: while (|L| < k) and (|Z+| > 0) do               //Enquanto |L| < k e |Z+| > 0
5:     Z+, Z− ← SeparateSequences(Z+, Φ)           //Separa Z+ em dominantes e dominadas
6:     L.append(Z+)                                //Acrescenta as sequências dominantes à lista L
7:     Z+ ← Z−                                     //Move as sequências dominadas de Z− para Z+
8: return L.getFirst(k)                            //Retorna as primeiras k sequências de L
```

As iterações do laço terminam quando L possui, no mínimo, k sequências ou todas as sequências recebidas já se encontram em L. No final, o algoritmo retorna as primeiras k sequências de L.

A função *SeparateSequences* recebe os mesmos parâmetros e trabalha de forma semelhante ao algoritmo *GetBestSeq*. A diferença é que a função *SeparateSequences* retorna dois conjuntos disjuntos, um contendo as sequências dominantes e outro contendo as sequências dominadas.

Algoritmo 34 – *SeparateSequences*(Φ, Z)

Entrada: Conjunto de sequências Z e teoria-pct Φ
Saída: Sequências dominantes e dominadas de Z

```
1: Z+ ← Z                                          //Copia as sequências recebidas para Z+
2: Z− ← {}                                         //Conjunto para armazenar as sequências dominadas
3: for all s, s' ∈ Z+ do                           //Para cada par de sequências s, s' em Z+
4:     if Dominates(Φ, s, s') then                 //Testa se s ≻Φ s'
5:         Z+ ← Z+ − {s'}                          //Remove a sequência dominada s' de Z+
6:         Z− ← Z− ∪ {s'}                          //Adiciona sequência dominada s' em Z−
7:     else if Dominates(Φ, s', s) then            //Testa se s' ≻Φ s
8:         Z+ ← Z+ − {s}                           //Remove a sequência dominada s de Z+
9:         Z− ← Z− ∪ {s}                           //Adiciona sequência dominada s' em Z−
10: return Z+, Z−                                  //Retorna sequências dominantes e dominadas
```

7.4.2 Algoritmo incremental

O processamento ingênuo dos operadores de preferência não usa qualquer informação do instante anterior. Isto pode causar a repetição desnecessária de testes de dominância. Por exemplo, em um instante, existem duas sequências s e s' tais que $s \succ_\Phi s'$. Se, no instante seguinte, estas sequências não sofrerem qualquer alteração, a comparação já realizada prevalece a mesma. Os algoritmos incrementais foram criados para reduzir este tipo de problema.

O processamento incremental dos operadores de preferência utiliza uma estrutura de índice baseada em árvore chamada de *árvore de sequências*. Esta estrutura permite saber

onde duas sequências devem ser comparadas e também mantém hierarquias de preferências para as tuplas das sequências nas posições onde devem acontecer as comparações. A árvore de sequências possui uma raiz na profundidade zero. Começando na profundidade um, os nós da árvore passam a ser associados às tuplas das sequências. Seja $s = \langle t_1, ..., t_z \rangle$. Cada tupla t_i de s é representada por um nó. A primeira tupla (t_1) se torna um nó filho da raiz. A partir de então, todo nó t_i é pai do nó t_{i+1}. A sequência s fica armazenada no último nó t_n. O Exemplo 27 mostra como armazenar sequências em uma árvore de sequências.

Exemplo 27 (Árvore de sequências)**.** Considere as seguintes sequências sobre os atributos `local` e `jogada`:

$s_1 = \langle (mc, pass), (mc, lbal) \rangle$;
$s_2 = \langle (mc, rec), (io, cond), (io, drib) \rangle$;
$s_3 = \langle (io, cond), (io, drib) \rangle$;
$s_4 = \langle (mc, lbal), (io, cond), (io, pass) \rangle$;
$s_5 = \langle (io, cond), (io, pass) \rangle$;
$s_6 = \langle (io, rec), (io, cond), (io, bpas) \rangle$;
$s_7 = \langle (io, cond), (io, bpas) \rangle$;
$s_8 = \langle (io, rec), (mc, cond), (ao, cond) \rangle$;
$s_9 = \langle (mc, cond), (ao, cond) \rangle$.

A Figura 34 mostra como estas sequências são armazenadas em uma árvore de sequências. A raiz da árvore, na profundidade 0, é representada pelo círculo preto. Os números ao lado dos retângulos pontilhados indicam a profundidade dos nós.

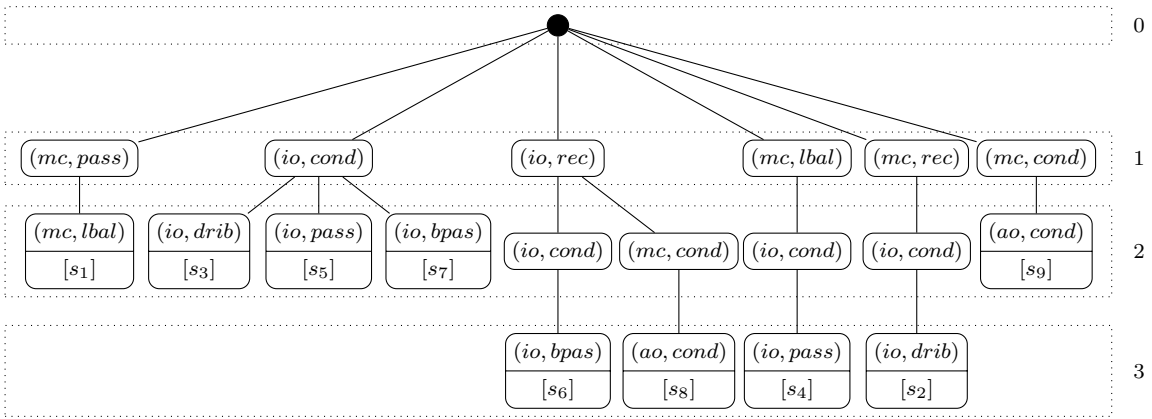


Figura 34 – Árvore de sequências

As ramificações da árvore indicam onde as sequências devem ser comparadas. Se ocorre uma ramificação na profundidade d, as sequências das ramificações distintas devem ser comparadas na posição $d+1$. Como exemplo, observe as sequências s_6 e s_8 do Exemplo 27, elas estão no mesmo ramo até a profundidade 1, onde ocorre uma ramificação que as separa. Isto significa que estas sequências devem ser comparadas na posição 2 que corresponde a primeira posição diferente nas sequências.

A árvore é alterada quando alguma sequência sofre alterações ou quando um nova sequência é inserida. As novas sequências são inseridas a partir da raiz da árvore. As alterações nas sequências podem ser causadas por inserções de novas posições (depois da última posição) ou por remoção (nas primeiras posições). Quando uma ou mais posições iniciais são removidas, a sequência precisa ser realocada na árvore pois o caminho da raiz para a sequência deve ser composto pelas tuplas da sequência. No caso das inserções, o início da sequência permanece o mesmo e a sequência é apenas movida para um ramo filho. As sequências vazias podem ser simplesmente removidas da árvore.

7.4.2.1 Hierarquias de preferência

A notação $nd.t$ denota a tupla t associada ao nó nd. A profundidade de um nó nd é denotada por $depth(nd)$. Já as sequências armazenadas em um nó nd são representadas pela notação $nd.Z$. Os filhos de um nó nd são armazenados na tabela *hash* $nd.Child_{\#}$ que mapeia cada tupla t para seu respectivo nó. Sendo assim, $nd.Child_{\#}(t)$ denota o nó filho de nd associado à tupla t. Além disto, cada nó nd possui uma hierarquia de preferência $nd.H$ sobre as tuplas associadas a seus filhos. Tal hierarquia permite saber se um filho de nd é dominante ou dominado e, por consequência, determinar se as sequências de um ramo são dominantes ou dominadas.

A construção da hierarquia de preferência $nd.H$ começa com a seleção das regras-pct temporalmente válidas nos filhos de nd. Estas regras são usadas para criar uma teoria-pc Γ que, por sua vez, é empregada na construção da base de conhecimento K_Γ. O processo de construção da base de conhecimento é o mesmo apresentado no Capítulo 3. A hierarquia de preferência dos nós usa as mesmas estruturas dos algoritmos baseados em particionamento do Capítulo 4. O Exemplo 28 mostra a hierarquia de preferência de um nó da árvore de sequências da Figura 34.

Exemplo 28 (Hierarquia de um nó)**.** Considere novamente a teoria-pct $\Phi' = \{\varphi_4, \varphi_5, \varphi_6, \varphi_7\}$ do Exemplo 23, onde:

φ_4 : **First** $\rightarrow$ (`jogada` $= rec$) $\succ$ (`jogada` $= lbal$);
φ_5 : **Prev**(`jogada` $= cond$) $\rightarrow$ (`jogada` $= drib$) $\succ$ (`jogada` $= pass$)[`local`];
φ_6 : $\rightarrow$ (`jogada` $= pass$) $\succ$ (`jogada` $= bpas$)[`local`];
φ_7 : **AllPrev**(`local` $= io$) $\rightarrow$ (`local` $= io$) $\succ$ (`local` $= mc$).

Leve em consideração também o nó $(io, cond)$ na profundidade 1 da árvore de sequências da Figura 34. Supondo que a teoria-pct Φ' seja usada para comparar as sequências. A hierarquia de preferência do nó $(io, cond)$ deve usar as regras-pct de Φ' temporalmente válidas na sequência $\langle (io, cond), t\rangle$ onde t é qualquer filho de $(io, cond)$. Para a validação temporal das regras, t pode ser qualquer tupla uma vez que tal validação é feita apenas com as tuplas posicionadas antes de t. Neste caso, as regras-pct temporalmente válidas são φ_5,

φ_6 e φ_7. A hierarquia $(io, cond).H$ é construída usando a base de conhecimento K_Γ onde $\Gamma = \{\varphi_5^\bullet, \varphi_6^\bullet, \varphi_7^\bullet\}$. A base de conhecimento K_Γ é composta pelas seguintes comparações:

b_1 : $(\texttt{jogada} = pass) \succ (\texttt{jogada} = bpas)[\texttt{local}, \texttt{jogada}]$;
b_2 : $(\texttt{jogada} = drib) \succ (\texttt{jogada} = bpas)[\texttt{local}, \texttt{jogada}]$;
b_3 : $(\texttt{jogada} = drib) \succ (\texttt{jogada} = pass)[\texttt{local}, \texttt{jogada}]$;
b_4 : $(\texttt{local} = io) \succ (\texttt{local} = mc)[\texttt{local}]$.

A Figura 35(a) mostra como as tuplas associadas aos nós filhos de $(io, cond)$ são particionadas segundo as comparações de K_Γ. A hierarquia de preferência $(io, cond).H$ é exibida na Figura 35(b). Com tal hierarquia é possível determinar que o filho $(io, drib)$ é dominante e os demais filhos são dominados.

	Partições
b_1	$(1) = \{(io, drib), (io, pass), (io, bpas)\}$ $\quad +\quad -$
b_2	$(2) = \{(io, drib), (io, pass), (io, bpas)\}$ $\quad +\quad -$
b_3	$(3) = \{(io, drib), (io, pass), (io, bpas)\}$ $\quad +\quad -$
b_4	$4, (drib) = \{(io, drib)\}$ $\quad +$ $4, (pass) = \{(io, pass)\}$ $\quad +$ $4, (bpas) = \{(io, bpas)\}$ $\quad +$

(a) Partições

	Partição	$Pref_\#$	$NPref_\#$
b_1	(1)	1	$\{(io, bpas)\}$
b_2	(2)	1	$\{(io, bpas)\}$
b_3	(3)	1	$\{(io, pass)\}$
b_4	$(4, drib)$	1	$\{\}$
b_4	$(4, pass)$	1	$\{\}$
b_4	$(4, bpas)$	1	$\{\}$

	$Count_\#$
$(io, drib)$	0
$(io, pass)$	1
$(io, bpas)$	2

(b) Hierarquia de preferência

Figura 35 – Partições e hierarquia de preferência sobre os filhos do nó $(io, cond)$

Como foi explicado no Capítulo 4, a construção da base de conhecimento não é uma tarefa trivial. Analisando a estrutura de uma árvore de sequências, é possível usar uma *estratégia de poda* para evitar a construção de bases de conhecimento e hierarquias de preferência em todos os nós. Um nó com filho único não precisa de hierarquia de preferência, uma vez que este filho sempre será dominante. Além disto, se um filho é dominado, todas as sequências armazenadas neste filho e em seus decrescentes serão dominadas. Portanto não é necessário criar hierarquias de preferência para tais nós.

Outra otimização usada na implementação é um dicionário de bases de conhecimento compartilhado pelos nós. Este dicionário armazena, para cada teoria-pc Γ, sua respectiva base de conhecimento K_Γ. Desta maneira, antes de criar uma base de conhecimento, os algoritmos verificam se a mesma já existe. Em caso afirmativo, basta utilizar a base de conhecimento existente, inclusive aquelas já criadas em instantes anteriores. O Exemplo 29 mostra uma árvore de sequências usando a estratégia de poda.

Exemplo 29 (Árvore de sequências usando estratégia de poda)**.** A Figura 36 mostra novamente a árvore de sequências da Figura 34, agora com os nós dominantes destacados em cinza. Aplicando a estratégia de poda, apenas o nó raiz e os nós $(io, cond)$ e (io, rec),

em cinza escuro, possuem hierarquia de preferência. Os demais nós não precisam de hierarquia de preferência porque possuem um único filho ou são descendentes de nós dominados.

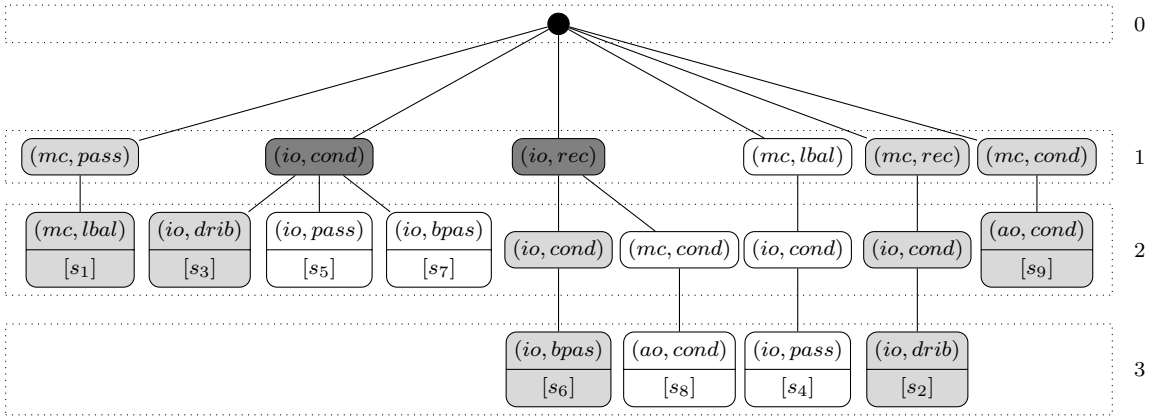


Figura 36 – Árvore de sequências com hierarquias de preferência

7.4.2.2 Algoritmos

Os algoritmos para processamento incremental dos operadores **BESTSEQ** e **TOPKSEQ** utilizam a árvore de sequências e a tabela *hash* $SeqNod_{\#}$. Esta tabela mapeia cada atributo $s.key$ para (s, nd) onde nd é o nó onde s está armazenada. O atributo key é o identificador único de sequência explicado na Seção 7.2. A raiz da árvore de sequências é denotada por *root*. A atualização do índice baseado em árvore de sequências é feita pelo algoritmo *IndexTreeUpdate* (Algoritmo 35).

O primeiro passo do algoritmo *IndexTreeUpdate* é criar o conjunto I. Este conjunto será preenchido com as novas sequências e sequências com posições removidas para que, no final, estas sequências sejam inseridas pela raiz da árvore. Em seguida, o algoritmo percorre todas as entradas já existentes na tabela *hash* $SeqNod_{\#}$. Para cada entrada key, o algoritmo obtém a sequência s correspondente e verifica se houveram remoções e inserções em s. Quando s possui remoções, o algoritmo remove s do índice para incluí-la novamente mais tarde. Se s possuir apenas inserções, o algoritmo move a sequência para um ramo descendente usando a função *AddSeq* (Algoritmo 36).

O segundo laço de repetição do algoritmo *IndexTreeUpdate* percorre as sequências do conjunto Z e verifica se tais sequências já se encontram na estrutura $SeqNod_{\#}$. As sequências não encontradas são adicionadas ao conjunto I. O último laço de repetição do algoritmo apenas insere todas as sequências de I nas estruturas do índice. Por fim, o algoritmo chama a função *Clean* para remover os nós vazios da árvore. Esta função inicia na raiz e trabalha de forma recursiva para realizar um varredura completa.

A função *AddSeq* faz a inserção na árvore a partir de um nó *nd*. Inicialmente, a função testa se a profundidade do nó *nd* é igual ao comprimento da sequência s. Em caso

Algoritmo 35 – *IndexTreeUpdate*(Φ, Z)

Entrada: Conjunto de sequências Z e teoria-pct Φ
Atualização: Estrutura $SeqNod_{\#}$ do índice
Saída: Sequências dominantes e dominadas de Z

```
1: I ← {}                                          //Conjunto de sequências a serem inseridas na raiz
2: for all key ∈ SeqNod# do                        //Para cada entrada key em SeqNod#
3:     (s, nd) ← SeqNod#(key)                      //Obtém a sequência s e nó nd associados à key
4:     if s.deleted > 0 then                       //Verifica se s teve posições removidas
5:         nd.Z ← nd.Z − {s}                       //Remove s das sequências armazenadas em nd
6:         SeqNod#.del(key)                        //Remove a entrada key de SeqNod#
7:         if |s| > 0 then                         //Testa se s não ficou vazia
8:             I ← I ∪ {s}                         //Adiciona s a I
9:     else if s.inserted > 0 then                 //Checa se s teve posições inseridas
10:        nd.Z ← nd.Z − {s}                       //Remove s das sequências armazenadas em nd
11:        new ← AddSeq(nd, s)                     //Move a sequência para o nó new
12:        SeqNod#(key) ← (s, new)                 //Atualiza a entrada key em SeqNod#
13: for all s ∈ Z do                               //Para cada sequência s in Z
14:    if key ∉ SeqNod# then                       //Verifica não existe a entrada key em SeqNod#
15:        I ← I ∪ {s}                             //Adiciona s a I
16: for all s ∈ I do                               //Para cada sequência s em I
17:    nd ← AddSeq(root, s)                        //Insere s iniciando pela raiz da árvore
18:    SeqNod#(key) ← (s, nd)                      //Associa (s, nd) à entrada key em SeqNod#
19: Clean(root)                                    //Remove nós vazios da árvore
```

afirmativo, significa que o caminho completo contendo todas as tuplas de s já encontra-se armazenado na árvore. Sendo assim, o nó nd equivale a última tupla de s e, portanto, s é armazenada neste nó.

Algoritmo 36 – *AddSeq*(nd, s)

Entrada: Nó nd e sequência s
Atualização: Nó nd
Saída: Nó onde s é armazenada

```
1: d ← depth(nd)                                   //Armazena a profundidade de nd em d
2: if d = |s| then                                 //Verifica se o comprimento de s é igual a d
3:     nd.Z ← nd.Z ∪ {s}                           //Armazena s no nó nd
4:     return nd                                   //Retorna o nó nd
5: t ← s[d + 1]                                    //Copia para t a tupla da posição d + 1 de s
6: if t ∈ nd.Child# then                           //Testa se nd possui um filho associado a t
7:     child ← nd.Child#(t)                        //Se sim, copia tal filho para child
8: else
9:     child ← NewChild(nd, t)                     //Se não, cria um novo filho child para nd
10: return AddSeq(child, s)                        //Faz uma chamada recursiva sobre child
```

Quando a profundidade de nd não corresponde ao comprimento de s, a função *AddSeq* pega a tupla t na posição $d+1$ de s e verifica se nd possui um filho *child* associado a esta tupla. Se o filho não existir, a função o cria por meio da função *NewChild*. Por fim, a função faz uma chamada recursiva sobre o filho *child*.

As hierarquias de preferências devem ser atualizadas sempre que um nó é removido ou inserido. Estas ações ocorrem na função *AddSeq* (quando um nó filho é criado) e na função *Clean* (quando um nó vazio é removido). Se um nó nd tem um filho *child* inserido ou removido, a hierarquia de preferência $nd.H$ é atualizada. A atualização de $nd.H$ é feita com base na tupla $child.t$. O processo de atualização é feito como descrito no Capítulo 4.

O Algoritmo *IncBestSeq* (Algoritmo 37) realiza o processamento incremental do operador **BESTSEQ** usando a árvore de sequências. O algoritmo inicia na raiz da árvore e faz sucessivas chamadas recursivas sobre os filhos dominantes de cada nó. O Apêndice B apresenta um exemplo de execução completo do algoritmo *IncBestSeq*.

Algoritmo 37 – *IncBestSeq*(nd)

Entrada: Nó nd
Saída: Sequências armazenadas em nd e nos descendentes dominantes de nd

```
1: Z ← nd.Z                                  //Inicializa Z com as sequências armazenadas em nd.Z
2: for all filho dominante child de nd do    //Para cada filho dominante child de nd
3:     Z ← Z ∪ IncBestSeq(child)             //Obtém as sequências de child e de seus descendentes dominantes
4: return Z                                  //Retorna as sequências dominantes
```

O algoritmo *IncTopkSeq* (Algoritmo 38) realiza a processamento incremental do operador **TOPKSEQ** usando a árvore de sequências. Primeiro, o algoritmo usa a função *CopyTree* para criar uma cópia da árvore de sequências em $root'$. Esta cópia é necessária porque o algoritmo altera a árvore de sequências e a estrutura da árvore original não deve ser perdida.

O algoritmo cria também uma lista L para armazenar as sequências a serem retornadas. Em seguida, o algoritmo inicia o laço de repetição para obter as sequências. As sequências são inseridas na lista L ordenadas pelo nível de preferência. Na primeira iteração, são obtidas as sequências com nível de preferência 0, na segunda iteração, as sequências com nível de preferência 1 e assim por diante. Cada iteração usa a função *RemoveBestFromTree* (Algoritmo 39) para obter as sequências do nível atual. Por fim, o algoritmo retorna as k primeiras sequências de L.

Algoritmo 38 – *IncTopkSeq*(k)

Entrada: Número de top-k sequências a serem retornadas
Saída: Top-k sequências

```
1: root' ← CopyTree(root)                          //Cria uma cópia da árvore de sequências
2: L ← NewList()                                   //Lista de sequências a serem retornadas L
3: while (|L| < k) and not IsEmpty(root') do       //Enquanto |L| < k e a árvore não for vazia
4:     Z' ← RemoveBestFromTree(root')              //Obtém as sequências do nível atual
5:     L.append(Z')                                //Adiciona as sequências no final da lista L
6: return L.getFirst(k)                            //Retorna as k primeiras sequências de L
```

A função *RemoveBestFromTree* utiliza recursão para obter as sequências do nó nd e de seus descendentes dominantes. Inicialmente, a função move as sequências armazenadas

Algoritmo 39 – *RemoveBestFromTree*(*nd*)

Entrada: Número de top-k sequências a serem retornadas
Saída: Top-k sequências

```
1: Z ← nd.Z                                   //Obtém as sequências do nó nd
2: nd.Z ← {}                                  //Remove as sequências do nó nd
3: for all filho dominante child de nd do     //Para cada filho dominante child de nd
4:     Z ← Z ∪ RemoveBestFromTree(child)      //Remove as sequências de child e de seus
   descendentes dominantes
5:     if IsEmpty(child) then                 //Verifica se o filho child ficou vazio
6:         RemoveChild(nd, child)             //Remove o filho child do nó nd
7: return Z                                   //Retorna as sequências removidas
```

do nó *nd* para o conjunto Z. Em seguida, é feita uma chamada recursiva para cada filho dominante *child* de *nd*. Após as chamadas recursivas, a função testa se o filho *child* tornou-se um nó vazio (sem sequências ou filhos). Em caso afirmativo, o filho *child* é removido. Ao final, as sequências de *nd* e de seus descendentes dominantes são retornadas.

7.4.3 Análise de complexidade

A função *Dominates*, ao encontrar a posição de comparação, constrói uma teoria-pc Γ e chama a função *SearchDom* para comparar tal posição. No pior caso, a posição de comparação é a última posição das sequências. Sendo assim, para cada regra-pct, a função *Dominates* precisa varrer todas as posições das sequências para verificar se a regra é válida na posição de comparação. Como foi explicado no Capítulo 3, a função *SearchDom* tem o custo de $O(lm^m)$, onde l é o número de atributos e m é o número de regras da teoria-pc Γ. Logo, o custo da função *Dominates* é $O(mn + lm^m)$ onde n é o comprimento da menor sequência.

A complexidade do algoritmo *GetBestSeq* é $O(z^2 \times (mn + lm^m))$, onde z é o número de sequências recebidas, n é o maior comprimento dentre estas sequências e m é o número de regras da teoria-pct. O fator z^2 representa o custo de buscar os pares de sequências de Z para comparação. Já o termo $(mn + lm^m)$ é o custo do teste de dominância feito pela função *Dominates*.

A complexidade do algoritmo *GetTopkSeq* é calculada em função do custo da função *SeparateSequences* e do número de vezes que ela é executada. A função *SeparateSequences* tem o mesmo custo do algoritmo *GetBestSeq*. O número de chamadas à função depende do nível de preferência máximo imposto pela teoria-pct Φ que, no pior caso, é igual ao número de regras ($m = |\Phi|$). Logo, a complexidade do algoritmo *GetTopkSeq* é $O(m \times (z^2 \times (mn + lm^m))) = O(z^2 \times (m^2 n + lm^{m+1}))$.

A complexidade do algoritmo *IndexTreeUpdate* depende do custo das funções *AddSeq* e *Clean*. No pior cenário, a inserção de uma sequência pela função *AddSeq* tem que criar todos os nós associados às tuplas das sequências e adicionar a sequência em um nó folha. Neste caso, a função cria n nós onde $n = |s|$ é o número de posições da sequência. Além

disto, a criação dos novos nós precisa atualizar as hierarquias de preferências existentes. A atualização de uma hierarquia de preferência custa $O(m^{4l})$ onde m é o número de regras e l é o número de atributos. O custo desta atualização é semelhante ao custo do algoritmo *PartitionInsert* do Capítulo 4. Porém, a função *AddSeq* causa a inserção de um única tupla em cada hierarquia. Desta maneira, o custo da função *AddSeq* é $O(nm^{4l})$.

A função *Clean* precisa varrer todos os nós da árvore para remover os nós vazios. No pior caso, o grau dos nós das árvores é z e a profundidade máxima da árvore é n. Deste modo, o custo para varrer todos os nós da árvore é $O(z^n)$. Além de varrer os nós, a função precisa remover aqueles que estão vazios o que causa a atualização das hierarquias de preferência existentes. Portanto, o custo da função *Clean* é $O(z^n m^{4l})$. No algoritmo *IndexTreeUpdate*, a função *Clean* é chamada uma única vez e a função *AddSeq* é chamada no máximo z vezes. Logo, a complexidade do algoritmo *IndexTreeUpdate* é $O(znm^{4l} + z^n m^{4l}) = O(z^n m^{4l})$.

A algoritmo *IncBestSeq*, no pior caso, precisa varrer toda a árvore de sequências. Em cada nó visitado, o algoritmo usa a estrutura $Count_{\#}$ da hierarquia de preferências do nó para selecionar os filhos dominantes. Portanto, a complexidade do algoritmo *IncBestSeq* é $O(z^n)$.

O algoritmo *IncTopkSeq* faz uma cópia da árvore de sequências e sucessivas chamadas à função *RemoveBestFromTree*. A cópia da árvore custa $O(z^n)$. A função *RemoveBestFromTree* varre a árvore, remove os nós dominantes e atualiza as hierarquias de preferência. A varredura completa da árvore tem o custo de $O(z^n)$. Já a atualização da hierarquia de preferência custa $O(zm^{4l})$. Assim, o custo da função *RemoveBestFromTree* é de $O(z^n \times zm^{4l})$. O número máximo de chamadas à função *RemoveBestFromTree* é igual ao nível de preferência máximo imposto pela teoria-pct que, por sua vez, é igual ao número de regras da teoria-pct. Assim, a complexidade do algoritmo *IncTopkSeq* é $O(m \times z^n \times zm^{4l}) = O(z^{n+1} \times m^{4l+1})$.

7.5 Considerações finais

Este capítulo apresentou os algoritmos para processamento dos operadores da linguagem StreamPref. Para o operador **SEQ**, foi projetado um algoritmo que trabalha de forma incremental. Os algoritmos dos operadores **MINSEQ** e **MAXSEQ** consistem em varrer o conjunto de sequências recebido e descartar as sequências com comprimento inválido. No caso dos operadores **CONSEQ**, **ENDSEQ BESTSEQ** e **TOPKSEQ**, foram criadas abordagens ingênuas e incrementais para os algoritmos. As abordagens incrementais utilizam estruturas de dados específicas que são atualizadas de acordo com as posições alteradas nas sequências, evitando assim a varredura completa das sequências a cada instante.

Em aplicações práticas de dados em fluxo, os algoritmos incrementais devem ter um

desempenho superior em relação a abordagem ingênua. O próximo capítulo apresenta um extensivo conjunto de experimentos com os algoritmos desenvolvidos, incluindo inclusive comparações entre as versões ingênuas e incrementais dos mesmos.

Capítulo 8

Experimentos

Como foi explicado no Capítulo 1, o trabalho descrito nesta tese foi dividido em três etapas. O presente capítulo apresenta os experimentos conduzidos no decorrer de cada uma destas etapas. Todos os experimentos foram conduzidos em um computador com as seguintes especificações:

- Processador de doze núcleos com frequência de 3.2 GHz;
- Memória RAM de 32 GB;
- Sistema operacional Linux.

Para os experimentos da primeira etapa, os algoritmos foram implementados por meio de funções dentro do SGBD relacional PostgreSQL utilizando a linguagem de programação Python[1]. As consultas processadas por estes experimentos foram submetidas ao SGBD PostgreSQL que, por sua vez, executa as funções implementadas.

No caso da segunda e terceira etapas, todos os algoritmos foram implementados em um protótipo de sistema de gerenciamento de fluxos de dados (SGFD) usando a linguagem de programação Python, inclusive os operadores da linguagem CQL com os quais alguns algoritmos são comparados (RIBEIRO et al., 2018)[2]. A linguagem Python foi escolhida para obtenção de uma maior agilidade no desenvolvimento do protótipo uma vez que a mesma possui diversas bibliotecas que facilitam a programação. Posteriormente, as partes críticas do SGFD pode ser gradativamente implementadas em uma linguagem de mais baixo nível para ganho de desempenho.

Todos os experimentos foram executados três vezes para calcular o intervalo de confiança dos resultados. As barras de erro não foram plotadas para não prejudicar a legibilidade dos gráficos. Contudo os intervalos de confiança dos resultados dos experimentos são apresentados do Apêndice C ao Apêndice E.

O restante deste capítulo está organizado como se segue. A Seção 8.1 descreve os experimentos realizados na primeira etapa. Na Seção 8.2 são explicados os experimentos

[1] <http://cprefsql.github.io/>
[2] <http://streampref.github.io/>

da segunda etapa. A Seção 8.3 apresenta os experimentos da terceira etapa. Por fim, a Seção 8.4 conclui o capítulo.

8.1 Primeira etapa de experimentos

Esta seção apresenta os experimentos conduzidos na primeira etapa do trabalho descrito nesta tese (RIBEIRO; PEREIRA; DIAS, 2016). Os experimentos visam comparar o desempenho dos algoritmos para processamento de consultas com preferências condicionais em bancos de dados tradicionais. A comparação leva em consideração os algoritmos do estado-da-arte *BNL*** e *R-BNL*** propostos por Pereira e Amo (2010) e as otimizações *BNL-KB*, *R-BNL-KB*, *PartitionBest* e *PartitionTopk* descritas no Capítulo 3. Os intervalos de confiança destes experimentos são apresentados no Apêndice C.

Os experimentos foram realizados sobre bases de dados sintéticas geradas pelo TPC Benchmark H (TPC-H) (TPC, 2014)[3]. As consultas do *benchmark* usadas nos experimentos passaram por algumas modificações para que os algoritmos pudessem ser avaliados. As modificações foram as seguintes:

1) Remoção das funções de agregação desnecessárias para as comparações realizadas;
2) Inclusão da cláusula `ACCORDING TO PREFERENCES` contendo as preferências;
3) Mudanças da cláusula `WHERE` para conter menos restrições impostas pelas seleções.

As consultas Q3, Q5, Q10 e Q18 foram usadas nos experimentos devido à expressiva variação no número de tuplas submetidas aos operadores de preferência. No caso do tamanho padrão de banco de dados, estes números são 698, 7596, 18644 e 34363 para as consultas Q3, Q5, Q10 e Q18,respectivamente.

As regras de preferências dos experimentos foram construídas levando em consideração o número de regras e o nível de preferência máximo. A escolha destes parâmetros se deve à sua interferência no desempenho dos algoritmos. Enquanto o número de regras afeta o número de comparações diretas entre tuplas, o nível de preferência máximo está relacionado ao número de chamadas recursivas do teste de dominância baseado em busca e ao número de comparações da base de conhecimento.

A Tabela 4 apresenta todos os parâmetros usados nos experimentos juntamente com seus valores padrões e variações. As notações vDB, vQUE, vLEV, vRUL e vTOP são usadas para indicar a variação de valores nos parâmetros DB, QUE, LEV, RUL e TOP, respectivamente.

Os experimentos com variações nos parâmetros da Tabela 4 permitem comparar o desempenho e a escalabilidade dos algoritmos. Os parâmetros LEV e RUL influenciam diretamente a eficiência dos algoritmos ao lidar com diferentes tipos de preferências. Já o parâmetro TOP permite avaliar os algoritmos que processam o operador **TOPK** (para

[3] <http://www.tpc.org/tpch/>

Tabela 4 – Parâmetros dos experimentos com os algoritmos *BNL***, *BNL-KB* e *PartitionBest*

Parâmetro	**Valor Padrão**	**Variação**
Tamanho da base em MB (DB)	32	16, 32, 64, 128, 256, 512 e 1024
Consulta (QUE)	Q5	Q3, Q5, Q10 e Q18
Número de regras (RUL)	6	6, 10, 20, 30 e 40
Nível de preferência máximo (LEV)	2	1, 2, 3, 4, 5 e 6
Número de top-k tuplas (TOP)	-1	-1, 10, 100, 1000 e 5000

$k > -1$). Quando TOP=-1, o operador **BEST** é utilizado. Os parâmetros DB e QUE estão relacionados com a escalabilidade dos algoritmos, ou seja, como os algoritmos se comportam para processar diferentes volumes de tuplas.

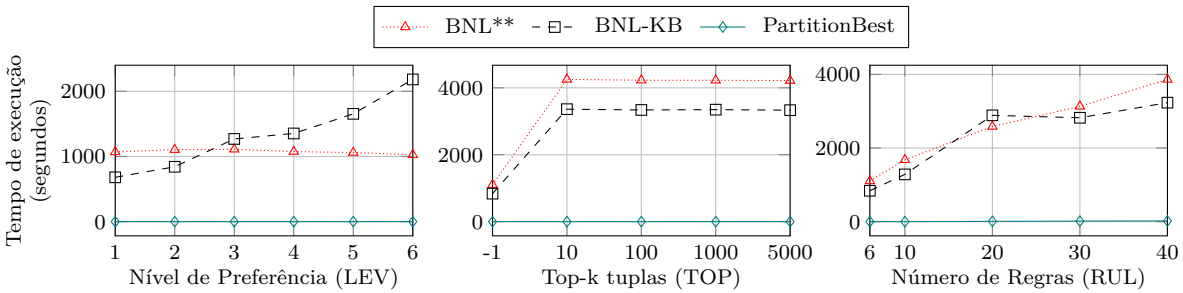


Figura 37 – Experimentos com os algoritmos *BNL***, *BNL-KB* e *PartitionBest* variando os parâmetro LEV, TOP e RUL

Os resultados dos experimentos com variações nos parâmetros LEV, TOP e RUL são mostrados na Figura 37. Considerando a variação no nível de preferência máximo (DB=32, vLEV, RUL=6, QUE=Q5, TOP=-1), o algoritmo *BNL-KB* é mais impactado pelo aumento no nível de preferência máximo. A medida que o nível de preferência máximo aumenta, o tamanho da base de conhecimento também aumenta pelo fato de existirem mais comparações por transitividade. Isto eleva o tempo de execução do algoritmo *BNL-KB*, pois este algoritmo precisa varrer a base de conhecimento para cada par de tuplas a ser comparado (pelo teste de dominância).

Já os algoritmos *BNL*** e *PartitionBest* sofrem pouco impacto com a variação do parâmetro LEV. O teste de dominância baseado em busca do algoritmo *BNL*** possui o custo de $O(m^m)$ apenas no pior caso. Mesmo com o aumento do nível de preferência máximo, na maioria das comparações, não é necessário realizar todas as recursões do teste de dominância baseado em busca. Com isto, o tempo de execução do algoritmo *BNL*** permanece praticamente constante para todos os níveis de preferência máximo testados. No caso do algoritmo *PartitionBest*, o crescimento da base de conhecimento não causa grande aumento no tempo de execução porque a base de conhecimento é varrida uma única vez.

Quanto ao experimento com variação no número de top-k tuplas (DB=32, LEV=2, RUL=6, QUE=Q5, vTOP), os algoritmos apresentam melhor desempenho para TOP=-1. Quando TOP=-1, significa que o operador **BEST** foi usado. Nesta situação, apenas as tuplas dominantes (com nível zero) são retornadas e não há necessidade de processar as tuplas de outros níveis. Novamente, o algoritmo *PartitionBest* apresentou melhor desempenho pelo fato da técnica de particionamento necessitar de menos varreduras no banco de dados.

No experimento com variação no número de regras (DB=32, LEV=2, vRUL, QUE=Q5, TOP=-1), é possível observar que o desempenho dos algoritmos BNL decai a medida que o número de regras aumenta. Quanto maior o número de regras, maiores as chances de duas tuplas serem comparáveis, o que torna o teste de dominância mais lento. Este problema não acontece com o algoritmo *PartitionBest*. Apesar da base de conhecimento se tornar maior com o aumento no número de regras, seu desempenho praticamente não é afetado.

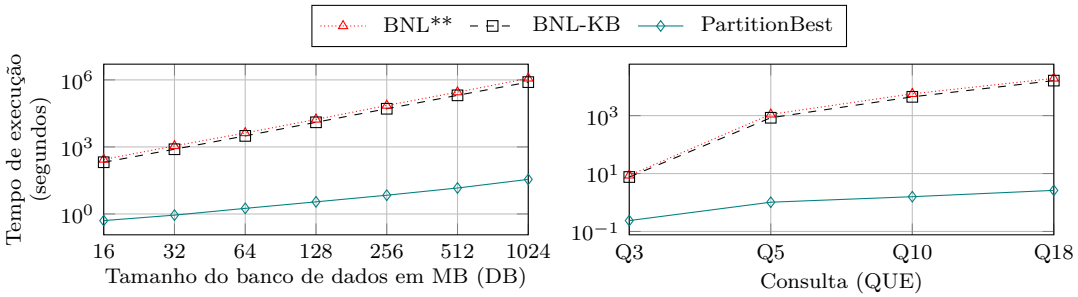


Figura 38 – Experimentos com os algoritmos *BNL***, *BNL-KB* e *PartitionBest* variando os parâmetros DB e QUE

Os gráficos da Figura 38 mostram como os algoritmos escalam de acordo com as variações no tamanho do banco de dados (vDB, LEV=2, RUL=6, QUE=Q5, TOP=-1) e nas consultas (DB=32, LEV=2, RUL=6, vQUE, TOP=-1). O eixo do tempo de execução está em escala logarítmica pelo fato dos algoritmos *BNL*** e *BNL-KB* apresentarem tempos de execuções muito próximos. Estes experimentos evidenciam a diferença de complexidade entre os algoritmos BNL e o algoritmo *PartitionBest*. A complexidade dos algoritmos BNL é quadrática em relação ao número de tuplas. Os bancos de dados maiores possuem mais tuplas, isto causa um impacto maior nestes algoritmos. Por outro lado, a complexidade do algoritmo *PartitionBest* é linear em relação ao número de tuplas. Portanto, o crescimento do tamanho do banco de dados causa menos impacto no tempo de execução do algoritmo *PartitionBest*.

O desempenho dos algoritmos com a variação nas consultas também é afetado pela complexidade. O algoritmo *PartitionBest* sofre pouco impacto com o aumento de tuplas recebidas enquanto os algoritmos BNL apresentam maior tempo de execução a medida que o número de tuplas recebidas aumenta.

8.2 Segunda etapa de experimentos

Esta seção apresenta os experimentos realizados na segunda etapa. Tais experimentos comparam os algoritmos incrementais para processamento de consultas contínuas com preferências condicionais apresentados no Capítulo 4. Além dos algoritmos incrementais, os experimentos levaram em consideração o algoritmo *PartitionBest* para bancos de dados tradicionais descrito no Capítulo 3 como linha base. Como o algoritmo *PartitionBest* tradicional não funciona de forma incremental, ele precisa varrer todas as tuplas para processar a consulta a cada instante. Por outro lado os algoritmos incrementais analisam apenas as remoções e inserções entre o instante anterior e o instante corrente.

Os experimentos executam os algoritmos por várias iterações. A análise dos resultados dos experimentos leva em consideração o tempo gasto em todas as iterações e a quantidade média de memória usada em cada iteração. Os intervalos de confiança dos experimentos são apresentados no Apêndice D.

O restante desta seção está organizada da seguinte maneira: A Seção 8.2.1 descreve os experimentos sobre dados sintéticos. Em seguida, a Seção 8.2.2 apresenta os experimentos sobre dados reais. Por último, a Seção 8.2.3 explica o teste de estresse.

8.2.1 Experimentos com dados sintéticos

Os dados sintéticos usados nos experimentos da primeira etapa foram gerados pelo TPC Benchmark H. Todavia, este *benchmark* não é adequado para gerar dados de ambientes de dados em fluxo. Para possibilitar a execução dos algoritmos, foi desenvolvido um gerador de dados sintéticos capaz de gerar bases de dados típicas de ambientes de dados em fluxo (RIBEIRO, 2017)[4].

O gerador de dados produz relações compostas por atributos inteiros com distribuição normal de valores entre 0 e 63. Além disto, o gerador permite controlar os parâmetros a serem usados nos experimentos. As consultas executadas são projeções simples sobre os atributos da relação gerada contendo as preferências na cláusula `ACCORDING TO PREFERENCES`. As preferências são terias-pc com diferentes números de regras e níveis de preferências. As regras-pc são compostas por regras no formato $Q(A_1) \wedge Q(A_2) \rightarrow Q^+(A_3) \succ Q^-(A3)[W]$ contendo variações nas proposições $Q(A_1)$, $Q(A_2)$, $Q^+(A_3)$, $Q^-(A_3)$ e no conjunto de atributos indiferentes W. O Código 9 mostra a consulta usada nos experimentos com os valores padrões para os parâmetros RUL, LEV e IND.

A Tabela 5 apresenta todos os parâmetros considerados nos experimentos juntamente com suas variações e valores padrões. Os parâmetros ATT, TUP, DEL e INS estão relacionados com a geração dos dados. As variações no número de atributos (ATT) possibilitam a análise do comportamento dos algoritmo em diferentes dimensionalidades.

[4] <http://streampref.github.io/prefgen/>

Código 9 – Consulta para experimentos com os algoritmos *PartitionBest*, *IncAncestorsBest*, *IncGraphBest* e *IncPartitionBest* sobre dados sintéticos com os valores padrões para RUL, LEV e IND

```
SELECT  * FROM r ACCORDING TO PREFERENCES
IF A1 = 1 AND A2 = 1 THEN A3 = 1 BETTER A3 = 2 [A4, A5, A6, A7] AND
IF A1 = 1 AND A2 = 1 THEN A3 = 2 BETTER A3 = 3 [A4, A5, A6, A7] AND
IF A1 = 1 AND A2 = 2 THEN A3 = 1 BETTER A3 = 2 [A4, A5, A6, A7] AND
IF A1 = 1 AND A2 = 2 THEN A3 = 2 BETTER A3 = 3 [A4, A5, A6, A7] AND
IF A1 = 1 AND A2 = 3 THEN A3 = 1 BETTER A3 = 2 [A4, A5, A6, A7] AND
IF A1 = 1 AND A2 = 3 THEN A3 = 2 BETTER A3 = 3 [A4, A5, A6, A7] AND
IF A1 = 1 AND A2 = 4 THEN A3 = 1 BETTER A3 = 2 [A4, A5, A6, A7] AND
IF A1 = 1 AND A2 = 4 THEN A3 = 2 BETTER A3 = 3 [A4, A5, A6, A7];
```

Tabela 5 – Parâmetros dos experimentos com os algoritmos *PartitionBest*, *IncAncestorsBest*, *IncGraphBest* e *IncPartitionBest* sobre dados sintéticos

Parâmetro	**Variação**	**Valor Padrão**
Número de atributos (ATT)	8, 16, 32, 64	8
Número inicial de tuplas (TUP)	500, 1000, 2000, 4000, 8000	1000
Inserções por instante (INS)	50, 100, 200, 400	50
remoções por instante (DEL)	50, 100, 200, 400	50
Número de regras (RUL)	2, 4, 8, 16, 32	8
Nível de preferência máximo (LEV)	1, 2, 4, 8	2
Número de atributos indiferentes (IND)	0, 1, 2, 4	4

O parâmetro TUP é o número de tuplas no instante inicial, antes de qualquer alteração. Os experimentos com variação no número de tuplas permitem avaliar a escalabilidade dos algoritmos considerando diferentes volumes de dados. Os parâmetros DEL e INS representam respectivamente o número de tuplas removidas por instante e o número de tuplas inseridas por instante. As variações destes parâmetros possibilitam analisar o comportamento dos algoritmos considerando diferentes tamanhos para os conjuntos Δ^+ e Δ^-.

Os parâmetros LEV, RUL e IND dizem respeito ao formato das preferências. Eles são importantes para analisar o impacto de diferentes configurações de preferências sobre os algoritmos uma vez que o formato das preferências também influencia na performance dos algoritmos. Para cada experimento executado, um parâmetro sofre variação e os demais são mantidos em seus valores padrões. Em todos os experimentos, foram executadas 100 iterações dos algoritmos.

8.2.1.1 Variação nos parâmetros da geração dos dados

A Figura 39 exibe o resultados dos experimentos com variação nos parâmetros ATT e TUP. No experimento com variação no número atributos (vATT, TUP=1000, DEL=50, INS=50, RUL=8, LEV=2, IND=4), o algoritmo *IncPartitionBest* obteve a melhor per-

formance, seguido pelo algoritmo *PartitionBest*. Entretanto, para ATT=64, o algoritmo *PartitionBest* tem um tempo de execução similar aos algoritmos *IncAncestorsBest* e *IncGraphBest*.

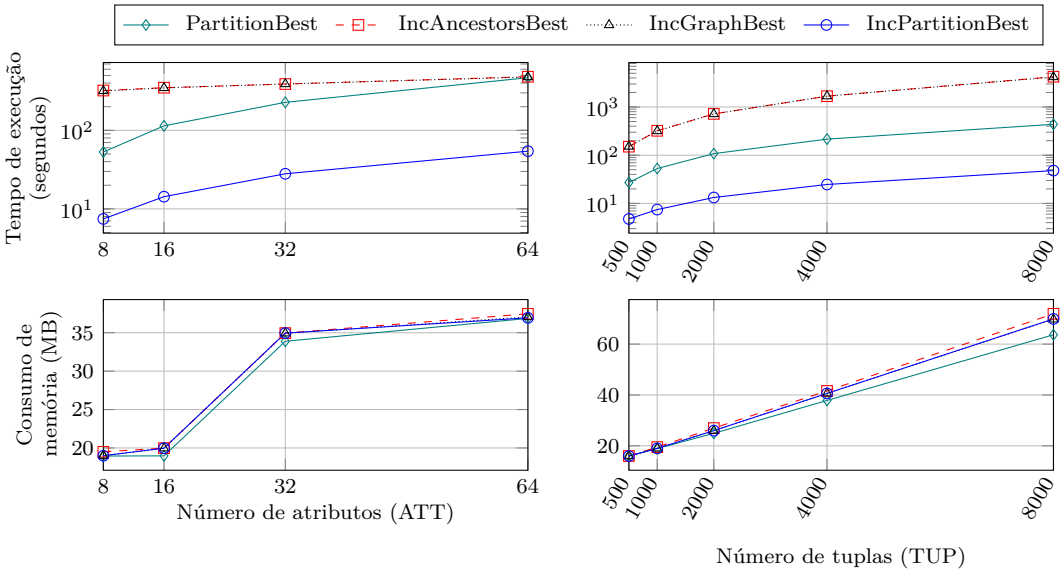


Figura 39 – Experimentos com os algoritmos *PartitionBest*, *IncAncestorsBest*, *IncGraphBest* e *IncPartitionBest* sobre dados sintéticos variando os parâmetros ATT, TUP, INS e DEL

Conforme explicado no Capítulo 3, números maiores de atributos causam um aumento significativo da base de conhecimento fazendo com que o algoritmo *PartitionBest*, que precisa varrer todas as tuplas a cada instante, tenha este alto tempo de execução. Já os algoritmos *IncAncestorsBest* e *IncGraphBest*, que analisam apenas as inserções e exclusões, são menos afetados. O algoritmo *IncPartitionBest* também faz uso da base de conhecimento, mas analisa apenas as tuplas inseridas e removidas e, com isso, consegue a melhor performance.

Analisando a variação no número de tuplas (ATT=8, vTUP, DEL=50, INS=50, RUL=8, LEV=2, IND=4), o algoritmo *IncPartitionBest* possui o menor tempo de execução, o algoritmo *PartitionBest* conseguiu o segundo menor tempo, os algoritmos *IncAncestorsBest* e *IncGraphBest* tiveram o maior tempo de execução. Observe que mesmo com o aumento do número de tuplas iniciais, a distância entre os tempos de execução dos algoritmos praticamente não é alterada. As hierarquias de preferência usadas pelos algoritmos *IncAncestorsBest* e *IncGraphBest* armazenam informações individuais sobre cada tupla. Com o aumento do número de tuplas, as chances de comparação aumentam, e estas hierarquias de preferências se tornam maiores e seu processo de atualização se torna mais caro.

Analisando o consumo de memória, em todos os cenários, os algoritmos incrementais possuem um comportamento muito similar. Isto indica que as estruturas de hierarquias de preferência usam uma quantidade de memória semelhante. O algoritmo *PartitionBest*

possui o menor consumo de memória pelo fato de não armazenar uma hierarquia de preferência em memória.

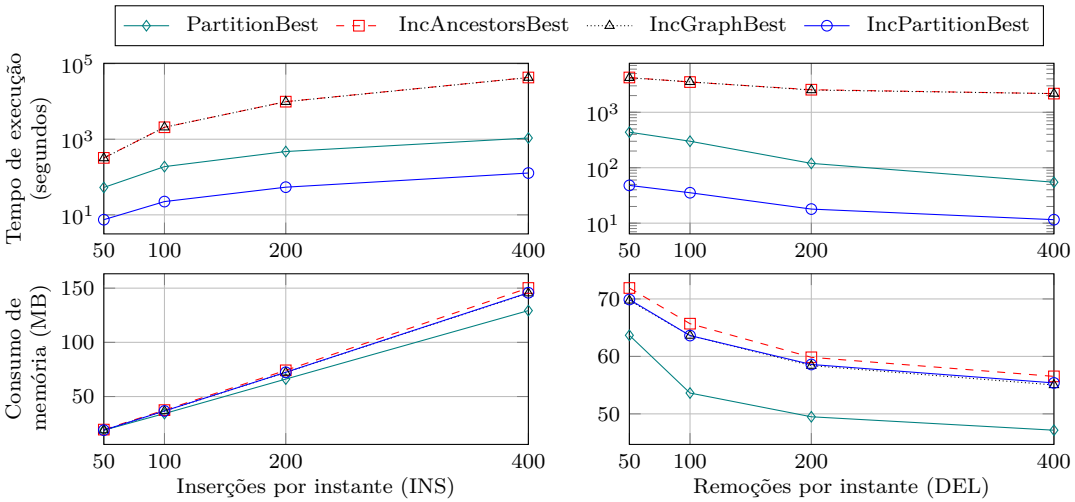


Figura 40 – Experimentos com os algoritmos *PartitionBest*, *IncAncestorsBest*, *IncGraphBest* e *IncPartitionBest* sobre dados sintéticos variando os parâmetros INS e DEL

A Figura 40 exibe o resultado dos experimentos com variação nos parâmetros INS e DEL. A variação no número de inserções por instante (ATT=8, TUP=1000, DEL=50, vINS, RUL=8, LEV=2, IND=4) faz com que os algoritmos gastem mais tempo a medida que o número de inserções aumenta. O tempo de execução maior fica com os algoritmos *IncAncestorsBest* e *IncGraphBest* porque eles comparam cada tupla inserida com todas as demais tuplas da relação. Como estes algoritmos usam o teste de dominância baseado em busca, é gasto um tempo maior na tarefa de inserir. Os algoritmos *PartitionBest* e *IncPartitionBest* possuem tempos de execução menores pelo fato de usarem a técnica de particionamento. Sendo que, o *IncPartitionBest* leva vantagem porque o algoritmo *PartitionBest* precisa comparar todas as tuplas da relação a cada instante e o *IncPartitionBest* analisa apenas as tuplas inseridas.

Para analisar a variação no número de remoções por instante, o número inicial de tuplas foi fixado em 8000 (ATT=8, TUP=8000, vDEL, INS=50, RUL=8, LEV=2, IND=4). Evitando assim que a relação ficasse vazia mais rapidamente. Os algoritmos têm um comportamento oposto àquele apresentado no experimento com variação no parâmetro INS, à medida que as remoções aumentam, os algoritmos gastam menos tempo para executar. Os algoritmos *PartitionBest* e *IncPartitionBest* apresentam o menor tempo de execução novamente. As hierarquias de preferências usadas por estes algoritmos armazenam relações de dominância individuais entre as tuplas. Assim, a remoção de uma única tupla t causa a alteração da hierarquia de preferência para todas as tuplas dominadas por t. A hierarquia de preferência do algoritmo *IncPartitionBest* é atualizada mais rapidamente por ser mais compacta e armazenar relações de dominância entra grupos de tuplas. O algoritmo *PartitionBest* não trabalha de forma incremental, mas seu tempo de execução

para comparar todas as tuplas a cada instante chega a ser menor do que o tempo gasto pelos algoritmos *PartitionBest* e *IncPartitionBest* para atualizar suas hierarquias de preferência.

Quanto ao consumo de memória, os algoritmos incrementais novamente apresentaram um consumo semelhante. Mais uma vez, o algoritmo *PartitionBest* consumiu menos memória porque o mesmo não armazena hierarquias de preferência como os demais algoritmos.

8.2.1.2 Variação nos parâmetros das preferências

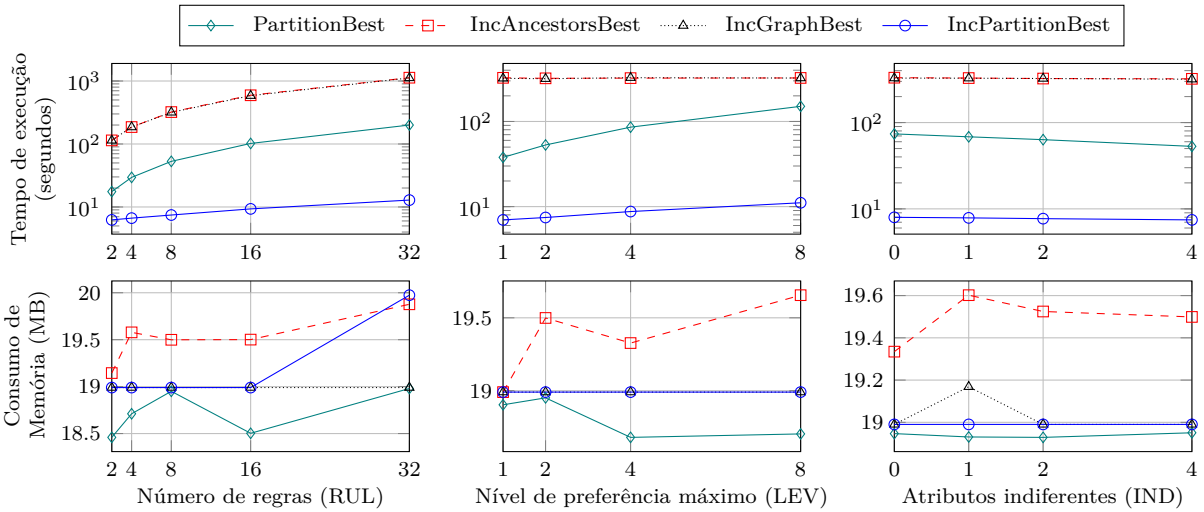


Figura 41 – Experimentos com os algoritmos *PartitionBest*, *IncAncestorsBest*, *IncGraphBest* e *IncPartitionBest* sobre dados sintéticos variando os parâmetros RUL, LEV e IND

A Figura 41 apresenta o tempo de execução dos algoritmos com variação nos parâmetros das preferências (RUL, LEV e IND). Com respeito à variação do número de regras (ATT=8, TUP=1000, DEL=50, INS=50, vRUL, LEV=2, IND=4), o aumento no número de regras tem grande impacto sobre o desempenho dos algoritmos *IncAncestorsBest* e *IncGraphBest*. Números maiores de regras tornam o teste de dominância baseado em busca usado por estes algoritmos significativamente mais lento. O número de regras também influencia o tamanho da base de conhecimento utilizada pelos algoritmos *PartitionBest* e *IncPartitionBest*, mas em uma escala muito menor. A melhor performance do algoritmo *IncPartitionBest* em relação ao algoritmo *PartitionBest* ocorre por que o primeiro é incremental e não precisa processar todas as tuplas a cada instante.

A variação no nível de preferência máximo (ATT=8, TUP=1000, DEL=50, INS=50, RUL=8, vLEV, IND=4) possui maior impacto nos algoritmos *PartitionBest* e *IncPartitionBest* e menor impacto nos algoritmos *IncAncestorsBest* e *IncGraphBest*. A complexidade do teste de dominância baseado em busca dos algoritmos *IncAncestorsBest* e *IncGraphBest* é $O(m^m)$ apenas no pior caso. Na prática, o teste termina antes que todas as ramificações da árvore de busca sejam atingidas. Contudo, estes algoritmos apresentam

o pior tempo de execução em todas as variações por causa do tipo de hierarquia de preferência utilizado que armazena as relações de dominância individuais entre as tuplas. O comportamento dos algoritmos *PartitionBest* e *IncPartitionBest* é muito similar, porém, o método incremental do algoritmo *IncPartitionBest* lhe garante o melhor desempenho.

A variação no número de atributos indiferentes (ATT=8, TUP=1000, DEL=50, INS=50, RUL=8, LEV=2, vIND) não causa grandes variações no tempo de execução dos algoritmos. Os algoritmos *PartitionBest* e *IncPartitionBest* sofrem uma pequena influência porque maiores números de atributos indiferentes proporcionam comparações essenciais mais genéricas que levam a bases de conhecimento menores. Os motivos para a diferença de desempenho entre todos os algoritmos são novamente o tipo de hierarquia de preferência e os métodos incrementais já explicados.

A Figura 41 exibe também o consumo de memória dos algoritmos para as variações nos parâmetros RUL, LEV e IND. As variações destes parâmetros não causaram grandes oscilações no consumo de memória. Em todas as execuções a memória usada pelos algoritmos ficou entre 18 e 20 megabytes. O algoritmo *PartitionBest* consumiu menos memória porque não armazena nenhuma estrutura de hierarquia de preferência. Contudo, os demais algoritmos, mesmo com manutenção das hierarquias de preferência, não tiveram um consumo de memória muito superior.

8.2.1.3 Tempo de execução por iteração

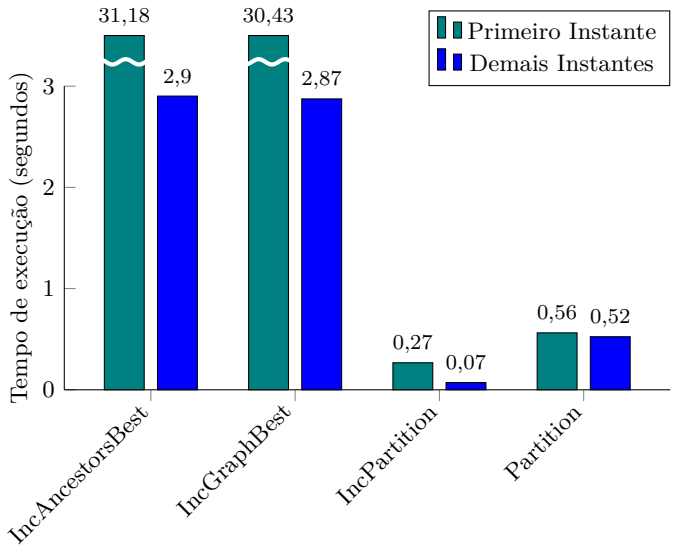


Figura 42 – Tempo de execução por instante dos algoritmos *PartitionBest*, *IncAncestorsBest*, *IncGraphBest* e *IncPartitionBest* sobre dados sintéticos

Em ambientes da dados em fluxos, é importante verificar o tempo de execução por iteração dos algoritmos, uma vez que este tipo de cenário requer um tempo de resposta rápido em qualquer instante. A Figura 42 mostra o tempo de execução dos algoritmos no

primeiro e demais instantes considerando os valores padrões para os todos os parâmetros (ATT=8, TUP=1000, DEL=50, INS=50, RUL=8, LEV=2, IND=4).

Os algoritmos *IncAncestorsBest* e *IncGraphBest* possuem um comportamento muito similar, ambos gastam um tempo maior na primeira iteração e um tempo menor nas demais iterações. O maior tempo gasto na primeira iteração é causado pela atualização das estruturas de hierarquia de preferência de todas as tuplas existentes. Já nas demais iterações, os algoritmos precisam apenas atualizar tais estruturas de acordo com as remoções e inserções.

O algoritmo *PartitionBest* apresenta um tempo de execução similar na primeira e demais iterações porque em todas as iterações o algoritmo precisa processar todas as tuplas. O algoritmo *IncPartitionBest* obteve melhor desempenho em relação aos demais tanto na primeira quanto nas demais iterações. Este melhor desempenho se deve às estruturas de dados mais compactas usadas para manter a hierarquia de preferência.

8.2.2 Experimentos com dados reais

Os experimentos com dados reais usaram dados históricos do mercado de ações importados do Yahoo Finance (Yahoo! Inc., 2015)[5]. Como foi mostrado do Capítulo 4, esta é um interessante aplicação prática onde as consultas com preferência condicionais podem ser usadas. As consultas dos experimentos utilizaram as mesmas preferências do trabalho de Petit et al. (2012). Estas preferências incluem proposições sobre um atributo de taxa de volatilidade calculada sobre os valores das ações. Para cada tupla importada, foram calculadas duas taxas de volatilidade usando dois métodos distintos. Como resultado foi criado o fluxo de dados `transactions` composto pelos seguintes atributos:

`symbol:` código identificador da ação;
`sector:` setor da empresa;
`country:` país da sede da empresa;
`price:` preço da ação;
`volume:` quantidade de transações com a ação;
`rate:` taxa de volatilidade calculada;
`method:` método usado no cálculo da volatilidade.

A base de dados original possui 36.661 tuplas da bolsa de valores de São Paulo entre os anos de 2014 e 2015. Após o cálculo da volatilidade, o fluxo de dados resultante ficou com 73.322 tuplas. A base de dados possui ações de 140 empresas com aproximadamente 148 tuplas por instante de tempo. Foi desenvolvida uma ferramenta para automatizar o processo de extração e cálculo de volatilidade da base de dados (RIBEIRO, 2017)[6].

[5] <http://finance.yahoo.com/>
[6] <http://streampref.github.io/yfimport/>

Os experimentos seguiram a mesma metodologia adotada no trabalho de Petit et al. (2012). No referido trabalho, os autores usaram janelas deslizantes baseadas em contagem. Os experimentos descritos nesta seção usaram janelas baseadas em tempo uma vez que estes operadores já haviam sido implementados. Contudo, considerando a quantidade de tuplas por instante, ambos experimentos possuem um tamanho de janela similar (em número de tuplas).

O Código 10 mostra o formato das consultas processadas durante a execução dos experimentos. As consultas possuem variações na abrangência temporal (RAN) e intervalo de deslocamento (SLI). O parâmetro RAN variou de 2 a 6 com valor padrão de 4. A variação no parâmetro SLI foi de 1 a 4 com valor padrão de 1.

Código 10 – Consulta para experimentos com os algoritmos *PartitionBest*, *IncAncestorsBest*, *IncGraphBest* e *IncPartitionBest* sobre dados reais

```
SELECT * FROM stocks [RANGE <RAN > SECOND, SLIDE <SLI > SECOND]
ACCORDING TO PREFERENCES
  IF sector = 'Basic Materials'
    THEN rate < 0.25 BETTER rate >= 0.25 [method, symbol, price] AND
  IF sector = 'Technology'
    THEN rate < 0.35 BETTER rate >= 0.35 [method, symbol, price] AND
  IF rate >= 0.35
    THEN country = 'Brazil' BETTER country = 'France' [symbol, price] AND
  IF rate >= 0.35
    THEN volume > 1000 BETTER volume <= 1000 [symbol, price];
```

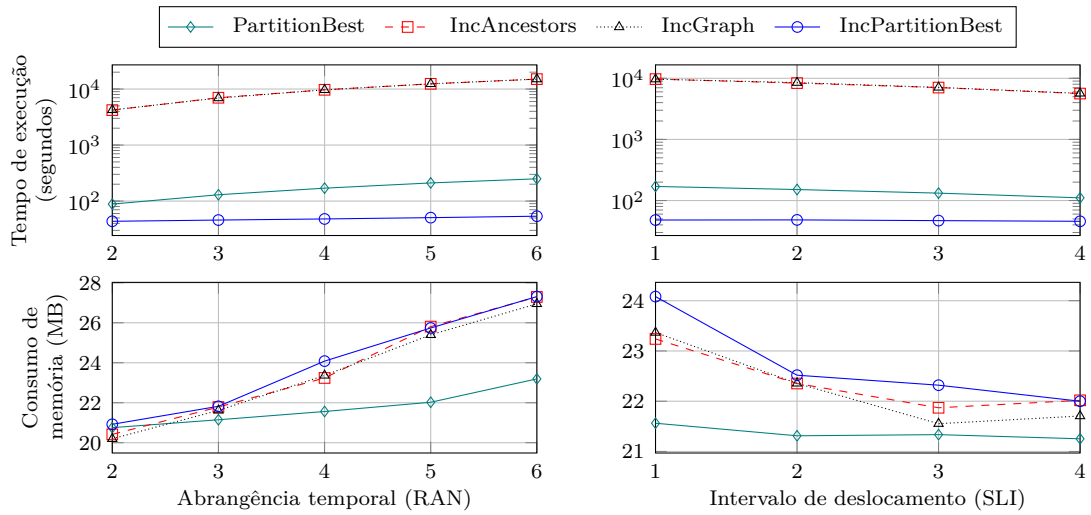


Figura 43 – Experimentos com os algoritmos *PartitionBest*, *IncAncestorsBest*, *IncGraphBest* e *IncPartitionBest* sobre dados reais

A Figura 43 exibe o tempo de execução e o consumo de memória dos algoritmos sobre os dados reais. Tanto no experimento com variação na abrangência temporal quanto no experimento com variação do intervalo de deslocamento, os algoritmos *IncAncestorsBest* e *IncGraphBest* possuem o maior tempo de execução. Mesmo processando apenas as inserções e exclusões, os algoritmos *IncAncestorsBest* e *IncGraphBest* gastam mais tempo

por causa da complexidade do teste de dominância baseado em busca e das hierarquias de preferência que armazenam relações de dominância individuais para as tuplas.

O algoritmo *PartitionBest*, mesmo não sendo um algoritmo incremental, supera os algoritmos *IncAncestorsBest* e *IncGraphBest* com o uso da técnica de particionamento. Já o algoritmo *IncPartitionBest* tem a melhor performance porque, além de ser um algoritmo incremental, utiliza uma hierarquia de preferência baseada na técnica de particionamento.

Comparando a variação na abrangência temporal com a variação no intervalo de deslocamento, os algoritmos têm comportamentos opostos. As abrangências temporais maiores aumentam o tamanho da janela e, por consequência, o tempo de execução dos algoritmos. Por outro lado, os intervalos de deslocamento maiores causam a exclusão de um maior número de tuplas das janelas, fazendo com que o tamanho da janela seja reduzido.

Quanto ao consumo de memória, todos os algoritmos apresentam um comportamento muito similar. Existe um consumo um pouco maior para as abrangências temporais maiores e os intervalos de deslocamentos menores, situações nas quais as janelas possuem um tamanho maior. Novamente, o algoritmo *PartitionBest* obteve o menor consumo de memória pelo fato de não ter que armazenar uma hierarquia de preferência.

8.2.3 Teste de estresse

Além dos experimentos comparativos entre os algoritmos, foi realizado também um teste de estresse com o algoritmo *IncPartitionBest* utilizando bases de dados sintéticas. O objetivo do teste de estresse é traçar os limites para o algoritmo *IncPartitionBest* em relação ao número de tuplas inseridas (INS) e ao número de regras (RUL). Este parâmetros foram escolhidos porque eles estão diretamente relacionados com a complexidade do algoritmo *PartitionInsert* que deve ser executado a cada instante para atualizar a hierarquia de preferências. A variação usada no parâmetro INS foi 500, 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000, 256000 e 512000. Já o parâmetro RUL teve os valores 8, 16, 32, 64, 128, 256, 512, 1024, 2048 e 4096.

A Figura 44 mostra os resultados do teste de estresse. O número de inserções igual a 512000 não aparece na plotagem porque esta configuração de experimentos não foi suportada pelo algoritmo no computador descrito no início deste capítulo. Observe que para INS=256000 o consumo de memória chegou a mais de 15GB. Com o número de inserções igual a 512000, a memória disponível (32GB) não foi suficiente para a execução do algoritmo.

Considerando a variação no número de regras, mesmo com o valor máximo da variação (4096), o consumo de memória ficou longe do total de memória disponível. Entretanto, a limitação para este parâmetro é o tempo de execução que chegou a mais de 11 dias para RUL=4096. No gráfico que mostra o tempo de execução para a variação do parâmetro RUL também foram incluídos o tempo para criar a base de conhecimento e o

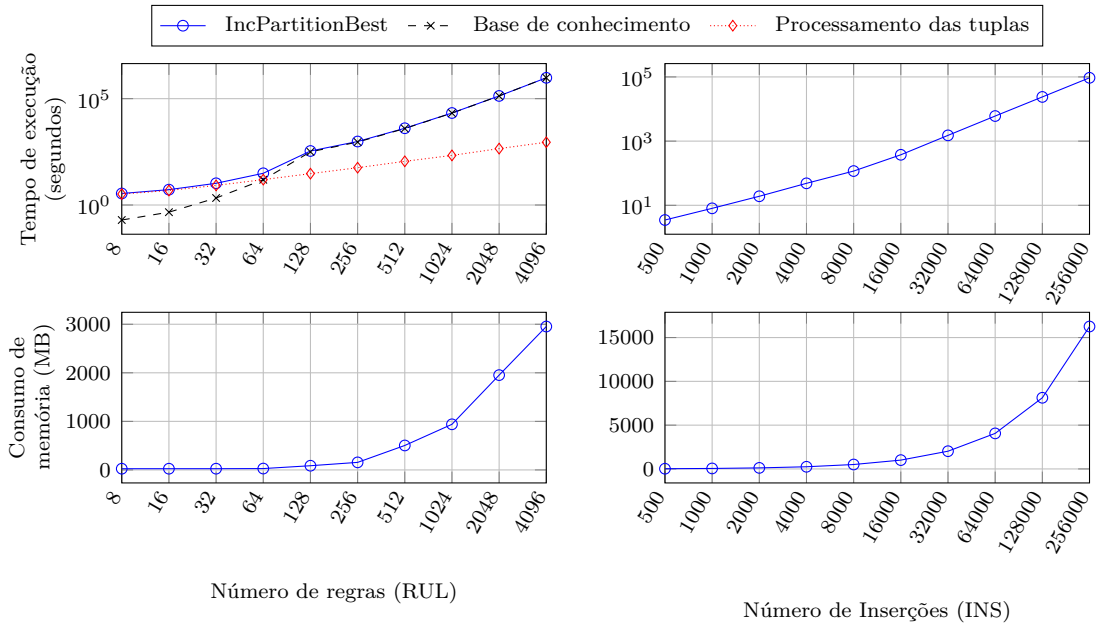


Figura 44 – Teste de estresse com o algoritmo *IncPartitionBest*

tempo gasto para processar as tuplas. A partir de 64 regras, o tempo gasto para criar a base de conhecimento passa a ser maior do que o tempo gasto para processar as tuplas. Contudo, em situações práticas, não é muito comum a especificação de preferências com grandes número de regras pelos usuários. Além disto, mesmo tendo gasto um tempo maior para construção da base de conhecimento, o algoritmo *IncPartitionBest* ainda é a melhor opção para o processamento de consultas contínuas com preferências condicionais.

8.3 Terceira etapa de experimentos

No Capítulo 6 foram apresentados os operadores da linguagem StreamPref juntamente com suas respectivas operações equivalentes na linguagem CQL. Foram executados experimentos para comparar o desempenho dos algoritmos de cada operador StreamPref com sua contraparte em CQL. Além disto, foram conduzidos experimentos para demonstrar como as possíveis combinações de operadores em consultas podem afetar a quantidade de sequências comparadas.

Bases de dados

Os experimentos usaram dados sintéticos e reais. Devido a inexistência de geradores de dados adequados aos parâmetros usados nos experimentos, foi desenvolvido um gerador de dados específico (RIBEIRO, 2017)[7]. O gerador desenvolvido gera fluxos de dados

[7] <http://streampref.github.io/streamprefgen/>

compostos por atributos inteiros e também as consultas StreamPref e CQL usadas nos experimentos.

Para os experimentos com dados reais, foi utilizada a base de dados *Soccer2014DS* contendo eventos de jogadores nas partidas da Copa do Mundo de Futebol de 2014 (RIBEIRO et al., 2017b)[8]. Os experimentos foram conduzidos sobre os dados da partida final com 2.742 tuplas distribuídas em 7.722 instantes de tempo.

Parâmetros

Os experimentos da terceira etapa também consideraram certos parâmetros com variações de valores. A Tabela 6 exibe todos os parâmetros usados. Certos parâmetros são específicos de alguns operadores que possuem características específicas para serem avaliadas

Tabela 6 – Parâmetros dos experimentos com os operadores da linguagem StreamPref

Parâmetro	**Variação**	**Valor Padrão**
Número de atributos (ATT)	8, 10, 12, 14, 16	8
Número de sequências (NSQ)	4, 8, 16, 24, 32	16
Porcentagem de tuplas consecutivas (PCT)	0%, 25%, 50%, 75% e 100%	50%
Abrangência temporal (RAN)	10, 20, 30, 40, 50, 60	30
Intervalo de deslocamento (SLI)	1, 10, 20, 30, 40	10
Comprimento mínimo (MIN)	2, 3, 4, 5, 6	2
Comprimento máximo (MAX)	2, 4, 6, 8, 10	10
Número de regras (RUL)	4, 8, 16, 24, 32	8
Nível de preferência máximo (LEV)	1, 2, 3, 4, 5, 6	2

Os parâmetros ATT, NSQ e PCT dizem respeito à geração dos dados sintéticos, sendo que a porcentagem de tuplas consecutivas (PCT) é considerada apenas nos experimentos com o operador **CONSEQ**. O parâmetro ATT permite comparar os algoritmos de acordo com a dimensionalidade dos dados. As variações no parâmetro NSQ afetam o número de identificadores de sequência por instante (igual a NSQ $\times 0.75$). Com as variações dos parâmetros RAN e SLI é possível analisar o comportamento dos algoritmos considerando diferentes abrangências temporais e intervalos de deslocamento.

Os parâmetros MIN e MAX são usados pelos operadores **MINSEQ** e **MAXSEQ**, respectivamente, para realizar a filtragem de sequências por comprimento. O número de regras (LEV) e o nível de preferência máximo (LEV) dizem respeito às preferências das consultas processadas pelo operador **BESTSEQ**. As variações nos valores destes parâmetros possibilitam analisar o funcionamento dos algoritmos usando diferentes configurações de preferências.

[8] <http://streampref.github.io/wcimport/>

Para cada experimento, ocorre a variação nos valores de um parâmetro e os demais parâmetros são fixados em seus valores padrões. O número de iterações nos experimentos sobre dados sintéticos é igual ao valor do parâmetro RAN acrescido do maior valor de intervalo de deslocamento. No caso dos dados reais, o número de iterações é igual ao número de instantes de tempo disponíveis (7.722). A análise dos resultados dos experimentos leva em consideração o tempo gasto em todas as iterações e a quantidade média de memória usada em todas as iterações. Os intervalos de confiança dos experimentos são apresentados no Apêndice E.

O restante desta seção está organizado como se segue. Da Seção 8.3.1 até a Seção 8.3.6 são apresentados os experimentos comparativos entre os algoritmos e as equivalências CQL usados para processar os operadores da linguagem StreamPref. Em seguida, a Seção 8.3.7 apresenta os experimentos para analisar como as combinações de operadores influenciam na comparação de sequências.

8.3.1 Experimentos com o operador SEQ

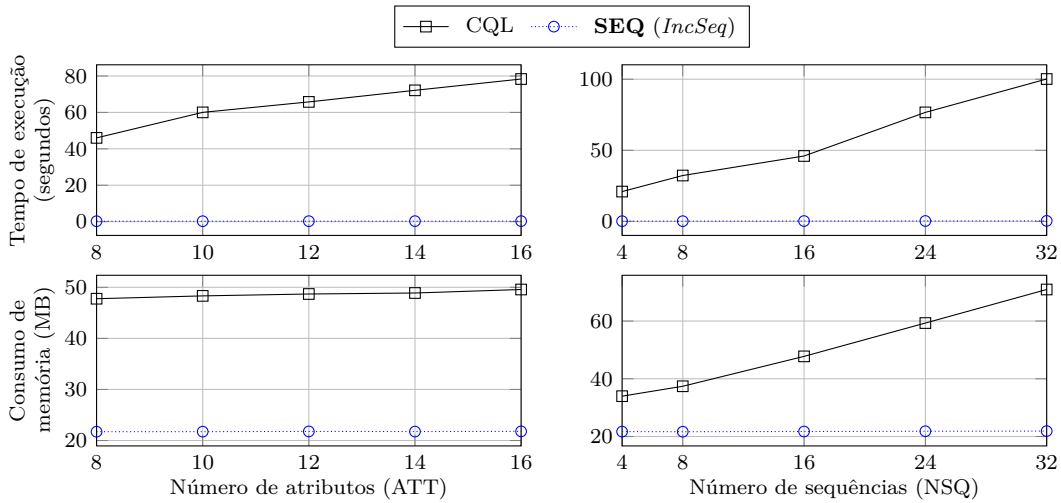


Figura 45 – Experimentos com o operador **SEQ** sobre dados sintéticos variando os parâmetros ATT e NSQ

A Figura 45 exibe os resultados dos experimentos sobre dados sintéticos com o operador **SEQ** variando os parâmetros ATT e NSQ. Os resultados dos experimentos mostram que, mesmo para o menor número de atributos, o algoritmo *IncSeq* possui melhor desempenho e menor consumo de memória do que sua equivalência em CQL. O número de atributos afeta as operações de projeção e união da equivalência CQL e o número de sequências impacta em todas as operações intermediárias da equivalência causando esta diferença de desempenho.

A Figura 46 mostra o resultado dos experimentos sobre dados sintéticos variando os parâmetros RAN e SLI. O aumento da abrangência temporal (RAN) causa um grande impacto no tempo de execução e consumo de memória da equivalência CQL. Isto ocorre

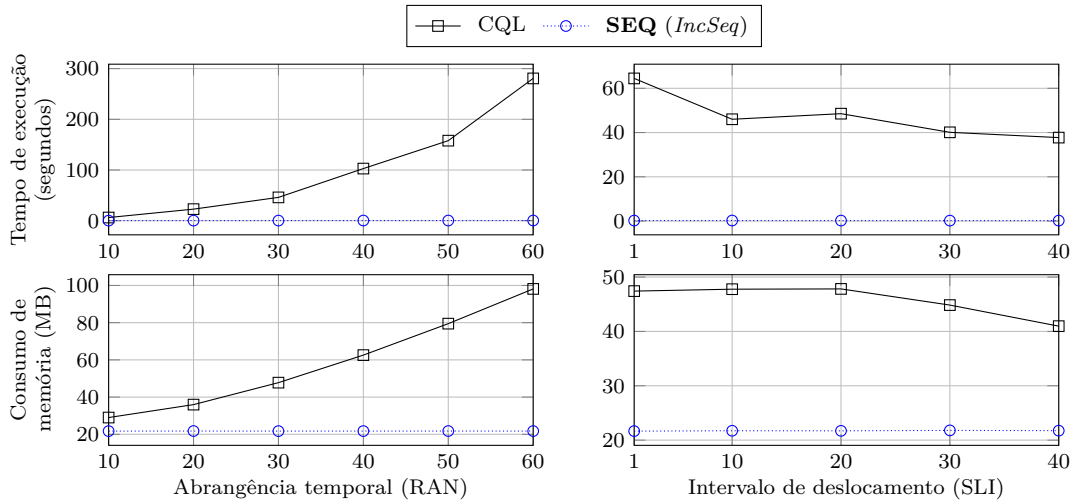


Figura 46 – Experimentos com o operador **SEQ** sobre dados sintéticos variando os parâmetros RAN e SLI

porque, para cada segundo da abrangência temporal, a equivalência CQL precisa computar uma relação W_i e uma relação P_i através de operações de diferença de conjunto e de junção. Estas operações elevam o tempo de execução ao mesmo tempo que o armazenamento das relações adicionais consome mais memória.

A variação do intervalo de deslocamento (SLI) causa menos impacto na equivalência CQL. Contudo, para intervalos maiores, as tuplas antigas expiram com mais frequência. Com isto, o processamento de uma quantidade menor de tuplas consome menos tempo e menos memória. O algoritmo *IncSeq*, por ser um algoritmo incremental, consegue um desempenho superior e consome menos memória do que a equivalência CQL.

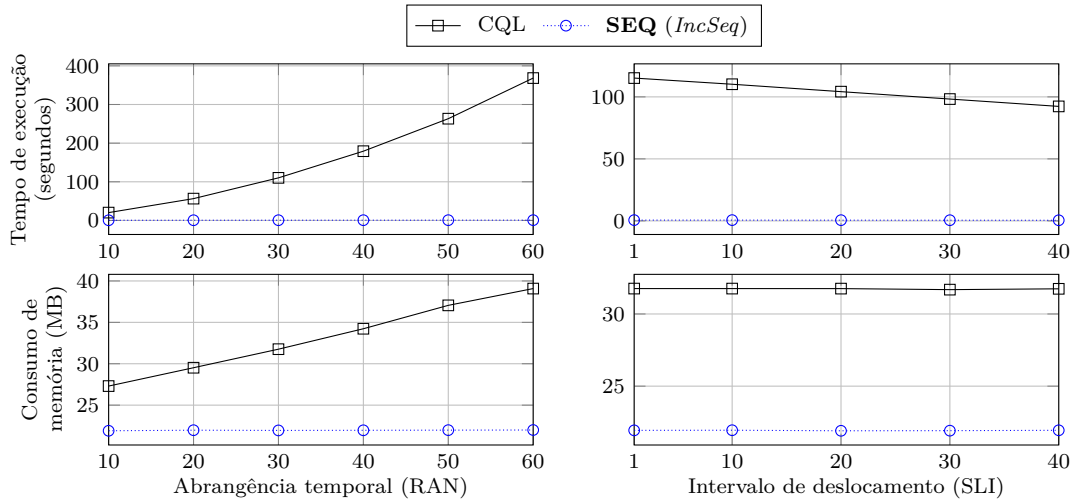


Figura 47 – Experimentos com o operador **SEQ** sobre dados reais variando os parâmetros RAN e SLI

A Figura 47 apresenta o resultado dos experimentos com dados reais variando os parâmetros RAN e SLI. O resultado é similar ao resultado do mesmo experimento com

dados sintéticos. Novamente, a equivalência CQL foi superada pelo algoritmo **SEQ** devido às operações intermediárias e relações temporárias em CQL.

8.3.2 Experimentos com o operador CONSEQ

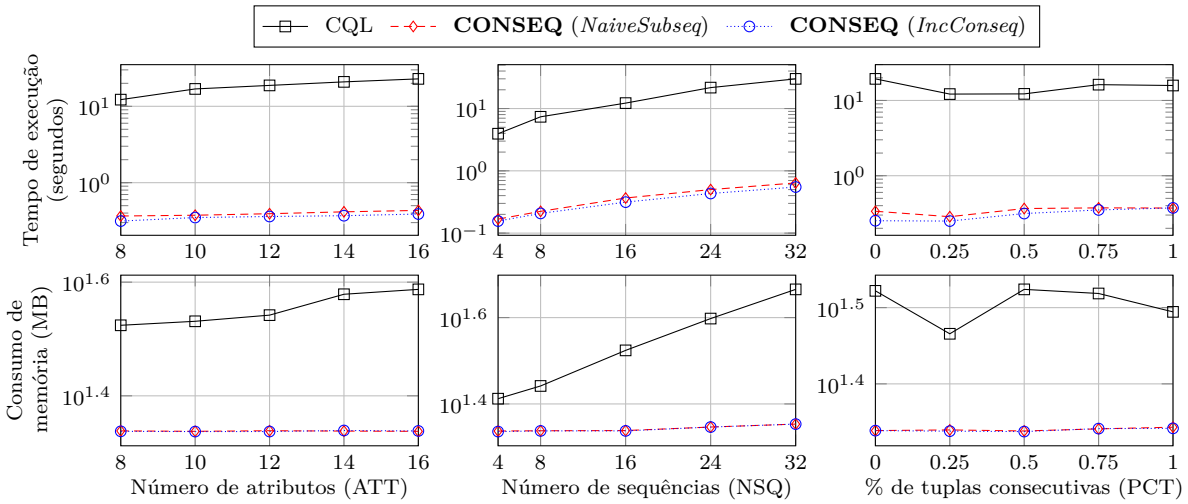


Figura 48 – Experimentos com o operador **CONSEQ** sobre dados sintéticos variando os parâmetros ATT, NSQ, PCT

Os experimentos com o operador **CONSEQ** consideram os algoritmos ingênuo (*NaiveSubseq*) e incremental (*IncConseq*) para processar o operador bem como sua equivalência CQL. A Figura 48 mostra o resultado dos experimentos dobre dados sintéticos variando os parâmetros ATT, NSQ e PCT. Em todas as variações, os algoritmos *NaiveSubseq* e *IncConseq* apresentam um comportamento parecido levando considerável vantagem sobre a equivalência CQL. O algoritmo incremental possui uma pequena vantagem em algumas situações porque não precisa processar todas as tuplas das sequências a cada instante.

Os parâmetros ATT e PCT causam pouco impacto no comportamento dos algoritmos. Já a variação do número de sequências (NSQ) provoca um notável impacto no tempo de execução e no consumo de memória. Independente do uso dos algoritmos ou equivalência CQL, todas as sequências precisam ser processadas pelo menos uma vez a cada instante. As operações intermediárias e as relações temporárias fazem com que a equivalência CQL tenha o pior desempenho e o maior consumo de memória.

As Figuras 49 e 50 apresentam o resultado dos experimentos variando os parâmetros RAN e SLI. Sendo que, a Figura 49 exibe o resultado dos experimentos sobre dados sintéticos e a Figura 50 exibe o resultado dos experimentos sobre dados reais. Os algoritmos e a equivalência CQL apresentam um comportamento muito similar. Mais uma vez, os algoritmos desenvolvidos se sobressaem em relação à equivalência CQL. Os valores maiores para o parâmetro RAN provocam a extração de sequências maiores e, por consequência, mais tuplas precisam ser processadas. Os intervalos de deslocamentos maiores causam um efeito contrário. Mais tuplas expiram e as sequências extraídas se tornam menores.

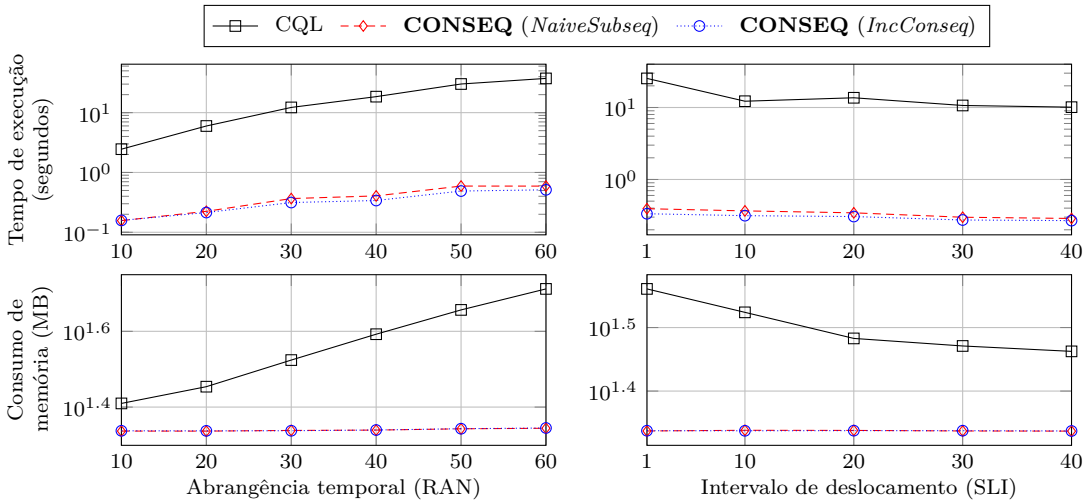


Figura 49 – Experimentos com o operador **CONSEQ** sobre dados sintéticos variando os parâmetros RAN e SLI

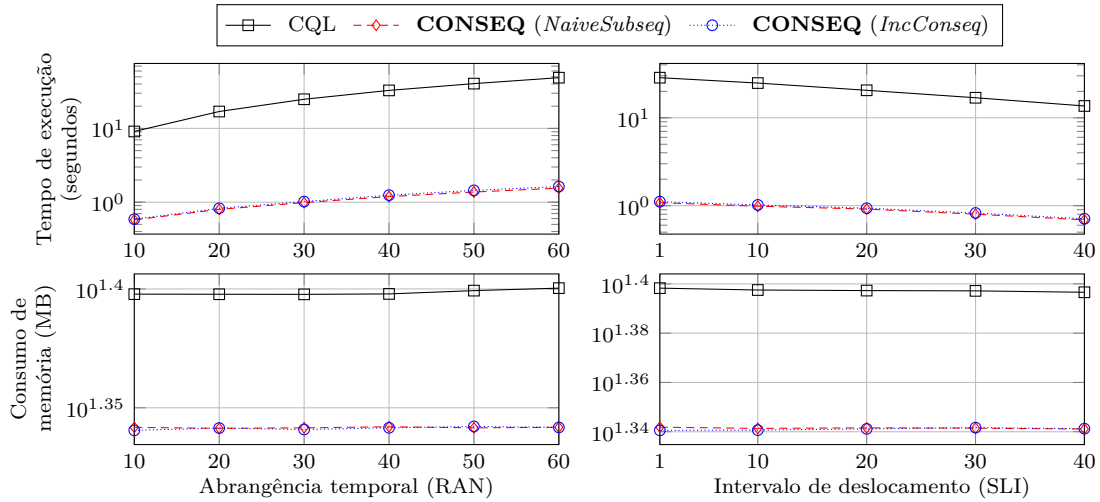


Figura 50 – Experimentos com o operador **CONSEQ** sobre dados reais variando os parâmetros RAN e SLI

Sendo assim, o processamento de um número maior de tuplas faz com que os algoritmos e a equivalência CQL consumam mais tempo de execução e mais memória. O desempenho inferior e o consumo de memória maior da equivalência CQL são causados pelas operações intermediárias e relações temporárias que se tornam mais dispendiosas na presença de um número maior de tuplas.

8.3.3 Experimentos com o operador ENDSEQ

Os experimentos com o operador **ENDSEQ** comparam a equivalência CQL com os algoritmos *NaiveSubseq* e *IncEndseq* apresentados no Capítulo 7. Os primeiros experimentos foram realizados sobre dados sintéticos e consideraram as variações dos parâmetros ATT e NSQ. A Figura 51 mostra o resultado destes experimentos. A variação no número de atributos (ATT) causa pouco impacto no comportamento dos algoritmos e da equivalência

CQL. Por outro lado, os números de sequências maiores afetam consideravelmente o tempo de execução e o consumo de memória. Uma quantidade maior de sequências causa um tempo de execução e um consumo de memória maiores tanto nos algoritmos quando na equivalência CQL.

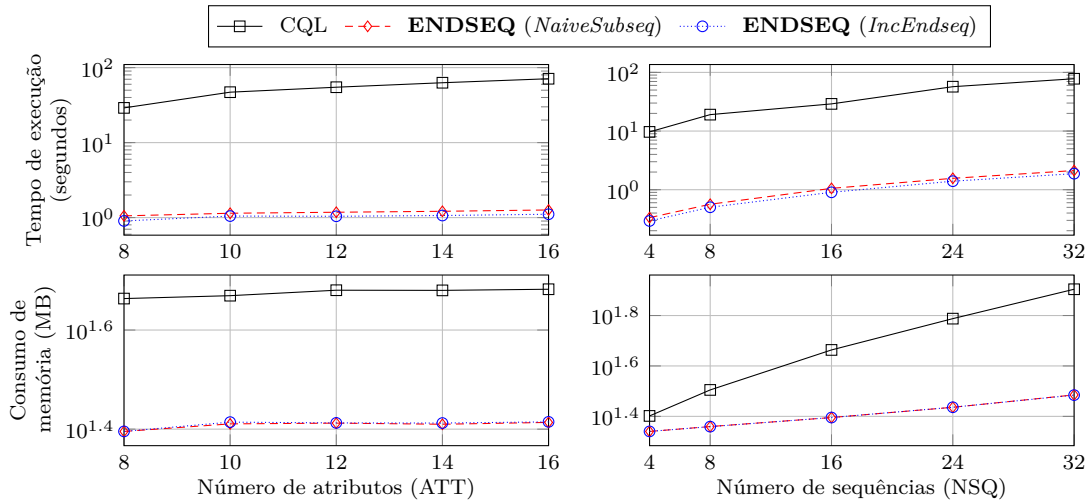


Figura 51 – Experimentos com o operador **ENDSEQ** sobre dados sintéticos variando os parâmetros ATT e NSQ

A Figura 52 exibe o resultado dos experimentos sobre dados sintéticos com variação na abrangência temporal (RAN) e no intervalo de deslocamento (SLI). Ambos parâmetros, RAN e SLI, afetam o comprimento das sequências extraídas. Os valores maiores de RAN proporcionam a extração de sequências maiores, enquanto os valores maiores de SLI causam maior expiração de tuplas e, como efeito, sequências menores. Desta maneira, nas situações com sequências maiores, os algoritmos consomem mais tempo e memória para processar todas as tuplas.

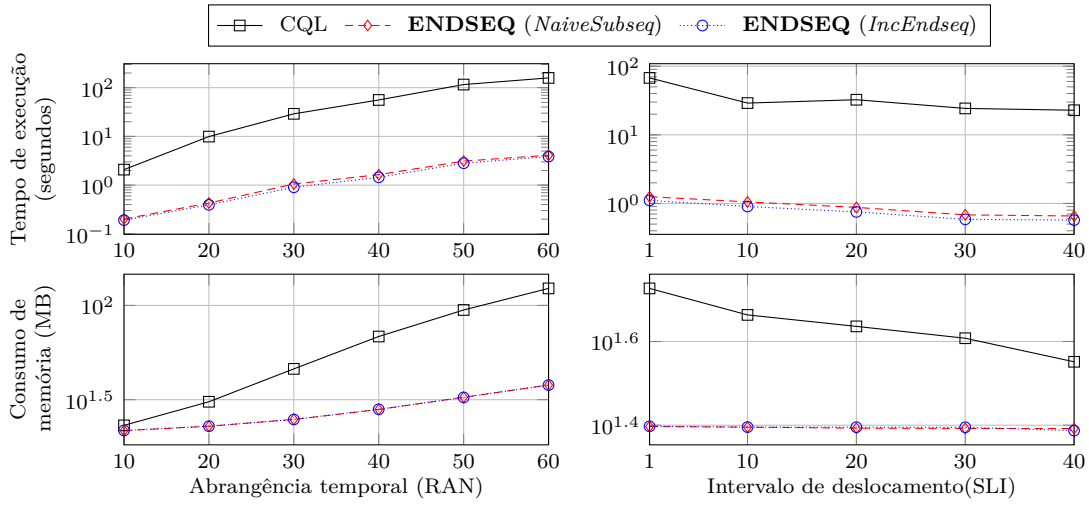


Figura 52 – Experimentos com o operador **ENDSEQ** sobre dados sintéticos variando os parâmetros RAN e SLI

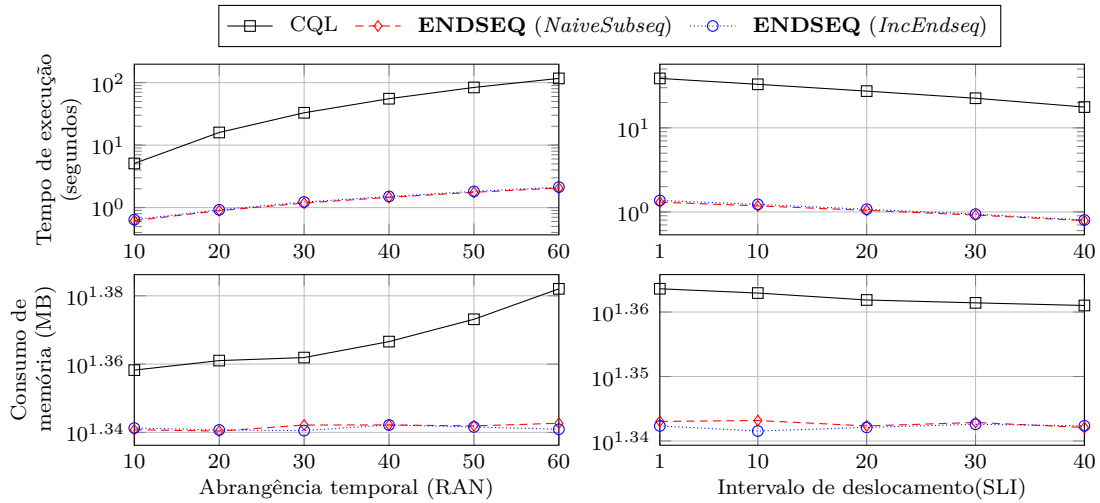


Figura 53 – Experimentos com o operador **ENDSEQ** sobre dados reais variando os parâmetros RAN e SLI

A Figura 53 apresenta o resultado dos experimentos sobre dados reais variando os parâmetros RAN e SLI. Assim como nos experimentos dos operadores **SEQ** e **CONSEQ**, o resultado dos experimentos sobre os dados reais é análogo ao resultado dos experimentos sobre os dados sintéticos considerando estes mesmos parâmetros (RAN e SLI). Em todos os experimentos, a equivalência CQL obteve maior tempo de execução e maior consumo de memória por causa das suas operações intermediárias e das relações temporárias criadas.

8.3.4 Experimentos com o operador MAXSEQ

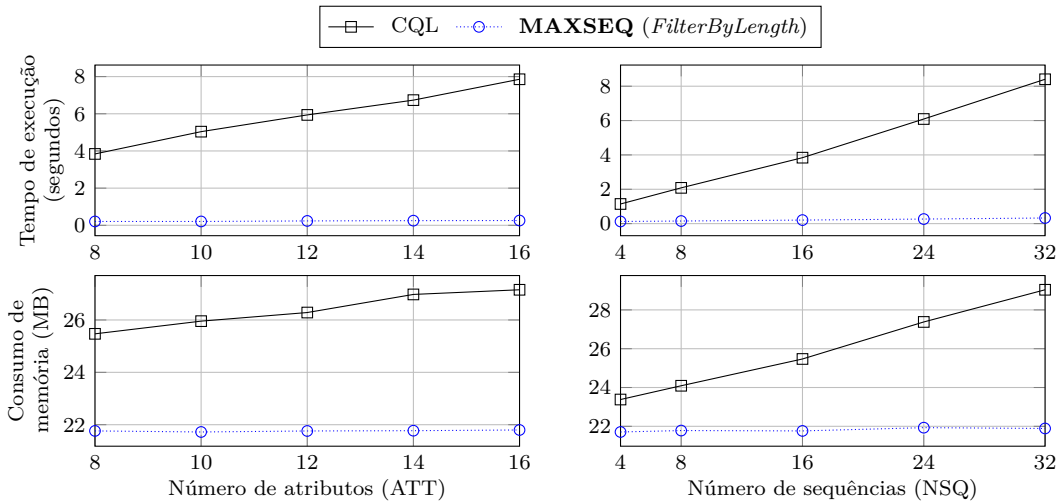


Figura 54 – Experimentos com o operador **MAXSEQ** sobre dados sintéticos variando os parâmetros ATT e NSQ

Os experimentos com o operador **MAXSEQ** comparam o algoritmo *FilterByLength* e sua contraparte em CQL. A Figura 54 mostra o resultado dos experimentos sobre dados sintéticos variando os parâmetros ATT e NSQ. A variação no número de atributos (ATT)

gera um pequeno impacto no desempenho e consumo de memória da equivalência CQL. Já a variação no número de sequências causa um impacto maior. A Equivalência CQL precisa processar operações de junção, diferença de conjunto e eliminação de duplicatas. Isto tem um custo cada vez maior a medida que o número de sequências aumenta.

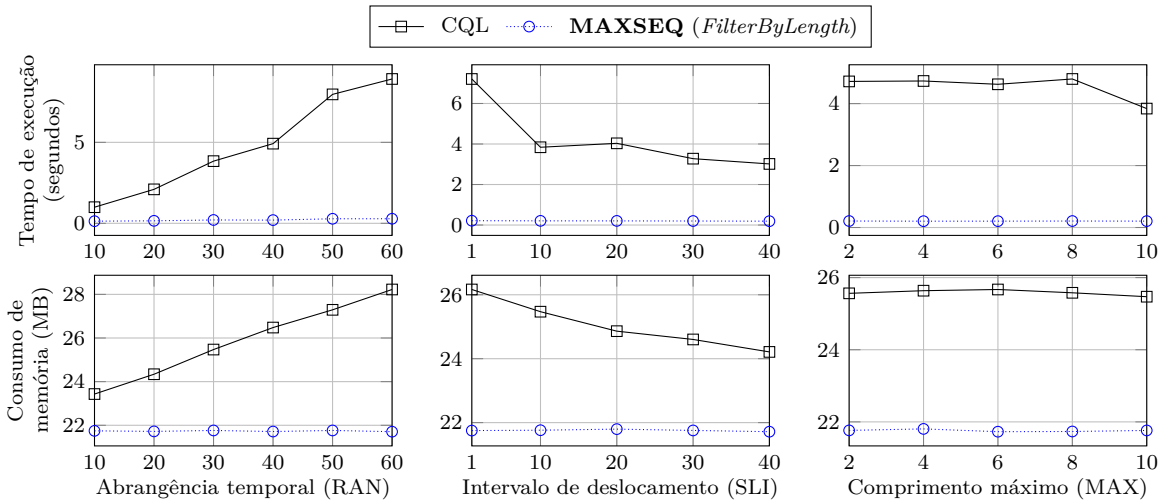


Figura 55 – Experimentos com o operador **MAXSEQ** sobre dados sintéticos variando os parâmetros RAN, SLI, MAX

A Figura 55 exibe o resultado dos experimentos sobre dados sintéticos com variações nos parâmetros RAN, SLI e MAX. A maior oscilação no tempo de execução da equivalência CQL acontece com a variação na abrangência temporal (RAN). Os valores maiores para RAN causam a extração de sequências mais longas, levando a um maior consumo de tempo e memória para processar uma quantidade maior de tuplas.

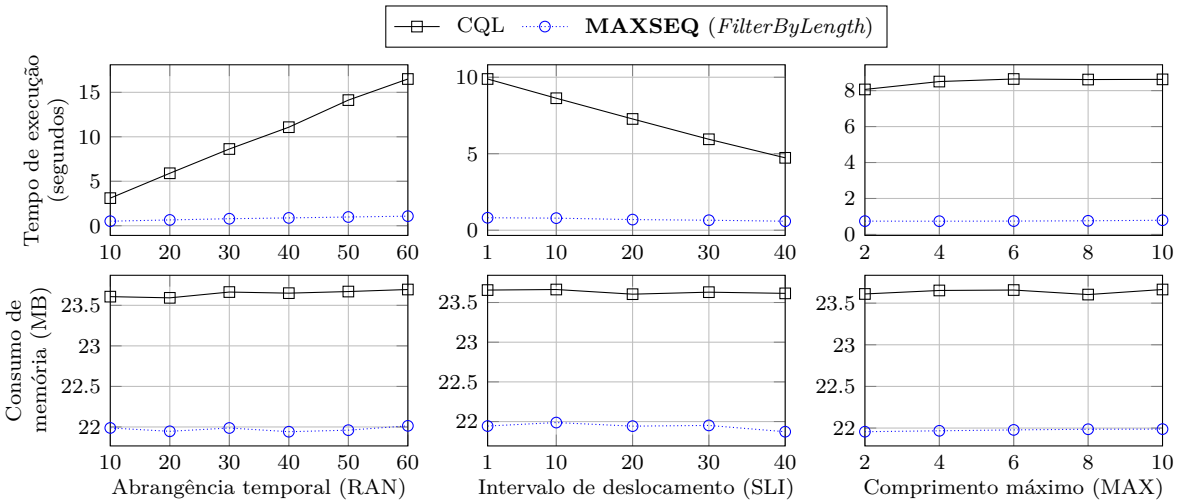


Figura 56 – Experimentos com o operador **MAXSEQ** sobre dados reais variando os parâmetros RAN, SLI, MAX

Os intervalos de deslocamento maiores provocam uma taxa de expiração de tuplas maior. Isto leva a sequências mais curtas com menos tuplas para serem processadas. A variação do parâmetro MAX não afetou significativamente a performance e o consumo

de memória dos algoritmos. O resultado dos experimentos com as mesmas variações de parâmetros sobre dados reais pode ser visto na Figura 56. Os experimentos possuem um resultado similar ao obtido nos mesmos experimentos com dados sintéticos.

Em todos os experimentos o algoritmo *FilterByLength* obteve melhor performance e menor consumo de memória. Isto ocorre porque o algoritmo não precisa computar as operações intermediárias e as relações temporárias da equivalência CQL. O algoritmo simplesmente analisa de forma direta o comprimento de cada sequência recebida.

8.3.5 Experimentos com o operador MINSEQ

Os experimentos com o operador **MINSEQ** comparam o algoritmo *FilterByLength* com sua equivalência CQL. O resultado dos primeiros experimentos sobre dados sintéticos com variação nos parâmetros ATT e NSQ podem ser vistos na Figura 57.

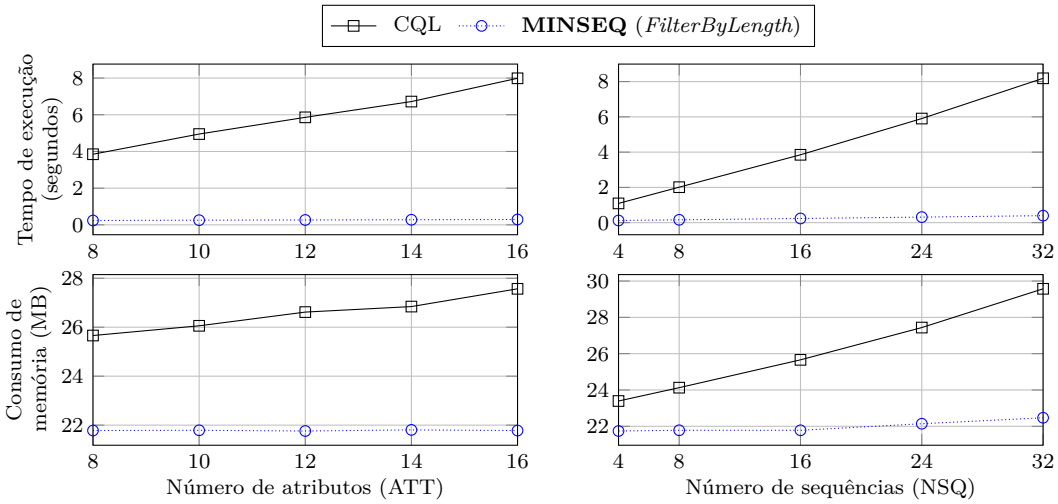


Figura 57 – Experimentos com o operador **MINSEQ** sobre dados sintéticos variando os parâmetros ATT e NSQ

A variação no número de atributos (ATT) tem efeito na operação de eliminação de duplicatas da equivalência CQL. Números maiores de atributos tornam tal operação mais lenta fazendo com que a equivalência CQL tenha um tempo de execução maior. Além disso, a quantidade maior de atributos faz com que as relações temporárias da equivalência ocupem mais espaço em memória. O número de sequências (NSQ) também causa um considerável impacto na equivalência CQL. A medida que o número de sequências aumenta, o número de tuplas também aumenta, fazendo com que o tempo de execução e o consumo de memória sejam maiores.

A Figura 58 apresenta o resultado dos experimentos sobre dados sintéticos variando os parâmetros RAN, SLI e MIN. De forma semelhante ao que aconteceu nos demais experimentos da terceira etapa, a variação nos parâmetros RAN e SLI afeta o comportamento da equivalência CQL de forma inversa. O crescimento da abrangência temporal (RAN) aumenta o número de tuplas a serem processadas fazendo com que a equivalência gaste

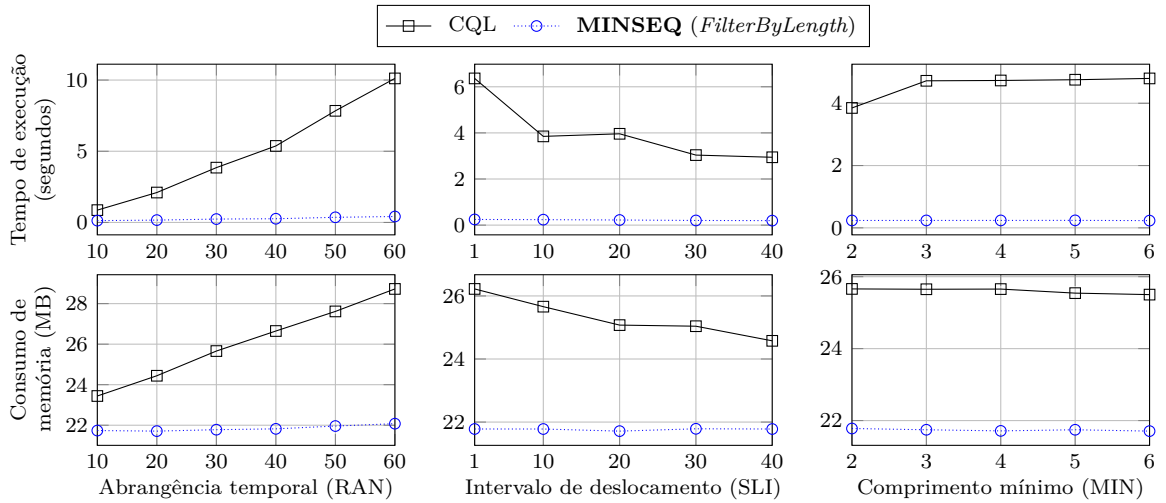


Figura 58 – Experimentos com o operador **MINSEQ** sobre dados sintéticos variando os parâmetros RAN, SLI, MIN

mais tempo e memória. Já os intervalos de deslocamento maiores causam maiores taxas de expiração de tuplas, e as sequências a serem processadas possuem menos tuplas. O parâmetro MIN praticamente não afetou o desempenho dos algoritmos uma vez que tal parâmetro não altera o número de tuplas a serem filtradas.

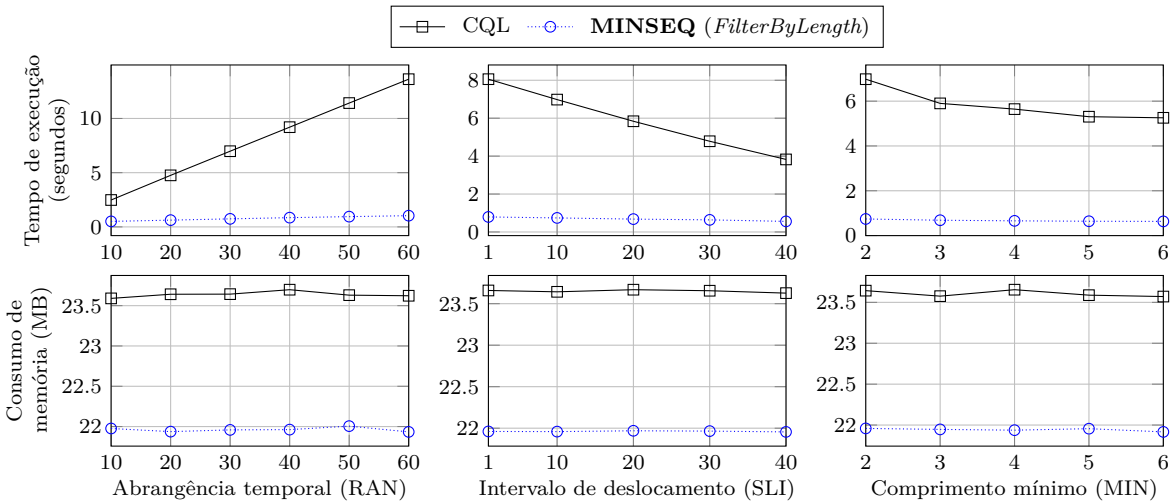


Figura 59 – Experimentos com o operador **MINSEQ** sobre dados reais variando os parâmetros RAN, SLI, MIN

Os mesmos experimentos variando os parâmetros RAN, SLI e MIN foram executados sobre os dados reais. A Figura 59 exibe o resultado destes experimentos. O comportamento dos algoritmos é praticamente o mesmo sobre os dados sintéticos e reais. Em todos os experimentos, o algoritmo *FilterByLength* obteve um desempenho e consumo de memória melhor do que a equivalência CQL. Assim como aconteceu nos experimentos anteriores, as operações intermediárias e as relações temporárias prejudicam a performance e o consumo de memória da equivalência CQL.

8.3.6 Experimentos com o operador BESTSEQ

No caso do operador **BESTSEQ** foram considerados os algoritmos *GetBestSeq*, *IncBestSeq* sem a estratégia de poda, *IncBestSeq* com a estratégia de poda e pela equivalência CQL. As regras-pct das consultas sobre dados sintéticos possuem o formato $\varphi_i : \mathbf{First} \wedge Q(A_3) \rightarrow Q^+(A_2) \succ Q^-(A_2)[A_4, A_5]$ e $\varphi_{i+1} : \mathbf{Prev}\, Q(A_3) \wedge \mathbf{SomePrev}\, Q(A_4) \wedge \mathbf{AllPrev}\, Q(A_5) \wedge Q(A_3) \rightarrow Q^+(A_2) \succ Q^-(A_2)[A_4, A_5]$ com variações nas proposições $Q^+(A_2)$, $Q^-(A_2)$, $Q(A_3)$, $Q(A_4)$, $Q(A_5)$. O Código 11 mostra a consulta usada nos experimentos sobre dados reais. Nestes experimentos ocorrem variações apenas nos parâmetros <RAN> e <SLI>. As preferências usadas na consulta sobre dados reais são representadas pela teoria-pct $\Phi' = \{\varphi_4, \varphi_5, \varphi_6, \varphi_7\}$, onde:

φ_4 : **First** $\rightarrow$ (jogada $= rec$) $\succ$ (jogada $= lbal$);
φ_5 : **Prev**(jogada $= cond$) $\rightarrow$ (jogada $= drib$) $\succ$ (jogada $= pass$)[local];
φ_6 : $\rightarrow$ (jogada $= pass$) $\succ$ (jogada $= bpas$)[local];
φ_7 : **AllPrev**(local $= io$) $\rightarrow$ (local $= io$) $\succ$ (local $= mc$).

Código 11 – Consulta para experimentos com o operador **BESTSEQ** sobre dados reais

```
SELECT SEQUENCE IDENTIFIED BY player_id
[RANGE <RAN> SECOND, SLIDE <SLI> SECOND] FROM s
ACCORDING TO TEMPORAL PREFERENCES
  IF FIRST
    THEN (move = 'rec') BETTER (move = 'lbal') AND
  IF PREVIOUS (move = 'cond')
    THEN (move = 'drib') BETTER (move = 'pass') [place] AND
  (move = 'pass') BETTER (move = 'bpas') [place] AND
  IF ALL PREVIOUS (place = 'io' )
    THEN (place = 'io' ) BETTER (place = 'mc');
```

A Figura 60 apresenta o resultado dos experimentos com variação nos parâmetros ATT e NSQ. O desempenho dos algoritmos foi proporcional à complexidade dos mesmos. A melhor performance ficou com o algoritmo *IncBestSeq* com poda, seguido pelo algoritmo *IncBestSeq* sem poda. O algoritmo *GetBestSeq* ficou com o terceiro melhor desempenho e a equivalência CQL apresentou o maior tempo de execução.

A variação no número de atributos não causou grandes alterações no tempo de execução dos algoritmos. Por outro lado, a medida que o número de sequências aumentou, o tempo de execução de todos os algoritmos tornou-se mais lento. Números maiores de sequências provocam mais comparações que levam mais tempo para serem computadas. Quanto ao consumo de memória, somente a equivalência CQL apresentou um alto consumo devido às diversas relações intermediárias que são armazenadas.

O segundo experimento analisou a variação dos parâmetros RAN e SLI. A Figura 61 exibe os resultados sobre dados sintéticos e a Figura 62 exibe os resultados sobre dados reais. Observe que não foram apresentados os resultados sobre dados reais para a equivalência CQL considerando abrangências temporais maiores do que 30. A equivalência

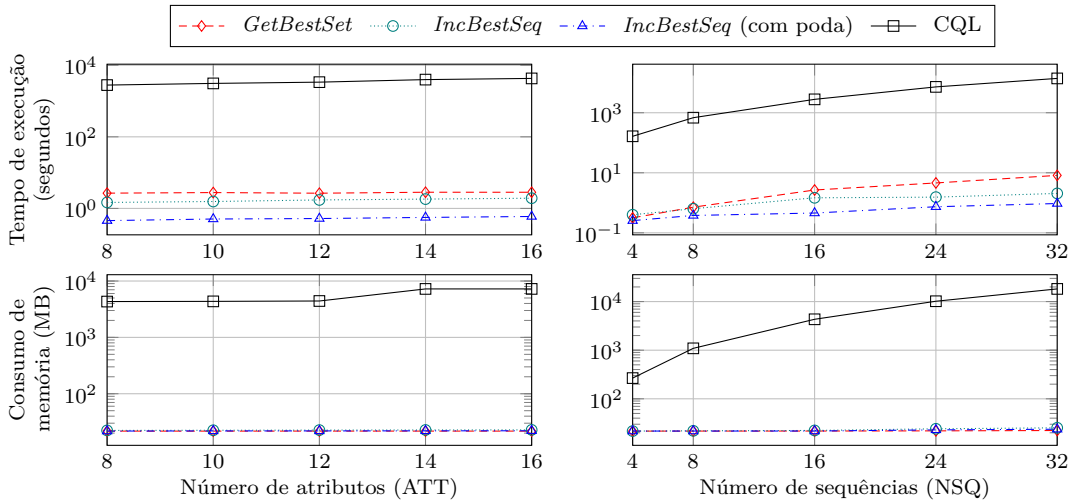


Figura 60 – Experimentos com o operador **BESTSEQ** sobre dados sintéticos variando os parâmetros ATT e NSQ

CQL do operador **BESTSEQ** é muito complexa e contém diversas relações temporárias. As abrangências temporais maiores que 30 geram um número maior de tuplas por instante que causa estouro de memória na execução da equivalência CQL, impossibilitando assim a execução da equivalência CQL nestes cenários.

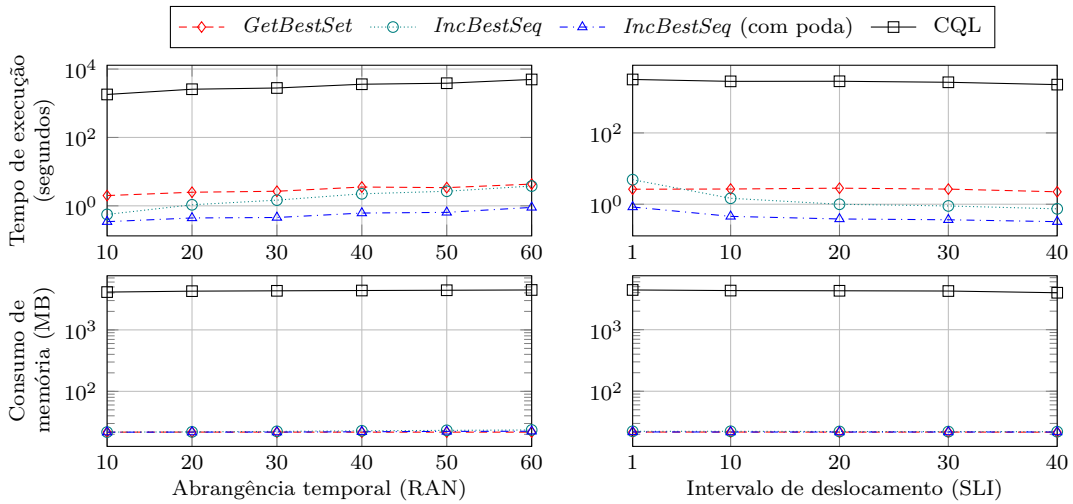


Figura 61 – Experimentos com o operador **BESTSEQ** sobre dados sintéticos variando os parâmetros RAN e SLI

O aumento da abrangência temporal (RAN) causa a extração de sequências mais longas que demandam mais tempo para serem comparadas. Os valores maiores no intervalo de deslocamento (SLI) tem o efeito oposto, as sequências tornam-se menores com a expiração de mais tuplas. Novamente, o algoritmo *IncBestSeq* obteve a melhor performance e a equivalência CQL ficou com o maior tempo de execução. A equivalência CQL obteve um consumo de memória muito maior do que os demais algoritmos por causa das diversas relações temporárias criadas pela equivalência.

A Figura 63 mostra o resultado dos experimentos com variações nos parâmetros

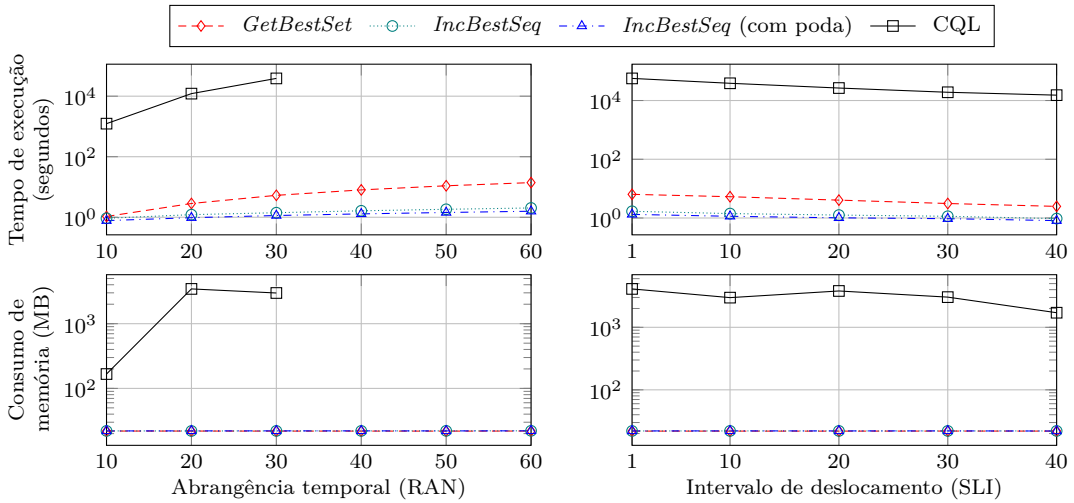


Figura 62 – Experimentos com o operador **BESTSEQ** sobre dados reais variando os parâmetros RAN e SLI

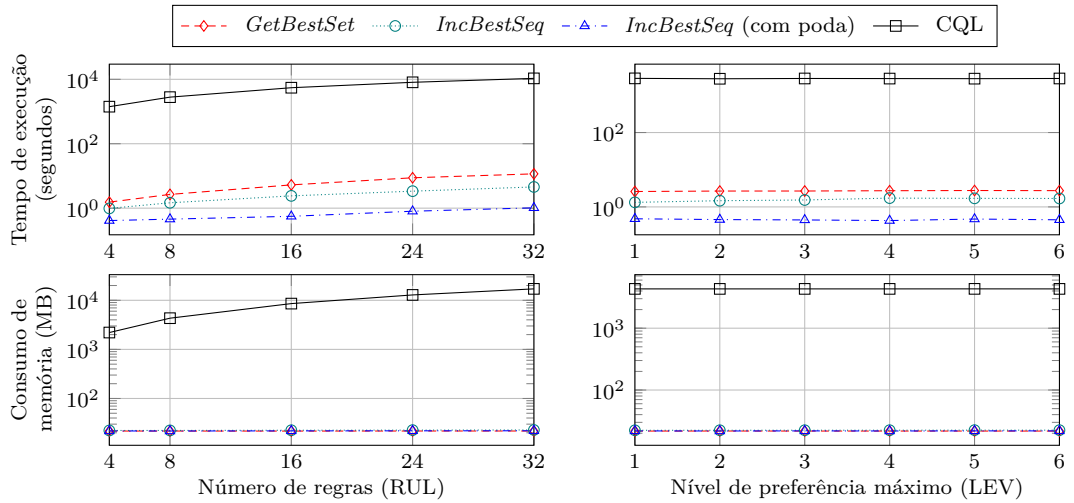


Figura 63 – Experimentos com o operador **BESTSEQ** sobre dados sintéticos variando os parâmetros RUL e LEV

RUL e LEV. A variação no nível de preferência máximo não provoca grandes impactos no desempenho dos algoritmos. Por outro lado, a variação no número de regras afeta significativamente todos os algoritmos. Apesar dos dois parâmetros afetarem a complexidade do teste de dominância, o número de regras causa um impacto maior. No teste de dominância baseado em busca, nem sempre é necessário chegar até as folhas da árvore de busca. Isto ocorre somente quando as sequências são incomparáveis. No caso da árvore de sequência, o nível de preferência afeta somente as hierarquias de preferências dos nós. Novamente, a base de conhecimento construída é afetada pelo nível de preferência, mas o teste de dominância não precisa considerar todas as comparações da base.

Os experimentos mostram que a equivalência CQL obteve o maior tempo de execução e o maior consumo de memória nas variações dos parâmetros RUL e LEV. Mais uma vez, isto ocorre por causa das operações intermediárias e das relações temporárias.

Os algoritmos *GetBestSeq* e *IncBestSeq* obtiveram o mesmo consumo de memória. Já o algoritmo *IncBestSeq* com a estratégia de poda atingiu a melhor performance em todas as variações.

8.3.7 Experimentos de combinações de operadores

O objetivo dos experimentos de combinações de operadores é mostrar como consultas usando diferentes operadores influenciam no número de sequências a serem comparadas e, consequentemente, no número de comparações. Foram analisadas as seguintes combinações de operadores:

- **SEQ**;
- **SEQ** / **CONSEQ**;
- **SEQ** / **CONSEQ** / **ENDSEQ**;
- **SEQ** / **CONSEQ** / **ENDSEQ** / **MINSEQ** / **MAXSEQ**.

Estas combinações foram escolhidas para analisar o processamento de consultas usando as mesmas preferências, mas com diferentes operações de subsequências e filtragem de sequências por tamanho. As sequências retornadas por cada combinação são processadas pelo operador **BESTSEQ**. Durante os experimentos, foram contabilizadas a média por instante de sequências recebidas pelo operador **BESTSEQ** e a média de comparações realizadas por instante.

Os experimentos de combinações de operadores adotaram os mesmos parâmetros da Tabela 6 já usados nos experimentos com cada operador. Os experimentos sobre dados sintéticos usaram o mesmo formato de preferência descrito na Seção 8.3.6 contendo 8 regras-pct e com um nível de preferência máximo igual a 2. A consulta sobre os dados reais foi a mesma usada nos experimentos com o operador **BESTSEQ** (Código 11).

A Figura 64 mostra o número de sequências geradas para comparação sobre os dados sintéticos de acordo com as variações dos parâmetros. O único parâmetro que não influência o número de sequências geradas é o número de atributos (ATT) uma vez que este parâmetro não tem relacionamento com o número de sequências dos dados sintéticos. Observe que, para as variações nos parâmetros MIN e MAX, apenas os resultados para combinações contendo os operadores **MINSEQ** e **MAXSEQ**, respectivamente, foram plotados. As outras combinações não são afetadas por esses parâmetros.

Considerando as variações nos parâmetros NSQ, RAN e SLI, cada combinação de operadores produz um número diferente de sequências. Quando apenas o operador **SEQ** é usado, o número de sequências é igual a NSQ $\times$ 0.75. A inclusão do operador **CONSEQ** aumenta o número de sequências devido à quebra de tuplas não consecutivas. O número máximo de sequências é alcançado quando incluímos o operador **ENDSEQ** para obter as subsequências-up. Lembre-se de que uma sequência de comprimento n produz

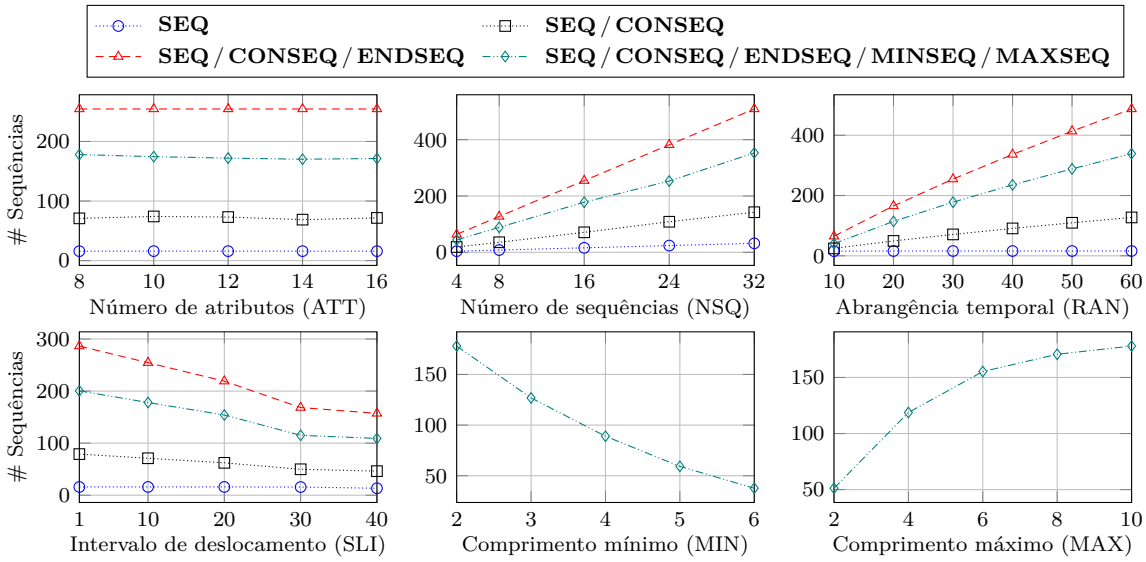


Figura 64 – Número de sequências obtidas nos experimentos de combinações de operadores sobre dados sintéticos

n subsequências-up distintas. Entretanto, o uso dos operadores **MINSEQ** e **MAXSEQ** filtra as sequências por comprimento e o número de sequências é reduzido.

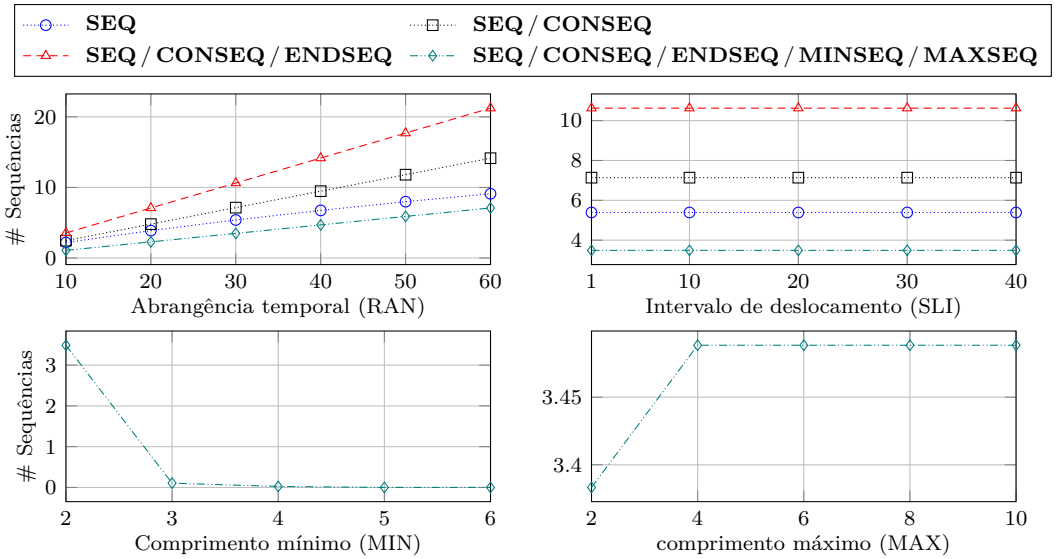


Figura 65 – Número de sequências obtidas nos experimentos de combinações de operadores sobre dados reais

A Figura 65 mostra o número de sequências geradas para comparação sobre os dados reais. Os experimentos sobre os dados reais apresentaram um resultado um pouco diferente em relação aos dados sintéticos. No caso dos experimentos sobre dados reais, a combinação **SEQ / CONSEQ / ENDSEQ / MINSEQ / MAXSEQ** retorna menos sequências do que a combinação **SEQ / CONSEQ**. Isto ocorre porque os dados reais possuem poucos identificadores por instante e poucas tuplas consecutivas. Assim, muitas sequências são eliminadas pela filtragem do operador **MINSEQ** (MIN = 2). No entanto, isso pode ser útil para aplicações reais, uma vez que uma sequência com uma única posição não armazena

informações de momentos passados. Quanto às outras combinações de operadores, o resultado prevalece similar. A combinação **SEQ** retorna o menor número de sequências e a combinação **SEQ / CONSEQ / ENDSEQ** retorna o maior número de sequências.

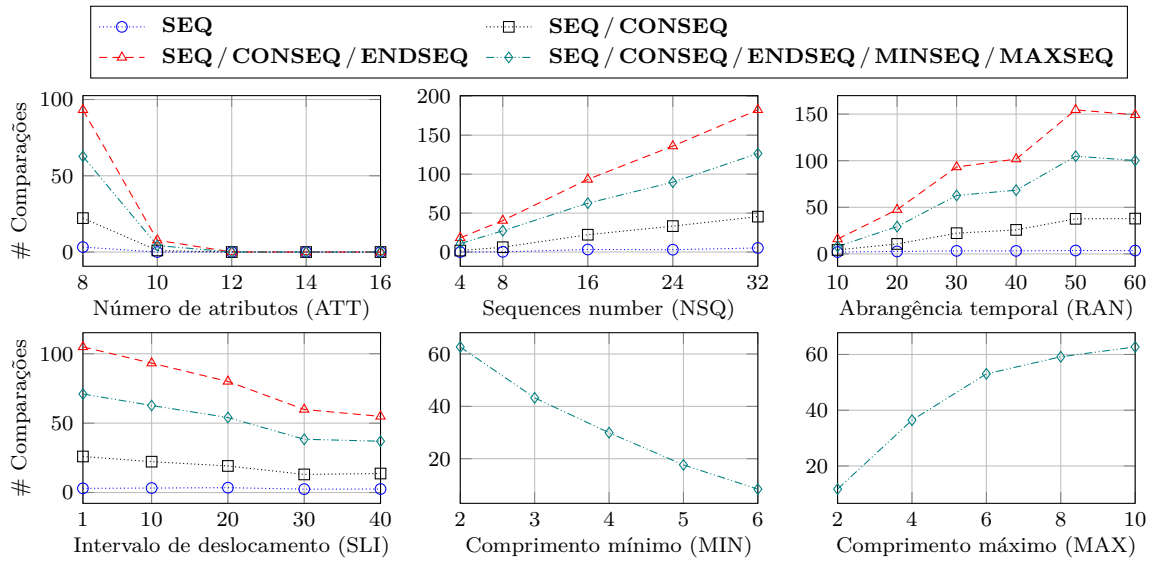


Figura 66 – Número de comparações realizadas nos experimentos de combinações de operadores sobre dados sintéticos

As Figuras 66 e 67 mostram o número médio de comparações por instante dos experimentos sobre dados sintéticos e reais, respectivamente. Exceto pelo parâmetro ATT, em todos os demais parâmetros, o número de comparações está diretamente relacionado com o número de sequências a serem comparadas. Os valores maiores no parâmetro ATT causam menos comparações devido à restrição da semântica *ceteris paribus*.

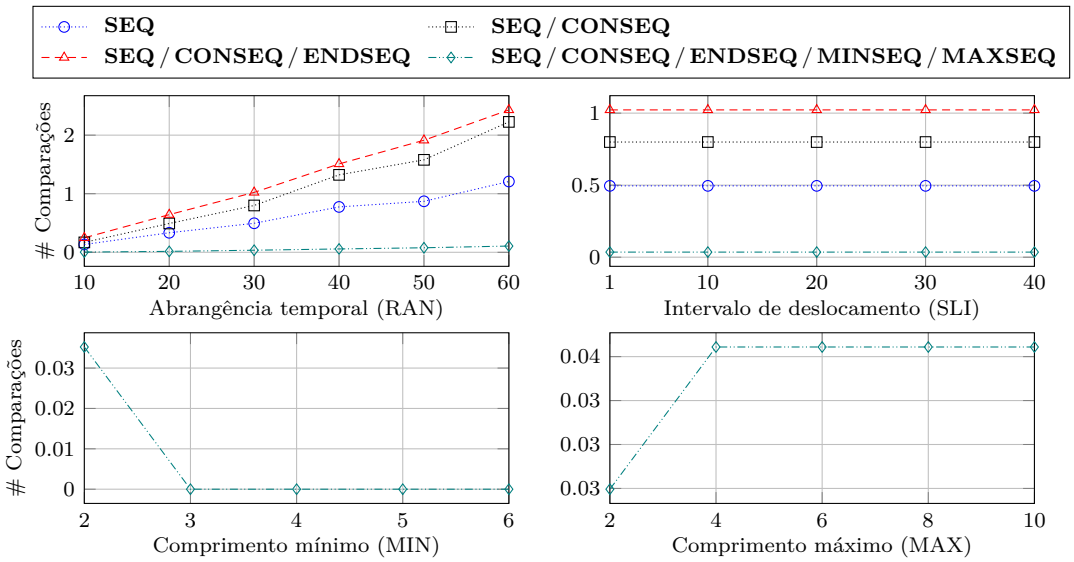


Figura 67 – Número de comparações realizadas nos experimentos de combinações de operadores sobre dados reais

No caso dos dados reais, o número de comparações da combinação **SEQ / CONSEQ / ENDSEQ / MINSEQ / MAXSEQ** ficou menor do que o nú-

mero de comparações da combinação **SEQ**. Isto não significa que a combinação **SEQ / CONSEQ / ENDSEQ / MINSEQ / MAXSEQ** retorna resultados piores. Em bancos de dados reais, o usuário precisa analisar a resposta retornada e mudar as combinações de operadores de acordo com sua necessidade. As respostas com menor número de sequências podem facilitar a tomada de decisão pelo usuário.

8.4 Considerações finais

Este capítulo apresentou todos os experimentos realizados pelo trabalho descrito nesta tese. Os experimentos foram organizados dentro das três etapas do trabalho. A primeira etapa de experimentos comparou o desempenho dos algoritmos para processamento de consultas com preferências em bancos de dados tradicionais. Os experimentos mostraram que o algoritmo *Partition* desenvolvido pelo presente trabalho obteve um desempenho muito superior em relação aos algoritmos do estado-da-arte.

A segunda etapa de experimentos confrontou o algoritmo *IncPartitionBest* com os demais algoritmos do estado-da-arte para processamento de consultas contínuas com preferências condicionais. Os experimentos foram executados sobre dados sintéticos e reais com o objetivo de medir o tempo de execução total e o consumo de memória. Nesta etapa também foram conduzidos experimentos para avaliar o tempo de execução por iteração de todos os algoritmos e um teste de estresse para testar os limites do algoritmo *IncPartitionBest*. O algoritmo *IncPartitionBest* apresentou melhor desempenho em todos experimentos. Além disto, este algoritmo apresentou o menor tempo de resposta médio por iteração, tornando-o mais adequado para cenários de dados em fluxo.

Na terceira etapa foram realizados experimentos para confrontar os algoritmos dos operadores StreamPref e suas respectivas contrapartes. Estes experimentos também consideraram dados sintéticos e reais. Em todos os experimentos, os algoritmos desenvolvidos levaram vantagem sobre as equivalências CQL. Em especial, no caso do operador **BESTSEQ**, o algoritmo *IncBestSeq* usando a estratégia de poda explicada no Capítulo 7 obteve o melhor desempenho.

A terceira etapa envolveu também a realização de experimentos de combinações de operadores que permitiram analisar a quantidade de tuplas extraídas e comparadas de acordo com a utilização de diferentes operadores combinados. Estes experimentos mostraram que a combinação de diferentes operadores poder ser muito útil para o usuário. Dependendo da aplicação, o usuário pode modificar as consultas, incluindo ou excluindo operadores, para que o resultado da consulta atenda melhor às suas necessidades. O próximo capítulo apresenta as considerações finais sobre o trabalho descrito nesta tese e apresenta perspectivas de trabalhos futuros.

Capítulo 9

Conclusão

A partir do ano 2000 houve um crescente aumento de aplicações relacionadas a dados em fluxo. Em muitas destas aplicações é interessante que o usuário possa realizar consultas considerando preferências temporais. Neste tipo de preferência, o usuário pode dizer como os dados de um momento afetam suas preferências sobre os dados em outro instante no tempo. O trabalho descrito nesta tese teve como principal objetivo a criação de um arcabouço teórico e prático que possibilite a realização de consultas contínuas contendo preferências condicionais temporais. Para atingir este objetivo, foi especificada a linguagem de consulta StreamPref juntamente com os formalismos e algoritmos relacionados.

A condução do trabalho foi realizada em três etapas. A primeira etapa, explicada no Capítulo 3, abordou a otimização do processamento de consultas com preferências condicionais em bancos de dados tradicionais. A otimização consistiu em um novo algoritmo que usa uma base de conhecimento contendo as comparações de tuplas (inclusive as comparações transitivas) para processar as consultas de forma mais eficiente.

A segunda etapa do trabalho desenvolveu um novo algoritmo incremental para a processamento mais eficiente de consultas contínuas contendo preferências condicionais. O novo algoritmo usa a técnica da base de conhecimento para manter em memória uma hierarquia de preferência que é atualizada usando apenas as inserções e remoções do último instante de tempo. Desta maneira, não é necessário reprocessar todas as tuplas a cada instante.

A terceira etapa envolveu a especificação completa da linguagem StreamPref contendo novos operadores adequados para lidar com sequências de tuplas. O modelo de preferência usado pela StreamPref teve como base o formalismo TPref. É importante destacar que a especificação das preferências na linguagem StreamPref apresenta algumas diferenças em relação ao formalismo TPref. As condições temporais da linguagem StreamPref não podem ser fórmulas arbitrárias como no formalismo TPref. Entretanto, este formato de condições temporais permitiu desenvolver um teste de consistência viável para qualquer conjunto de preferências temporais da StreamPref. Além disto, as proposições básicas da linguagem StreamPref são mais expressivas do que no formalismo TPref,

permitindo a especificação de intervalos de valores.

No que diz respeito aos operadores algébricos da linguagem, a StreamPref é uma extensão da linguagem CQL. Desta maneira a StreamPref possui os operadores originais da linguagem CQL e novos operadores para extração de sequências e filtragem por comprimento, obtenção de subsequências e operadores de preferência capazes de selecionar as sequências que melhor atendem às preferências temporais do usuário. Foram estabelecidas equivalências entre os novos operadores da linguagem StreamPref e os operadores originais da CQL. Desta maneira, é possível obter o mesmo resultado dos novos operadores por meio de uma combinação de operações na linguagem CQL. No entanto, a escrita de consultas usando as equivalências não é uma tarefa trivial, pois as mesmas envolvem a combinação de diversas operações. Além disto, os experimentos comparativos mostraram um desempenho muito superior dos operadores StreamPref em relação às suas contrapartes em CQL.

Os experimentos conduzidos no decorrer do trabalho comprovaram a eficiência dos algoritmos desenvolvidos. Os algoritmos desenvolvidos na primeira e segunda etapa de trabalho tiveram um desempenho superior ao serem confrontados com os algoritmos estado-da-arte. No caso dos algoritmos da terceira etapa, os experimentos compararam os operadores StreamPref com suas contrapartes em CQL. Sendo que, os operadores StreamPref se sobressaíram em relação às suas equivalências CQL.

Foram conduzidos também experimentos para analisar como diferentes combinações de operadores StreamPref podem afetar as comparações de sequências. Os resultados dos experimentos mostraram variações no número de comparações de acordo com a combinação de operadores usada. Desta maneira, a linguagem StreamPref possui um conjunto de operadores interessante que podem ser combinados de acordo com a necessidade do usuário.

Como foi descrito no Capítulo 1, a hipótese de trabalho era a seguinte: "é possível estender a linguagem CQL com a incorporação de novos operadores que possibilitem o processamento eficiente de consultas contínuas contendo preferências condicionais temporais". Analisando as equivalências dos novos operadores para a linguagem StreamPref, é possível concluir que a linguagem StreamPref não aumenta o poder de expressão da linguagem CQL, pois as consultas com os novos operadores podem ser escritas com os operadores CQL. Considerando também os resultados dos experimentos realizados, é possível concluir que a linguagem StreamPref permite o processamento mais eficiente de consultas contendo preferências condicionais temporais do que o processamento das consultas equivalentes em CQL.

9.1 Principais contribuições

As principais contribuições do trabalho descrito nesta tese são as seguintes:

1) Especificação da linguagem StreamPref, que possui operadores capazes de processar consultas contínuas contendo preferências condicionais temporais. Estes novos ope-

radores englobam operadores de extração de sequências, operadores de obtenção de subsequências, operadores de filtragem de sequências por comprimento e operadores de preferência temporal;

2) Desenvolvimento de algoritmos eficientes para os operadores da linguagem StreamPref;
3) Especificação das equivalências em CQL para os operadores da linguagem StreamPref;
4) Criação de um modelo de preferência que permite expressar preferências condicionais temporais adequadas para o contexto de dados em fluxo;
5) Definição de um teste de consistência viável para qualquer conjunto de preferências temporais na linguagem StreamPref;
6) Desenvolvimento de algoritmos mais eficientes do que os algoritmos estado-da-arte para processamento de consultas contendo preferências condicionais tanto no cenário de dados em fluxo quanto em bancos de dados tradicionais;
7) Criação da base de dados Soccer2014DS contendo eventos de jogadores nas partidas da Copa do Mundo de Futebol de 2014;
8) Desenvolvimento de um protótipo de SGFD que utiliza a linguagem de consulta StreamPref;
9) Criação de ferramentas para geração de dados sintéticos e condução de experimentos no protótipo desenvolvido.

9.2 Trabalhos futuros

A conclusão do trabalho descrito nesta tese deverá abrir um leque de novas propostas de trabalhos futuros. A seguir são listadas algumas destas propostas que merecem destaque.

Análise de desempenho em planos de execução globais

No que se refere a otimizações e execução de consultas concorrentes, é preciso investigar como gerenciar planos de execução globais envolvendo os operadores propostos para a linguagem StreamPref. Esta investigação é importante para a obtenção de ganho de desempenho em ambientes de dados em fluxo com execução de consultas concorrentes.

Preferências compartilhadas entre consultas

Também pode ser útil o desenvolvimento de novos algoritmos específicos para o processamento de múltiplas consultas com preferências. Os algoritmos desenvolvidos no trabalho descrito nesta tese, usaram técnicas de base de conhecimento. O processamento de múltiplas consultas poderia tirar vantagem de preferências comuns a mais de uma consulta construindo uma base de conhecimento para as preferências comuns. Desta maneira, várias consultas poderiam usar esta mesma base de conhecimento.

Novos modelos de preferência

Quanto ao modelo de preferência, seria interessante analisar possibilidades de aumentar o poder de expressão da linguagem e criar novos tipos de testes de dominância. Os operadores de subsequência permitem quebrar as sequências extraídas e aumentar as chances de comparação de sequências. Contudo, pode ser interessante pensar em modelos de preferência que permitam a comparação de sequências considerando não apenas uma posição mas um conjunto de posições do par de sequência comparado. No caso destes novos modelos de preferência é preciso analisar também se o teste de consistência proposto continua válido.

Consultas aproximadas

Outra possibilidade de trabalho futuro é o processamento de consultas aproximadas, de forma que os resultados que não atendem exatamente às preferências possam ser retornados para o usuário como resultados aproximados. Esta proposta precisa definir uma forma de medir o quão próximo de ideal estão estes resultados através de medidas de distância entre sequências que consideram as preferências.

Ferramentas gráficas para especificação de preferências

Apesar da linguagem StreamPref ser bastante expressiva, a escrita de uma consulta em tal linguagem pode não ser trivial para o usuário. Neste sentido, seria interessante desenvolver ferramentas gráficas que facilitem a especificação de consultas contendo preferências por parte do usuário.

Mineração de preferências temporais

Outra maneira de especificar consultas seria por meio de um processo automático que envolve mineração de dados. O desafio desta proposta é desenvolver algoritmos de mineração capazes de minerar regras de preferências temporais que, posteriormente, possam ser usadas em consultas contínuas para monitorar dados que atendam à estas regras mineradas.

Consultas contínuas com evolução de preferências

Outro trabalho futuro interessante seria investigar como avaliar consultas com preferências que mudam com o passar do tempo. Se uma consulta contínua contendo preferências está sendo processada e parte destas preferências sofrem uma mudança, pode ser mais viável modificar as hierarquias de preferências já criadas do que submeter um nova consulta com as preferências alteradas.

Persistência da base de conhecimento

O trabalho descrito nesta tese mostrou que a construção de uma base de conhecimento associada a um conjunto de regras de preferências requer um processamento considerável. Se um mesmo conjunto de regras de preferências for usado por diversas consultas, seria interessante persistir a base de conhecimento relacionada a estas regras para que a mesma não precisasse ser criada novamente sempre que as mesmas preferências forem usadas em futuras consultas.

Outras aplicações da linguagem StreamPref

Além dos exemplos de aplicações em monitoramento de atleta e de bolsa de valores apresentados no trabalho descrito nesta tese, a linguagem StreamPref pode ser aplicada em diversas outras áreas que lidem com fluxos de dados. A linguagem StreamPref pode inclusive ser combinada com outras técnicas como processamento de imagens. Neste caso, uma possível aplicação seria monitorar vídeos considerando padrões temporais de movimentação de objetos.

9.3 Contribuições em produção bibliográfica

A seguir são apresentadas as contribuições bibliográficas produzidas durante a realização deste trabalho:

(1) RIBEIRO, M. R.; PEREIRA, F. S. F.; DIAS, V. V. S. Efficient algorithms for processing preference queries. In: ACM SYMPOSIUM ON APPLIED COMPUTING (ACM SAC), XXXI. **Proceedings...** Pisa, Itália, 2016. p. 972–979. (Ciência da Computação: Qualis A1, Conferências - 2016);

(2) RIBEIRO, M. R.; BARIONI, M. C. N.; RONCANCIO, C.; LABBÉ, C. Incremental evaluation of continuous preference queries. **Information Sciences**, 2016. (submetido). (Ciência da Computação: Qualis A1, Periódicos Quadriênio 2013-2016);

(3) RIBEIRO, M. R.; BARIONI, M. C. N.; AMO, S. de; RONCANCIO, C.; LABBÉ, C. Reasoning with temporal preferences over data streams. In: INTERNATIONAL FLORIDA ARTIFICIAL INTELLIGENCE RESEARCH SOCIETY CONFERENCE (FLAIRS), XXX. **Proceedings...** Marco Island, Flórida, EUA, 2017. p. 700–705. (Ciência da Computação: Qualis B1, Conferências - 2016);

(4) RIBEIRO, M. R.; BARIONI, M. C. N.; AMO, S. de; RONCANCIO, C.; LABBÉ, C. Temporal conditional preference queries on streams. In: INTERNATIONAL CONFERENCE ON DATABASE AND EXPERT SYSTEMS APPLICATIONS (DEXA), XXVIII. **Proceedings...** Lyon, França, 2017. p. 143–158. (Ciência da Computação: Qualis B1, Conferências - 2016);

(5) RIBEIRO, M. R.; BARIONI, M. C. N.; AMO, S. de; RONCANCIO, C.; LABBÉ, C. Soccer2014DS: a dataset containing player events from the 2014 World Cup. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS (SBBD), XXXII. **Anais...** Uberlândia, Minas Gerais, Brasil, 2017. (Dataset Showcase Workshop (DSW)), p. 700–705.;

(6) RIBEIRO, M. R.; BARIONI, M. C. N.; RONCANCIO, C.; LABBÉ, C. Streampref: A query language for temporal conditional preferences on data streams. **IEEE Transactions on Knowledge and Data Engineering (TKDE)**, 2018. (submetido). (Ciência da Computação: Qualis A1, Periódicos Quadriênio 2013-2016).

A publicação (1) se refere ao algoritmo *BestPartition* para processamento de consultas com preferências condicionais em bancos de dados tradicionais explicado no Capítulo 3. Já o artigo (2) diz respeito ao processamento de consultas contínuas com preferências condicionais abordado no Capítulo 4. O artigo passou por alterações solicitadas pelos revisores e está aguardando o segundo processo de revisão. O trabalho (3) está relacionado com o modelo de preferência temporal descrito no Capítulo 5. Neste trabalho também foi apresentado o algoritmo *IncSeq* para extração de sequências e o algoritmo *GetBestSeq* para selecionar as sequências preferidas. A publicação (4) aborda os operadores **SEQ** e **BESTSEQ** juntamente as equivalências CQL destes operadores e com o algoritmo *IncBestSeq*. O artigo (5) trata da obtenção de uma base de dados reais contendo os eventos de jogadores de todas as partidas da Copa do Mundo de Futebol de 2014. Esta base de dados foi usada em diversos experimentos descritos no Capítulo 8. Por fim, o artigo (6) apresenta os demais operadores da linguagem StreamPref, suas equivalências CQL e algoritmos. Tal artigo foi submetido para a revista *IEEE Transactions on Knowledge & Data Engineering*.

Referências

ABADI, D. J. et al. Aurora: a new model and architecture for data stream management. **The VLDB Journal**, v. 12, n. 2, p. 120–139, 2003.

AMO, S. de; BUENO, M. L. d. P. Continuous processing of conditional preference queries. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS (SBBD), XXVI. **Anais...** Florianópolis, Santa Catarina, Brasil, 2011.

AMO, S. de; GIACOMETTI, A. Temporal conditional preferences over sequences of objects. In: IEEE INTERNATIONAL CONFERENCE ON TOOLS WITH ARTIFICIAL INTELLIGENCE (ICTAI), XIX. **Proceedings...** Patras, Grécia, 2007. p. 246–253.

AMO, S. de; GIACOMETTI, A. Preferences over objects, sets and sequences. In: _____. **Tools in Artificial Intelligence**. Rijeka, Croácia: InTech, 2008. cap. 4, p. 49–76.

AMO, S. de; RIBEIRO, M. R. CPref-SQL: A query language supporting conditional preferences. In: ACM SYMPOSIUM ON APPLIED COMPUTING (ACM SAC), XXIV. **Proceedings...** Honolulu, Hawaii, EUA, 2009. p. 1573–1577.

ARASU, A. et al. STREAM: The stanford stream data manager. **IEEE Data Engineering Bulletin**, v. 26, n. 1, p. 19–26, 2003.

ARASU, A.; BABCOCK, B.; BABU, S.; MCALISTER, J.; WIDOM, J. Characterizing memory requirements for queries over continuous data streams. **ACM Transactions on Database Systems (TODS)**, v. 29, n. 1, p. 162–194, 2004.

ARASU, A. et al. Stream: The stanford data stream management system. In: _____. **Data Stream Management: Processing High-Speed Data Streams**. Berlin, Alemanha: Springer, 2016. p. 317–336.

ARASU, A.; BABU, S.; WIDOM, J. The CQL continuous query language: semantic foundations and query execution. **The VLDB Journal**, v. 15, n. 2, p. 121–142, 2006.

ARASU, A.; WIDOM, J. A denotational semantics for continuous queries over streams and relations. **ACM SIGMOD Record**, v. 33, n. 3, p. 6–11, 2004.

BABCOCK, B.; BABU, S.; DATAR, M.; MOTWANI, R.; WIDOM, J. Models and issues in data stream systems. In: ACM SIGMOD SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS (PODS), XXI. **Proceedings...** Madison, Wisconsin, EUA, 2002. p. 1–16.

BABU, S.; MUNAGALA, K.; WIDOM, J.; MOTWANI, R. Adaptive caching for continuous queries. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING (ICDE), XXI. **Proceedings...** Tóquio, Japão, 2005. p. 118–129.

BABU, S.; WIDOM, J. Continuous queries over data streams. **ACM SIGMOD Record**, v. 30, n. 3, p. 109–120, 2001.

BENTON, J.; COLES, A. J.; COLES, A. Temporal planning with preferences and time-dependent continuous costs. In: INTERNATIONAL CONFERENCE ON AUTOMATED PLANNING AND SCHEDULING (ICAPS), XXII. **Proceedings...** Atibaia, São Paulo, Brazil, 2012. v. 77, p. 78.

BIENVENU, M.; FRITZ, C.; MCILRAITH, S. A. Planning with qualitative temporal preferences. In: INTERNATIONAL CONFERENCE ON KNOWLEDGE REPRESENTATION AND REASONING (KR), X. **Proceedings...** Lake District, Inglaterra, 2006. p. 134–144.

BÖRZSÖNYI, S.; KOSSMANN, D.; STOCKER, K. The skyline operator. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING (ICDE), XVII. **Proceedings...** Heidelberg, Alemanha, 2001. p. 421–430.

BOUTILIER, C.; BRAFMAN, R. I.; DOMSHLAK, C.; HOOS, H. H.; POOLE, D. CP-Nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. **Journal of Artificial Intelligence Research (JAIR)**, v. 21, p. 135–191, 2004.

BOUTILIER, C.; BRAFMAN, R. I.; HOOS, H. H.; POOLE, D. Reasoning with conditional ceteris paribus preference statements. In: CONFERENCE ON UNCERTAINTY IN ARTIFICIAL INTELLIGENCE (UAI), XV. **Proceedings...** Estocolmo, Suécia, 1999. p. 71–80.

BRAFMAN, R. I.; DOMSHLAK, C. Introducing variable importance tradeoffs into CP-Nets. In: CONFERENCE ON UNCERTAINTY IN ARTIFICIAL INTELLIGENCE (UAI), XVIII. **Proceedings...** Edmonton, Canadá, 2002. p. 69–76.

CHAN, C.-Y.; JAGADISH, H. V.; TAN, K.-L.; TUNG, A. K. H.; ZHANG, Z. Finding k-dominant skylines in high dimensional space. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, XXXII. **Proceedings...** Chicago, Illinois, EUA, 2006. p. 503–514.

CHANDRASEKARAN, S. et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In: CONFERENCE ON INNOVATIVE DATA SYSTEMS RESEARCH (CIDR), I. **Proceedings...** Asilomar, Califórnia, EUA, 2003.

CHEN, J.; DEWITT, D. J.; TIAN, F.; WANG, Y. NiagaraCQ: A scalable continuous query system for internet databases. **ACM SIGMOD Record**, v. 29, n. 2, p. 379–390, 2000.

CHOMICKI, J.; CIACCIA, P.; MENEGHETTI, N. Skyline queries, front and back. **ACM SIGMOD Record**, v. 42, n. 3, p. 6–18, 2013.

GEDIK, B.; ANDRADE, H.; WU, K.-L.; YU, P. S.; DOO, M. SPADE: The system s declarative stream processing engine. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, XXXIV. **Proceedings...** Vancouver, Canadá, 2008. p. 1123–1134.

GOLAB, L.; ÖZSU, M. T. Issues in data stream management. **ACM SIGMOD Record**, v. 32, n. 2, p. 5–14, 2003.

HÖFNER, P.; MÖLLER, B. Dijkstra, Floyd and Warshall meet Kleene. **Formal Aspects of Computing**, v. 24, n. 4-6, p. 459–476, 2012.

HRISTIDIS, V.; KOUDAS, N.; PAPAKONSTANTINOU, Y. PREFER: A system for the efficient execution of multi-parametric ranked queries. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, XXVII. **Proceedings...** Santa Bárbara, Califórnia, EUA, 2001. p. 259–270.

JAIN, N. et al. Towards a streaming SQL standard. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES (VLDB), XXXIV. **Proceedings...** Auckland, Nova Zelândia, 2008. p. 1379–1390.

KAHN, A. B. Topological sorting of large networks. **Communications of the ACM**, v. 5, n. 11, p. 558–562, 1962.

KONTAKI, M.; PAPADOPOULOS, A. N.; MANOLOPOULOS, Y. Continuous top-k dominating queries. **IEEE Transactions on Knowledge and Data Engineering (TKDE)**, v. 24, n. 5, p. 1041–4347, 2012.

KRÄMER, J. **Continuous queries over data streams - semantics and implementation**. Tese (Doutorado) — University of Marburg, Marburg, Alemanha, 2007.

LEE, Y. W.; LEE, K. Y.; KIM, M. H. Efficient processing of multiple continuous skyline queries over a data stream. **Information Sciences**, v. 221, p. 316–337, 2013.

LIN, X.; YUAN, Y.; WANG, W.; LU, H. Stabbing the sky: efficient skyline computation over sliding windows. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING (ICDE), XXI. **Proceedings...** Tóquio, Japão, 2005. p. 502–513.

MORSE, M.; PATEL, J. M.; GROSKY, W. I. Efficient continuous skyline computation. **Information Sciences**, v. 177, n. 17, p. 3411–3437, 2007.

MOTWANI, R.; THOMAS, D. Caching queues in memory buffers. In: ACM-SIAM SYMPOSIUM ON DISCRETE ALGORITHMS (SODA), XV. **Proceedings...** New Orleans, Louisiana, EUA, 2004. p. 541–549.

MOURATIDIS, K.; BAKIRAS, S.; PAPADIAS, D. Continuous monitoring of top-k queries over sliding windows. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, XXXII. **Proceedings...** Chicago, Illinois, EUA, 2006. p. 635–646.

PANDEY, S.; RAMAMRITHAM, K.; CHAKRABARTI, S. Monitoring the dynamic web to respond to continuous queries. In: INTERNATIONAL CONFERENCE ON WORLD WIDE WEB, XII. **Proceedings...** Budapeste, Hungria, 2003. p. 659–668.

PAPADIAS, D.; TAO, Y.; FU, G.; SEEGER, B. Progressive skyline computation in database systems. **ACM Transactions on Database Systems (TODS)**, v. 30, n. 1, p. 41–82, 2005.

PEREIRA, F. S. F. **CPref-SQL: uma Linguagem de Consulta com Suporte a Preferências Condicionais - Teoria e Implementação**. Dissertação (Mestrado) — Universidade Federal de Uberlândia, Uberlândia, Minas Gerais, Brasil, 2011.

PEREIRA, F. S. F.; AMO, S. de. Evaluation of conditional preference queries. **Journal of Information and Data Management (JIDM)**, v. 1, n. 3, p. 503–518, 2010.

PETIT, L.; AMO, S. de; RONCANCIO, C.; LABBÉ, C. Top-k context-aware queries on streams. In: INTERNATIONAL CONFERENCE ON DATABASE AND EXPERT SYSTEMS APPLICATIONS (DEXA), XXIII. **Proceedings...** Vienna, Áustria, 2012. p. 397–411.

PETIT, L.; LABBÉ, C.; RONCANCIO, C. An algebric window model for data stream management. In: ACM INTERNATIONAL WORKSHOP ON DATA ENGINEERING FOR WIRELESS AND MOBILE ACCESS (MOBIDE), IX. **Proceedings...** Indianápolis, Indiana, EUA, 2010. p. 17–24.

PETIT, L.; LABBÉ, C.; RONCANCIO, C. Revisiting formal ordering in data stream querying. In: ACM SYMPOSIUM ON APPLIED COMPUTING (ACM SAC), XXVII. **Proceedings...** Trento, Itália, 2012. p. 813–818.

PRIOR, A. N. **Past, Present and Future**. Oxford, Nova York, EUA: Oxford University Press, 1967.

RIBEIRO, M. R. **Linguagens de consulta para a banco de dados com suporte a preferências condicionais**. Dissertação (Mestrado) — Universidade Federal de Uberlândia, Uberlândia, Minas Gerais, Brasil, 2008.

RIBEIRO, M. R. **StreamPref DSMS Prototype**. 2017. Disponível em: <http://streampref.github.io>. Acesso em: 26/01/2018.

RIBEIRO, M. R.; BARIONI, M. C. N.; AMO, S. de; RONCANCIO, C.; LABBÉ, C. Reasoning with temporal preferences over data streams. In: INTERNATIONAL FLORIDA ARTIFICIAL INTELLIGENCE RESEARCH SOCIETY CONFERENCE (FLAIRS), XXX. **Proceedings...** Marco Island, Flórida, EUA, 2017. p. 700–705.

RIBEIRO, M. R.; BARIONI, M. C. N.; AMO, S. de; RONCANCIO, C.; LABBÉ, C. Soccer2014DS: a dataset containing player events from the 2014 World Cup. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS (SBBD), XXXII. **Anais...** Uberlândia, Minas Gerais, Brasil, 2017. (Dataset Showcase Workshop (DSW)), p. 700–705.

RIBEIRO, M. R.; BARIONI, M. C. N.; AMO, S. de; RONCANCIO, C.; LABBÉ, C. Temporal conditional preference queries on streams. In: INTERNATIONAL CONFERENCE ON DATABASE AND EXPERT SYSTEMS APPLICATIONS (DEXA), XXVIII. **Proceedings...** Lyon, França, 2017. p. 143–158.

RIBEIRO, M. R.; BARIONI, M. C. N.; RONCANCIO, C.; LABBÉ, C. Incremental evaluation of continuous preference queries. **Information Sciences**, 2016. (submetido).

RIBEIRO, M. R.; BARIONI, M. C. N.; RONCANCIO, C.; LABBÉ, C. Streampref: A query language for temporal conditional preferences on data streams. **IEEE Transactions on Knowledge and Data Engineering (TKDE)**, 2018. (submetido).

RIBEIRO, M. R.; PEREIRA, F. S. F.; DIAS, V. V. S. Efficient algorithms for processing preference queries. In: ACM SYMPOSIUM ON APPLIED COMPUTING (ACM SAC), XXXI. **Proceedings...** Pisa, Itália, 2016. p. 972–979.

SANTOSO, B. J.; CHIU, G.-M. Close dominance graph: An efficient framework for answering continuous top-dominating queries. **IEEE Transactions on Knowledge and Data Engineering (TKDE)**, v. 26, n. 8, p. 1853–1865, 2014.

SARKAS, N.; DAS, G.; KOUDAS, N.; TUNG, A. K. H. Categorical skylines for streaming data. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, XXXIV. **Proceedings...** Vancouver, Canadá, 2008. p. 239–250.

SHEN, Z.; CHEEMA, M. A.; LIN, X.; ZHANG, W.; WANG, H. A generic framework for top-k pairs and top-k objects queries over sliding windows. **IEEE Transactions on Knowledge and Data Engineering (TKDE)**, v. 26, n. 6, p. 1349–1366, 2012.

SRIVASTAVA, U.; WIDOM, J. Flexible time management in data stream systems. In: ACM SIGMOD SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS (PODS), XXIII. **Proceedings...** Paris, França, 2004. p. 263–274.

TAO, Y.; PAPADIAS, D. Maintaining sliding window skylines on data streams. **IEEE Transactions on Knowledge and Data Engineering (TKDE)**, v. 18, n. 3, p. 377–391, 2006.

TERRY, D.; GOLDBERG, D.; NICHOLS, D.; OKI, B. Continuous queries over append-only databases. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, XVIII. **Proceedings...** San Diego, Califórnia, EUA, 1992. p. 321–330.

TRANSACTION PROCESSING PERFORMANCE COUNCIL. **TPC Benchmark H**. 2014. Disponível em: <http://www.tpc.org/tpch>. Acesso em: 10/08/2014.

WILSON, N. Extending CP-Nets with stronger conditional preference statements. In: AAAI NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, XIX. **Proceedings...** San Jose, Califórnia, EUA, 2004. p. 735–741.

Yahoo! Inc. **Yahoo Finance**. 2015. Disponível em: <http://finance.yahoo.com/>. Acesso em: 30/07/2016.

YIU, M. L.; MAMOULIS, N. Efficient processing of top-k dominating queries on multi-dimensional data. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES (VLDB), XXXIII. **Proceedings...** Vienna, Áustria, 2007. p. 483–494.

ZHU, R.; WANG, B.; YANG, X.; ZHENG, B.; WANG, G. SAP: Improving continuous top-k queries over streaming data. **IEEE Transactions on Knowledge and Data Engineering (TKDE)**, v. 29, n. 6, p. 1310–1328, June 2017.

Apêndices

APÊNDICE A

Gramática da linguagem StreamPref

Gramática da linguagem de consulta StreamPref usando o Formalismo de Backus-Naur Estendido (EBNF).

```
query = ( stream_query | sequence_query | bag_query ) , ";" ;
stream_query = "SELECT" , stream_operation , "FROM" , table_window ;
sequence_query = "SELECT" , [ topk ] ,
                 [ [ end_position ] , consecutive_tuples ] ,
                 sequence_operation ,
                 "FROM" , from_stream ,
                 [ "WHERE" , min_max_lenght ] ,
                 [ temporal_preferences ] ;
bag_query = simple_query , { bag_operation , simple_query } ;
end_position = "SUBSEQUENCE" , "END" , "POSITION" , "FROM" ;
consecutive_tuples = "SUBSEQUENCE" , "CONSECUTIVE" , "TUPLES" , "FROM" ;
sequence_operation = "SEQUENCE" , "IDENTIFIED" , "BY" attribute_list ,
                     sequence_window ;
temporal_preferences = [ "ACCORDING" , "TO"] , "TEMPORAL" , "PREFERENCES" ,
                       temporal_theory ;
from_stream =  stream , [ "AS" , identifier] |
              stream_operation , "FROM" , "(" , table_window , ")" ;
stream_operation = "DSTREAM" | "ISTREAM" | "RSTREAM" ;
sequence_window = "[" , "RANGE" , time_term  ,
                  [ "," , "SLIDE" , time_term  ] , "]" ;
min_max_lenght = minimum | maximum | minimum , "AND" , maximum ;
minimum = "MINIMUM" , "LENGTH" , "IS" , integer ;
maximum = "MAXIMUM" , "LENGTH" , "IS" , integer ;
simple_query = simple_select , simple_from , [ where ] , [ preferences ] ,
               [ group_by ] ;
simple_select = "SELECT" , [ "DISTINCT" ] , [ topk ] ,
                select_term , { "," , select_term } ;
```

```
simple_from = "FROM" , table_window , { "," , table_window } ;
group_by = "GROUP BY" , identifier , { "," , identifier } ;
bag_operation = "UNION" | "DIFFERENCE" | "EXCEPT" ;
select_term = expression , [ "AS" , identifier ] |
              aggregation_expression , [ "AS" , identifier ] |
              [ identifier , "." ] , "*" ;
where = "WHERE" , where_term , { "AND" , where_term } |
        "WHERE" , where_term , { "OR"  , where_term } ;
where_term = [ "NOT" ] , expression , operator , expression ;
preferences = [ "ACCORDING" , "TO" ] , "PREFERENCES" , theory ;
table_window = identifier , [ "[" , window , "]" , ] ,
               [ "AS" , identifier ] ;
window = "NOW" | [ "RANGE" ] , "UNBOUNDED" |
         "RANGE" , time_term , [ "," , "SLIDE" , time_term ] ;
time_term = integer ( "SECOND" | "MINUTE" | "HOUR" | "DAY" ) ;
topk = "TOP" , "(" , integer , ")";
temporal_theory = temporal_rule , { "AND" , temporal_rule } ;
theory = rule , { "AND" , rule } ;
temporal_rule = [ "IF" , temporal_condition , "THEN" ] ,
                preference , [ indifferent_attributes] ;
rule = [ "IF" , condition , "THEN" ],  preference ,
       [ indifferent_attributes] ;
temporal_condition = temporal_predicate , { "AND", temporal_predicate } ;
condition = predicate , { "AND", predicate } ;
temporal_predicate = predicate | derived_formula ;
derived_formula = "FIRST" | "PREVIOUS" , "(", predicate , ")" |
                  "SOME" , "PREVIOUS" , "(", predicate , ")" |
                  "ALL" , "PREVIOUS" , "(", predicate , ")" ;
preference = predicate , ( "BETTER" | ">" ) , predicate ;
predicate = identifier , operator , value |
            value , interval_operator , identifier ,
            interval_operator , value ;
indifferent_attributes = "[" , identifier , { identifier } , "]" ;
expression = identifier | value | expression , arithmetic , expression |
             "(" , expression , ")" ;
aggregation_expression = aggregation_function, "(", expression , ")" ;
aggregation_function = "MIN" | "MAX" | "SUM" | "COUNT" ;
operator = "<" | "<=" | "=" | ">=" | ">" ;
interval_operator = "<" | "<=" ;
identifier = alpha , { alpha | digit } ;
value = integer | float | string ;
float = integer | integer , "." , { integer } | { integer } , "." , integer ;
```

```
integer = digit , { digit } ;
string = "'" , { alpha | digit } , "'" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
alpha = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" |
        "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "W" | "X" |
        "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
        "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" |
        "w" | "x" | "y" | "z" ;
arithmetic = "+" | "-" | "*" | "/" ;
```

APÊNDICE B

Exemplos de execução dos algoritmos dos operadores StreamPref

Este apêndice apresenta exemplos de execução dos algoritmos dos operadores StreamPref. As novas tuplas são representadas em negrito, já uma tupla t expirada é denotada como $\not{t}$. Todos os exemplos de execução usam o fluxo `evento` a seguir:

TS()	jid	jogada	local
1	1	pass	mc
2	1	lbal	mc
6	3	rec	io
7	3	cond	io
8	3	bpas	io
12	1	rec	mc
13	1	cond	io

TS()	jid	jogada	local
14	1	drib	io
15	2	lbal	mc
16	2	cond	io
17	2	pass	io
18	4	rec	io
19	4	cond	mc

TS()	jid	jogada	local
20	4	cond	ao
21	1	cond	io
22	1	drib	io
23	2	lbal	mc
24	3	cond	io
25	2	pass	io

B.1 Exemplo de execução do algoritmo *IncSeq*

Esta seção apresenta a execução do algoritmo *IncSeq* sobre o fluxo `evento` usando o identificador `jid`, uma abrangência temporal de 20 segundos e um intervalo de deslocamento de um segundo durante 25 iterações.

- **Instante 0:**
  - Tabela hash $Z_{\#}$: vazia;
  - Sequências retornadas: nenhuma.
- **Instante 1:**
  - Tabela hash $Z_{\#}$:

$$1 \mapsto \boxed{\langle(\mathbf{pass}, \mathbf{mc})\rangle}$$

  - Sequências retornadas:

    $s_1 = \langle(pass, mc)\rangle$.

- **Instante 2:**
  - Tabela hash $Z_{\#}$:

$$1 \mapsto \boxed{\langle (pass, mc), (\mathbf{lbal}, \mathbf{mc}) \rangle}$$

  - Sequências retornadas:

$s_1 = \langle (pass, mc), (lbal, mc) \rangle$.

- **Instantes 3, 4 e 5:** Mesmo resultado do instante 2.
- **Instante 6:**
  - Tabela hash $Z_{\#}$:

$$1 \mapsto \boxed{\langle (pass, mc), (lbal, mc) \rangle}$$
$$3 \mapsto \boxed{\langle (\mathbf{rec}, \mathbf{io}) \rangle}$$

  - Sequências retornadas:

$s_1 = \langle (pass, mc), (lbal, mc) \rangle$;
$s_3 = \langle (rec, io) \rangle$.

- **Instante 7:**
  - Tabela hash $Z_{\#}$:

$$1 \mapsto \boxed{\langle (pass, mc), (lbal, mc) \rangle}$$
$$3 \mapsto \boxed{\langle (rec, io), (\mathbf{cond}, \mathbf{io}) \rangle}$$

  - Sequências retornadas:

$s_1 = \langle (pass, mc), (lbal, mc) \rangle$;
$s_3 = \langle (rec, io), (cond, io) \rangle$.

- **Instante 8:**
  - Tabela hash $Z_{\#}$:

$$1 \mapsto \boxed{\langle (pass, mc), (lbal, mc) \rangle}$$
$$3 \mapsto \boxed{\langle (rec, io), (cond, io), (\mathbf{bpas}, \mathbf{io}) \rangle}$$

  - Sequências retornadas:

$s_1 = \langle (pass, mc), (lbal, mc) \rangle$;
$s_3 = \langle (rec, io), (cond, io), (bpas, io) \rangle$.

- **Instantes 9, 10 e 11:** Mesmo resultado do instante 8.
- **Instante 12:**
  - Tabela hash $Z_{\#}$:

$$1 \mapsto \boxed{\langle (pass, mc), (lbal, mc), (\mathbf{rec}, \mathbf{mc}) \rangle}$$
$$3 \mapsto \boxed{\langle (rec, io), (cond, io), (bpas, io) \rangle}$$

– Sequências retornadas:

$s_1 = \langle (pass, mc), (lbal, mc), (rec, mc) \rangle$;
$s_3 = \langle (rec, io), (cond, io), (bpas, io) \rangle$.

- **Instante 13:**

– Tabela hash $Z_{\#}$:

$$1 \mapsto \boxed{\langle (pass, mc), (lbal, mc), (rec, mc), (\mathbf{cond}, \mathbf{io}) \rangle}$$
$$3 \mapsto \boxed{\langle (rec, io), (cond, io), (bpas, io) \rangle}$$

– Sequências retornadas:

$s_1 = \langle (pass, mc), (lbal, mc), (rec, mc), (cond, io) \rangle$;
$s_3 = \langle (rec, io), (cond, io), (bpas, io) \rangle$.

- **Instante 14:**

– Tabela hash $Z_{\#}$:

$$1 \mapsto \boxed{\langle (pass, mc), (lbal, mc), (rec, mc), (cond, io), (\mathbf{drib}, \mathbf{io}) \rangle}$$
$$3 \mapsto \boxed{\langle (rec, io), (cond, io), (bpas, io) \rangle}$$

– Sequências retornadas:

$s_1 = \langle (pass, mc), (lbal, mc), (rec, mc), (cond, io), (drib, io) \rangle$;
$s_3 = \langle (rec, io), (cond, io), (bpas, io) \rangle$.

- **Instante 15:**

– Tabela hash $Z_{\#}$:

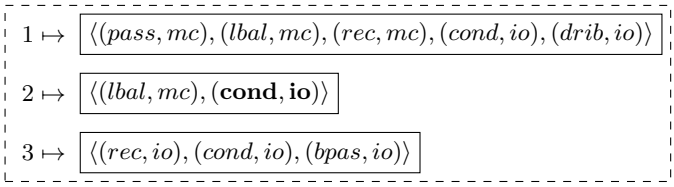
$$1 \mapsto \boxed{\langle (pass, mc), (lbal, mc), (rec, mc), (cond, io), (drib, io) \rangle}$$
$$2 \mapsto \boxed{\langle (\mathbf{lbal}, \mathbf{mc}) \rangle}$$
$$3 \mapsto \boxed{\langle (rec, io), (cond, io), (bpas, io) \rangle}$$

– Sequências retornadas:

$s_1 = \langle (pass, mc), (lbal, mc), (rec, mc), (cond, io), (drib, io) \rangle$;
$s_2 = \langle (lbal, mc) \rangle$;
$s_3 = \langle (rec, io), (cond, io), (bpas, io) \rangle$.

- **Instante 16:**

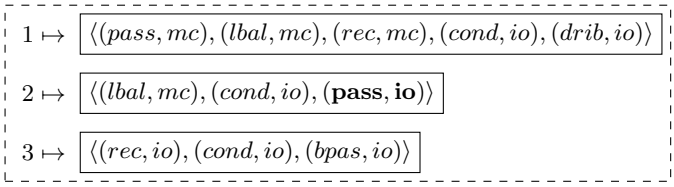
– Tabela hash $Z_{\#}$:



- Sequências retornadas:

$s_1 = \langle(pass, mc), (lbal, mc), (rec, mc), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

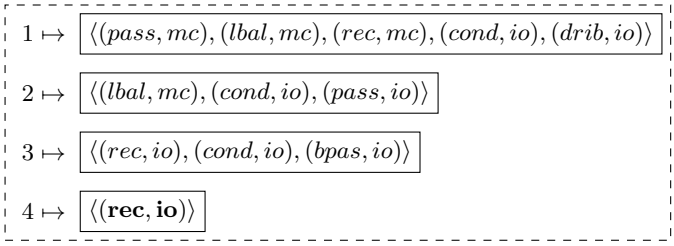
- **Instante 17:**
  - Tabela hash $Z_{\#}$:



  - Sequências retornadas:

$s_1 = \langle(pass, mc), (lbal, mc), (rec, mc), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

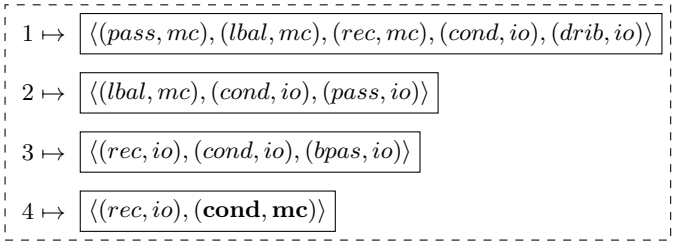
- **Instante 18:**
  - Tabela hash $Z_{\#}$:



  - Sequências retornadas:

$s_1 = \langle(pass, mc), (lbal, mc), (rec, mc), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s_4 = \langle(rec, io)\rangle$.

- **Instante 19:**
  - Tabela hash $Z_{\#}$:



– Sequências retornadas:

$s_1 = \langle(pass, mc), (lbal, mc), (rec, mc), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc)\rangle$.

- **Instante 20:**

– Tabela hash $Z_{\#}$:

$1 \mapsto \boxed{\langle(pass, mc), (lbal, mc), (rec, mc), (cond, io), (drib, io)\rangle}$
$2 \mapsto \boxed{\langle(lbal, mc), (cond, io), (pass, io)\rangle}$
$3 \mapsto \boxed{\langle(rec, io), (cond, io), (bpas, io)\rangle}$
$4 \mapsto \boxed{\langle(rec, io), (cond, mc), (\mathbf{cond}, \mathbf{ao})\rangle}$

– Sequências retornadas:

$s_1 = \langle(pass, mc), (lbal, mc), (rec, mc), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

- **Instante 21:**

– Tabela hash $Z_{\#}$:

$1 \mapsto \boxed{\langle\cancel{(pass, mc)}, (lbal, mc), (rec, mc), (cond, io), (drib, io), (\mathbf{cond}, \mathbf{io})\rangle}$
$2 \mapsto \boxed{\langle(lbal, mc), (cond, io), (pass, io)\rangle}$
$3 \mapsto \boxed{\langle(rec, io), (cond, io), (bpas, io)\rangle}$
$4 \mapsto \boxed{\langle(rec, io), (cond, mc), (cond, ao)\rangle}$

– Sequências retornadas:

$s_1 = \langle(lbal, mc), (rec, mc), (cond, io), (drib, io), (cond, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

- **Instante 22:**

– Tabela hash $Z_{\#}$:

$1 \mapsto \boxed{\langle\cancel{(lbal, mc)}, (rec, mc), (cond, io), (drib, io), (cond, io), (\mathbf{drib}, \mathbf{io})\rangle}$
$2 \mapsto \boxed{\langle(lbal, mc), (cond, io), (pass, io)\rangle}$
$3 \mapsto \boxed{\langle(rec, io), (cond, io), (bpas, io)\rangle}$
$4 \mapsto \boxed{\langle(rec, io), (cond, mc), (cond, ao)\rangle}$

– Sequências retornadas:

$s_1 = \langle(rec, mc), (cond, io), (drib, io), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

- **Instante 23:**

– Tabela hash $Z_\#$:

$1 \mapsto \boxed{\langle(rec, mc), (cond, io), (drib, io), (cond, io), (drib, io)\rangle}$
$2 \mapsto \boxed{\langle(lbal, mc), (cond, io), (pass, io), (\mathbf{lbal}, \mathbf{mc})\rangle}$
$3 \mapsto \boxed{\langle(rec, io), (cond, io), (bpas, io)\rangle}$
$4 \mapsto \boxed{\langle(rec, io), (cond, mc), (cond, ao)\rangle}$

– Sequências retornadas:

$s_1 = \langle(rec, mc), (cond, io), (drib, io), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (pass, io), (lbal, mc)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

- **Instante 24:**

– Tabela hash $Z_\#$:

$1 \mapsto \boxed{\langle(rec, mc), (cond, io), (drib, io), (cond, io), (drib, io)\rangle}$
$2 \mapsto \boxed{\langle(lbal, mc), (cond, io), (pass, io), (lbal, mc)\rangle}$
$3 \mapsto \boxed{\langle(rec, io), (cond, io), (bpas, io), (\mathbf{cond}, \mathbf{io})\rangle}$
$4 \mapsto \boxed{\langle(rec, io), (cond, mc), (cond, ao)\rangle}$

– Sequências retornadas:

$s_1 = \langle(rec, mc), (cond, io), (drib, io), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (pass, io), (lbal, mc)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io), (cond, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

- **Instante 25:**

– Tabela hash $Z_\#$:

$1 \mapsto \boxed{\langle(rec, mc), (cond, io), (drib, io), (cond, io), (drib, io)\rangle}$
$2 \mapsto \boxed{\langle(lbal, mc), (cond, io), (pass, io), (lbal, mc), (\mathbf{pass}, \mathbf{io})\rangle}$
$3 \mapsto \boxed{\langle(rec, io), (cond, io), (bpas, io), (cond, io)\rangle}$
$4 \mapsto \boxed{\langle(rec, io), (cond, mc), (cond, ao)\rangle}$

– Sequências retornadas:

$s_1 = \langle(rec, mc), (cond, io), (drib, io), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (pass, io), (lbal, mc), (pass, io)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io), (cond, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

B.2 Exemplo de execução do algoritmo *IncConseq*

Esta seção apresenta a execução do algoritmo *IncConseq* sobre as sequências retornadas pela operação $\mathbf{SEQ}_{\{\texttt{jid}\},20,1}(\texttt{evento})$ durante 25 iterações.

- **Instante 0:**
  - Sequências recebidas: nenhuma;
  - Tabela hash $L_{\#}$: vazia;
  - Sequências retornadas: nenhuma.
- **Instante 1:**
  - Sequências recebidas:

    $s_1 = \langle(\mathbf{pass}, \mathbf{mc})\rangle$.

  - Tabela hash $L_{\#}$:

    $s_1.key \mapsto \boxed{\langle(\mathbf{pass}, \mathbf{mc})\rangle}$

  - Sequências retornadas:

    $s_1 = \langle(pass, mc)\rangle$.
- **Instante 2:**
  - Sequências recebidas:

    $s_1 = \langle(pass, mc), (\mathbf{lbal}, \mathbf{mc})\rangle$.

  - Tabela hash $L_{\#}$:

    $s_1.key \mapsto \boxed{\langle(pass, mc), (\mathbf{lbal}, \mathbf{mc})\rangle}$

  - Sequências retornadas:

    $s_1 = \langle(pass, mc), (lbal, mc)\rangle$.
- **Instantes 3, 4 e 5:** Mesmo resultado do instante 2.
- **Instante 6:**
  - Sequências recebidas:

    $s_1 = \langle(pass, mc), (lbal, mc)\rangle$;
    $s_3 = \langle(\mathbf{rec}, \mathbf{io})\rangle$.

– Tabela hash $L_{\#}$:

$$s_1.key \mapsto \boxed{\langle (pass, mc), (lbal, mc) \rangle}$$
$$s_3.key \mapsto \boxed{\langle (\mathbf{rec}, \mathbf{io}) \rangle}$$

– Sequências retornadas:

$s_1 = \langle (pass, mc), (lbal, mc) \rangle$;
$s_3 = \langle (rec, io) \rangle$.

- **Instante 7:**

– Sequências recebidas:

$s_1 = \langle (pass, mc), (lbal, mc) \rangle$;
$s_3 = \langle (rec, io), (\mathbf{cond}, \mathbf{io}) \rangle$.

– Tabela hash $L_{\#}$:

$$s_1.key \mapsto \boxed{\langle (pass, mc), (lbal, mc) \rangle}$$
$$s_3.key \mapsto \boxed{\langle (rec, io), (\mathbf{cond}, \mathbf{io}) \rangle}$$

– Sequências retornadas:

$s_1 = \langle (pass, mc), (lbal, mc) \rangle$;
$s_3 = \langle (rec, io), (cond, io) \rangle$.

- **Instante 8:**

– Sequências recebidas:

$s_1 = \langle (pass, mc), (lbal, mc) \rangle$;
$s_3 = \langle (rec, io), (cond, io), (\mathbf{bpas}, \mathbf{io}) \rangle$.

– Tabela hash $L_{\#}$:

$$s_1.key \mapsto \boxed{\langle (pass, mc), (lbal, mc) \rangle}$$
$$s_3.key \mapsto \boxed{\langle (rec, io), (cond, io), (\mathbf{bpas}, \mathbf{io}) \rangle}$$

– Sequências retornadas:

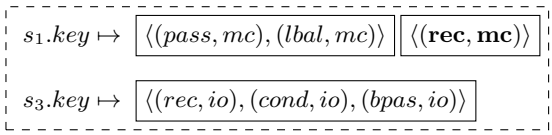
$s_1 = \langle (pass, mc), (lbal, mc) \rangle$;
$s_3 = \langle (rec, io), (cond, io), (bpas, io) \rangle$.

- **Instantes 9, 10 e 11:** Mesmo resultado do instante 8.
- **Instante 12:**

– Sequências recebidas:

$s_1 = \langle (pass, mc), (lbal, mc), (\mathbf{rec}, \mathbf{mc}) \rangle$;
$s_3 = \langle (rec, io), (cond, io), (bpas, io) \rangle$.

– Tabela hash $L_{\#}$:



– Sequências retornadas:

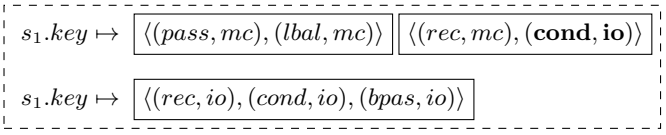
$s_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s_{1_b} = \langle(rec, mc)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

- **Instante 13:**

– Sequências recebidas:

$s_1 = \langle(pass, mc), (lbal, mc), (rec, mc), (\mathbf{cond}, \mathbf{io})\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

– Tabela hash $L_{\#}$:



– Sequências retornadas:

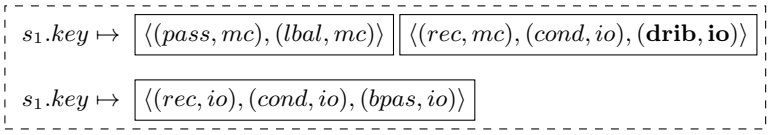
$s_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s_{1_b} = \langle(rec, mc), (cond, io)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

- **Instante 14:**

– Sequências recebidas:

$s_1 = \langle(pass, mc), (lbal, mc), (rec, mc), (cond, io), (\mathbf{drib}, \mathbf{io})\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

– Tabela hash $L_{\#}$:



– Sequências retornadas:

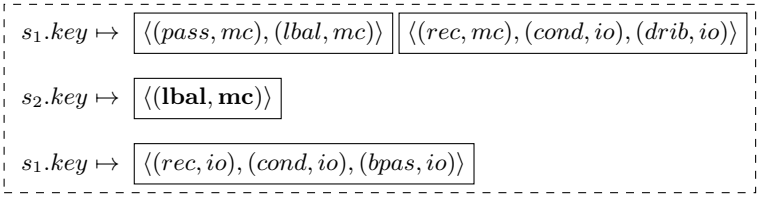
$s_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

- **Instante 15:**

– Sequências recebidas:

$s_1 = \langle(pass, mc), (lbal, mc), (rec, mc), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(\mathbf{lbal}, \mathbf{mc})\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

– Tabela hash $L_{\#}$:



– Sequências retornadas:

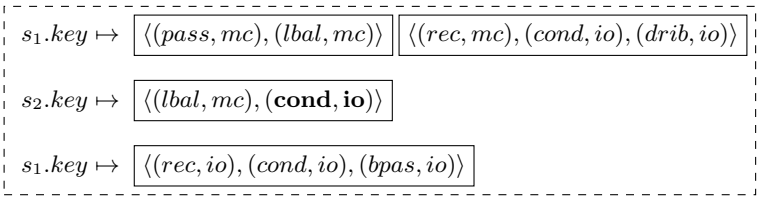
$s_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

- **Instante 16:**

– Sequências recebidas:

$s_1 = \langle(pass, mc), (lbal, mc), (rec, mc), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (\mathbf{cond}, \mathbf{io})\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

– Tabela hash $L_{\#}$:



– Sequências retornadas:

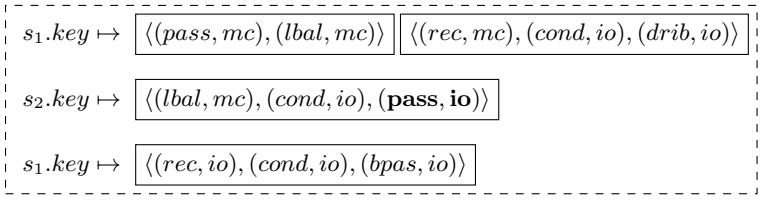
$s_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

- **Instante 17:**

– Sequências recebidas:

$s_1 = \langle(pass, mc), (lbal, mc), (rec, mc), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (\mathbf{pass}, \mathbf{io})\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

– Tabela hash $L_{\#}$:



– Sequências retornadas:

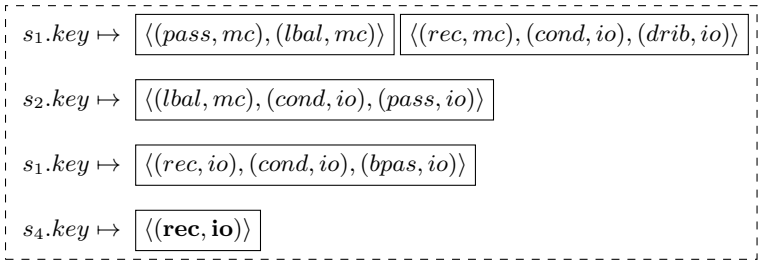
$s_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

- **Instante 18:**

  – Sequências recebidas:

  $s_1 = \langle(pass, mc), (lbal, mc), (rec, mc), (cond, io), (drib, io)\rangle$;
  $s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
  $s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
  $s_4 = \langle(\mathbf{rec}, \mathbf{io})\rangle$.

  – Tabela hash $L_\#$:



  – Sequências retornadas:

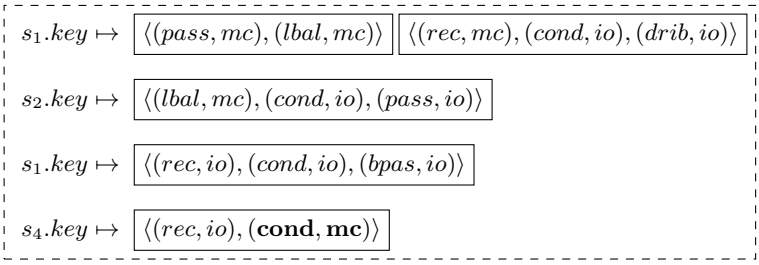
  $s_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
  $s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
  $s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
  $s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
  $s_4 = \langle(rec, io)\rangle$.

- **Instante 19:**

  – Sequências recebidas:

  $s_1 = \langle(pass, mc), (lbal, mc), (rec, mc), (cond, io), (drib, io)\rangle$;
  $s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
  $s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
  $s_4 = \langle(rec, io), (\mathbf{cond}, \mathbf{mc})\rangle$.

  – Tabela hash $L_\#$:



– Sequências retornadas:

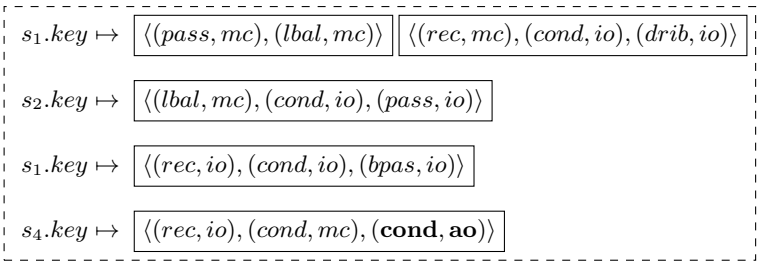
$s_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc)\rangle$.

- **Instante 20:**

– Sequências recebidas:

$s_1 = \langle(pass, mc), (lbal, mc), (rec, mc), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc), (\mathbf{cond}, \mathbf{ao})\rangle$.

– Tabela hash $L_{\#}$:



– Sequências retornadas:

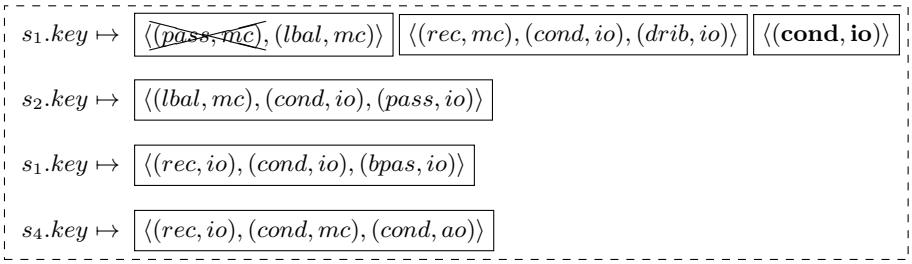
$s_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

- **Instante 21:**

– Sequências recebidas:

$s_1 = \langle\cancel{(pass, mc)}, (lbal, mc), (rec, mc), (cond, io), (drib, io), (\mathbf{cond}, \mathbf{io})\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

– Tabela hash $L_{\#}$:



– Sequências retornadas:

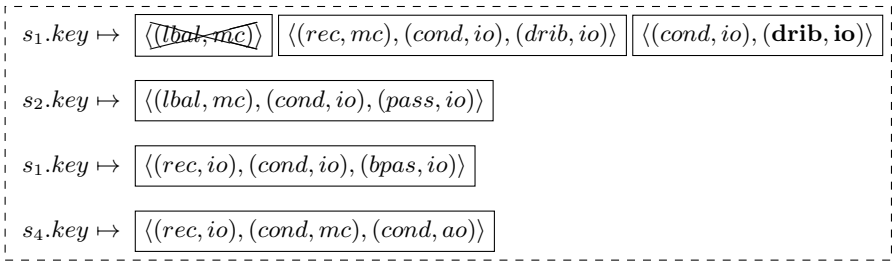
$s_{1_a} = \langle(lbal, mc)\rangle;$
$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s_{1_c} = \langle(cond, io)\rangle;$
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle;$
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle;$
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle.$

- **Instante 22:**

– Sequências recebidas:

$s_1 = \langle\cancel{(lbal, mc)}, (rec, mc), (cond, io), (drib, io), (cond, io), (\mathbf{drib}, \mathbf{io})\rangle;$
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle;$
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle;$
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle.$

– Tabela hash $L_{\#}$:



– Sequências retornadas:

$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s_{1_c} = \langle(cond, io), (drib, io)\rangle;$
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle;$
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle;$
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle.$

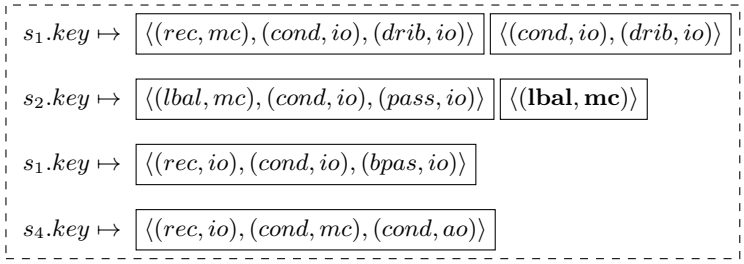
- **Instante 23:**

– Sequências recebidas:

$s_1 = \langle(rec, mc), (cond, io), (drib, io), (cond, io), (drib, io)\rangle;$
$s_2 = \langle(lbal, mc), (cond, io), (pass, io), (\mathbf{lbal}, \mathbf{mc})\rangle;$

$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

– Tabela hash $L_{\#}$:



– Sequências retornadas:

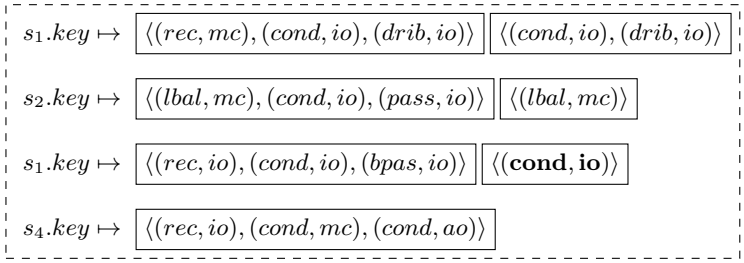
$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s_{1_c} = \langle(cond, io), (drib, io)\rangle$;
$s_{2_a} = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s_{2_b} = \langle(lbal, mc)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

- **Instante 24:**

– Sequências recebidas:

$s_1 = \langle(rec, mc), (cond, io), (drib, io), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (pass, io), (lbal, mc)\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io), (\mathbf{cond}, \mathbf{io})\rangle$;
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

– Tabela hash $L_{\#}$:



– Sequências retornadas:

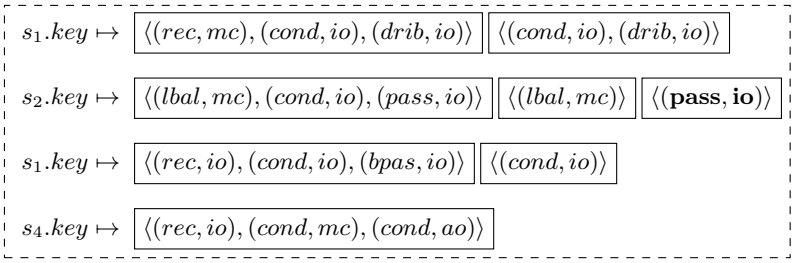
$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s_{1_c} = \langle(cond, io), (drib, io)\rangle$;
$s_{2_a} = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s_{2_b} = \langle(lbal, mc)\rangle$;
$s_{3_a} = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s_{3_b} = \langle(cond, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

- **Instante 25:**

– Sequências recebidas:

$s_1 = \langle(rec, mc), (cond, io), (drib, io), (cond, io), (drib, io)\rangle$;
$s_2 = \langle(lbal, mc), (cond, io), (pass, io), (lbal, mc), (\mathbf{pass}, \mathbf{io})\rangle$;
$s_3 = \langle(rec, io), (cond, io), (bpas, io), (cond, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

– Tabela hash $L_{\#}$:



– Sequências retornadas:

$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s_{1_c} = \langle(cond, io), (drib, io)\rangle$;
$s_{2_a} = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s_{2_b} = \langle(lbal, mc)\rangle$;
$s_{2_c} = \langle(pass, io)\rangle$;
$s_{3_a} = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s_{3_b} = \langle(cond, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

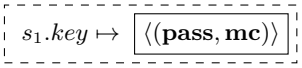
B.3 Exemplo de execução do algoritmo *IncEndseq*

Esta seção apresenta a execução do algoritmo *IncEndseq* sobre as sequências retornadas pela operação **CONSEQ**(**SEQ**$_{\{\texttt{jid}\},20,1}$(`evento`)) durante 25 iterações.

- **Instante 0:**
  - Sequências recebidas: nenhuma;
  - Tabela hash $L_{\#}$: vazia;
  - Sequências retornadas: nenhuma.
- **Instante 1:**
  - Sequências recebidas:

    $s_1 = \langle(\mathbf{pass}, \mathbf{mc})\rangle$.

  - Tabela hash $L_{\#}$:



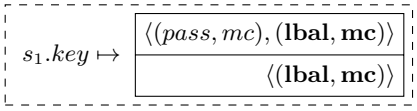
  - Sequências retornadas:

$s'_1 = \langle(pass, mc)\rangle$.

- **Instante 2:**
  - Sequências recebidas:

    $s_1 = \langle(pass, mc), (\mathbf{lbal}, \mathbf{mc})\rangle$.

  - Tabela hash $L_\#$:



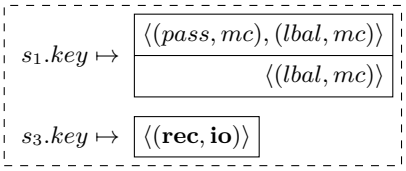
  - Sequências retornadas:

    $s'_1 = \langle(lbal, mc)\rangle$;
    $s''_1 = \langle(pass, mc), (lbal, mc)\rangle$.

- **Instantes 3, 4 e 5:** Mesmo resultado do instante 2.
- **Instante 6:**
  - Sequências recebidas:

    $s_1 = \langle(pass, mc), (lbal, mc)\rangle$;
    $s_3 = \langle(\mathbf{rec}, \mathbf{io})\rangle$.

  - Tabela hash $L_\#$:



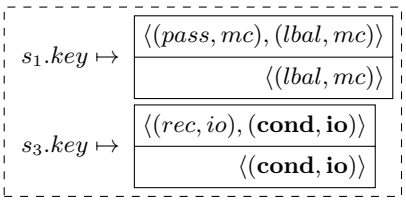
  - Sequências retornadas:

    $s'_1 = \langle(lbal, mc)\rangle$;
    $s''_1 = \langle(pass, mc), (lbal, mc)\rangle$;
    $s'_3 = \langle(rec, io)\rangle$.

- **Instante 7:**
  - Sequências recebidas:

    $s_1 = \langle(pass, mc), (lbal, mc)\rangle$;
    $s_3 = \langle(rec, io), (\mathbf{cond}, \mathbf{io})\rangle$.

  - Tabela hash $L_\#$:



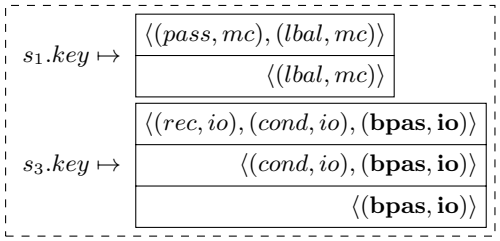
  - Sequências retornadas:

$s'_1 = \langle(lbal, mc)\rangle$;
$s''_1 = \langle(pass, mc), (lbal, mc)\rangle$;
$s'_3 = \langle(cond, io)\rangle$;
$s''_3 = \langle(rec, io), (cond, io)\rangle$.

- **Instante 8:**
  - Sequências recebidas:

    $s_1 = \langle(pass, mc), (lbal, mc)\rangle$;
    $s_3 = \langle(rec, io), (cond, io), (\mathbf{bpas}, \mathbf{io})\rangle$.

  - Tabela hash $L_\#$:



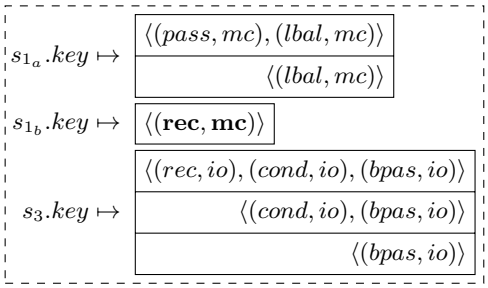
  - Sequências retornadas:

    $s'_1 = \langle(lbal, mc)\rangle$;
    $s''_1 = \langle(pass, mc), (lbal, mc)\rangle$;
    $s'_3 = \langle(bpas, io)\rangle$;
    $s''_3 = \langle(cond, io), (bpas, io)\rangle$.
    $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

- **Instantes 9, 10 e 11:** Mesmo resultado do instante 8.
- **Instante 12:**
  - Sequências recebidas:

    $s_{1_a} = \langle(pass, mc), (lbal, mc), \rangle$;
    $s_{1_b} = \langle(\mathbf{rec}, \mathbf{mc})\rangle$;
    $s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

  - Tabela hash $L_\#$:



  - Sequências retornadas:

    $s'_{1_a} = \langle(lbal, mc)\rangle$;
    $s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;

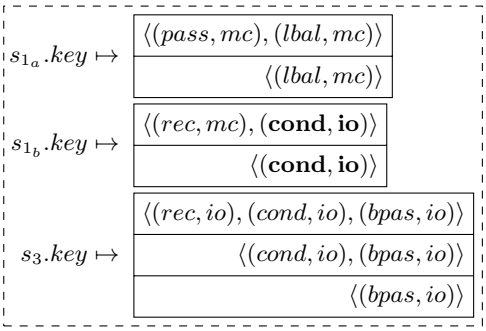
$s'_{1_b} = \langle(rec, mc)\rangle$;
$s'_3 = \langle(bpas, io)\rangle$;
$s''_3 = \langle(cond, io), (bpas, io)\rangle$.
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

- **Instante 13:**

  – Sequências recebidas:

  $s_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
  $s_{1_b} = \langle(rec, mc), (\mathbf{cond}, \mathbf{io})\rangle$;
  $s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

  – Tabela hash $L_\#$:



  – Sequências retornadas:

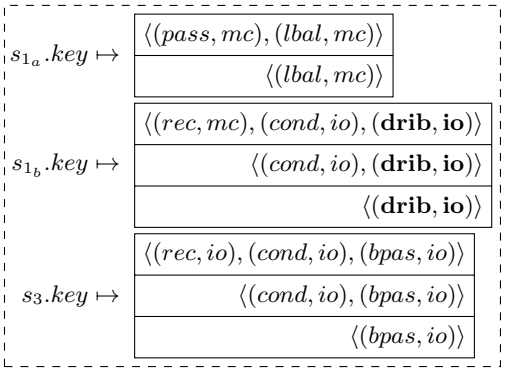
  $s'_{1_a} = \langle(lbal, mc)\rangle$;
  $s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
  $s'_{1_b} = \langle(cond, io)\rangle$;
  $s''_{1_b} = \langle(rec, mc), (cond, io)\rangle$;
  $s'_3 = \langle(bpas, io)\rangle$;
  $s''_3 = \langle(cond, io), (bpas, io)\rangle$.
  $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

- **Instante 14:**

  – Sequências recebidas:

  $s_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
  $s_{1_b} = \langle(rec, mc), (cond, io), (\mathbf{drib}, \mathbf{io})\rangle$;
  $s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$.

  – Tabela hash $L_\#$:



– Sequências retornadas:

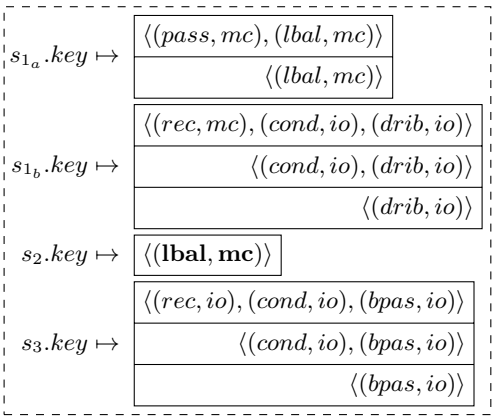
$s'_{1_a} = \langle(lbal, mc)\rangle;$
$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle;$
$s'_{1_b} = \langle(drib, io)\rangle;$
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle;$
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s'_3 = \langle(bpas, io)\rangle;$
$s''_3 = \langle(cond, io), (bpas, io)\rangle.$
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle.$

- **Instante 15:**

– Sequências recebidas:

$s_{1_a} = \langle(pass, mc), (lbal, mc)\rangle;$
$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s_2 = \langle(\mathbf{lbal}, \mathbf{mc})\rangle;$
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle.$

– Tabela hash $L_\#$:



– Sequências retornadas:

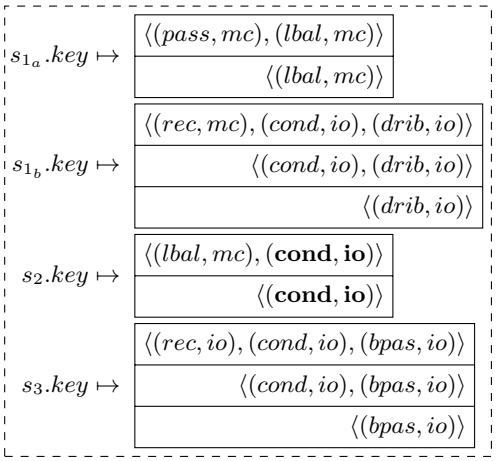
$s'_{1_a} = \langle(lbal, mc)\rangle;$
$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle;$
$s'_{1_b} = \langle(drib, io)\rangle;$
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle;$

$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s'_2 = \langle(lbal, mc)\rangle;$
$s'_3 = \langle(bpas, io)\rangle;$
$s''_3 = \langle(cond, io), (bpas, io)\rangle.$
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle.$

- **Instante 16:**
  - Sequências recebidas:

    $s_{1_a} = \langle(pass, mc), (lbal, mc)\rangle;$
    $s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
    $s_2 = \langle(lbal, mc), (\mathbf{cond}, \mathbf{io})\rangle;$
    $s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle.$

  - Tabela hash $L_{\#}$:



  - Sequências retornadas:

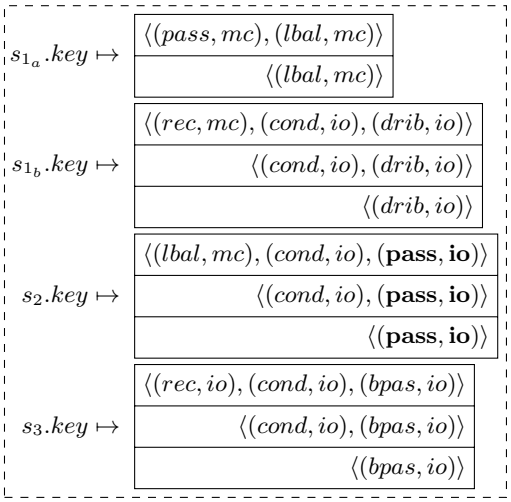
    $s'_{1_a} = \langle(lbal, mc)\rangle;$
    $s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle;$
    $s'_{1_b} = \langle(drib, io)\rangle;$
    $s''_{1_b} = \langle(cond, io), (drib, io)\rangle;$
    $s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
    $s'_2 = \langle(cond, io)\rangle;$
    $s''_2 = \langle(lbal, mc), (cond, io)\rangle;$
    $s'_3 = \langle(bpas, io)\rangle;$
    $s''_3 = \langle(cond, io), (bpas, io)\rangle.$
    $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle.$

- **Instante 17:**
  - Sequências recebidas:

    $s_{1_a} = \langle(pass, mc), (lbal, mc)\rangle;$
    $s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
    $s_2 = \langle(lbal, mc), (cond, io), (\mathbf{pass}, \mathbf{io})\rangle;$

$s_3 = \langle (rec, io), (cond, io), (bpas, io) \rangle$.

– Tabela hash $L_\#$:



– Sequências retornadas:

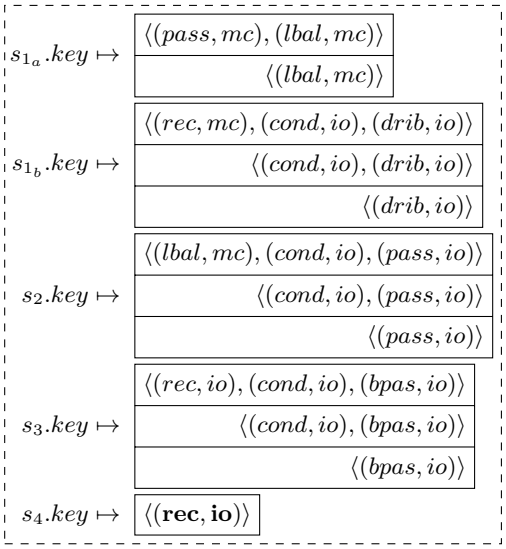
$s'_{1_a} = \langle (lbal, mc) \rangle$;
$s''_{1_a} = \langle (pass, mc), (lbal, mc) \rangle$;
$s'_{1_b} = \langle (drib, io) \rangle$;
$s''_{1_b} = \langle (cond, io), (drib, io) \rangle$;
$s'''_{1_b} = \langle (rec, mc), (cond, io), (drib, io) \rangle$;
$s'_2 = \langle (pass, io) \rangle$;
$s''_2 = \langle (cond, io), (pass, io) \rangle$;
$s'''_2 = \langle (lbal, mc), (cond, io), (pass, io) \rangle$;
$s'_3 = \langle (bpas, io) \rangle$;
$s''_3 = \langle (cond, io), (bpas, io) \rangle$.
$s'''_3 = \langle (rec, io), (cond, io), (bpas, io) \rangle$.

- **Instante 18:**

– Sequências recebidas:

$s_{1_a} = \langle (pass, mc), (lbal, mc) \rangle$;
$s_{1_b} = \langle (rec, mc), (cond, io), (drib, io) \rangle$;
$s_2 = \langle (lbal, mc), (cond, io), (pass, io) \rangle$;
$s_3 = \langle (rec, io), (cond, io), (bpas, io) \rangle$;
$s_4 = \langle (\mathbf{rec}, \mathbf{io}) \rangle$.

– Tabela hash $L_\#$:



– Sequências retornadas:

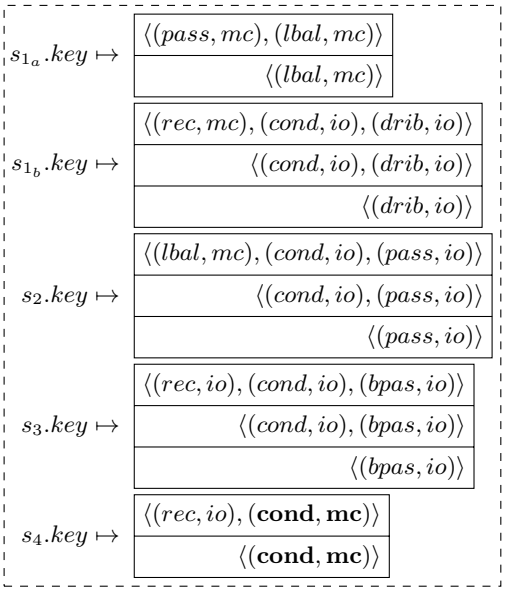
$s'_{1_a} = \langle(lbal, mc)\rangle;$
$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle;$
$s'_{1_b} = \langle(drib, io)\rangle;$
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle;$
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s'_2 = \langle(pass, io)\rangle;$
$s''_2 = \langle(cond, io), (pass, io)\rangle;$
$s'''_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle;$
$s'_3 = \langle(bpas, io)\rangle;$
$s''_3 = \langle(cond, io), (bpas, io)\rangle.$
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle.$
$s'_4 = \langle(rec, io)\rangle.$

- **Instante 19:**

– Sequências recebidas:

$s_{1_a} = \langle(pass, mc), (lbal, mc)\rangle;$
$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle;$
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle;$
$s_4 = \langle(rec, io), (\mathbf{cond}, \mathbf{mc})\rangle.$

– Tabela hash $L_{\#}$:



– Sequências retornadas:

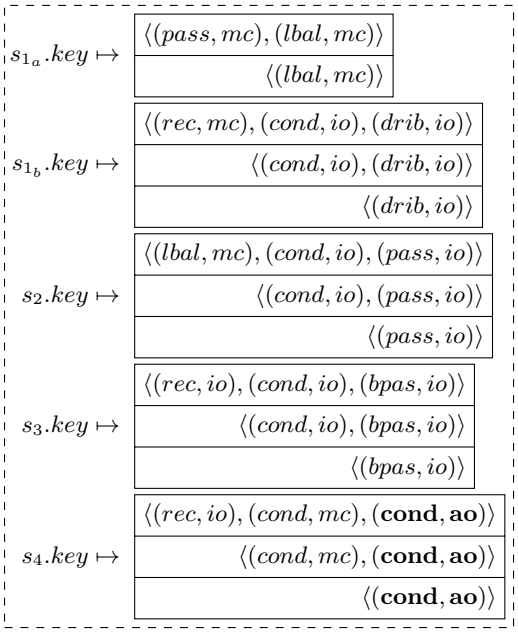
$s'_{1_a} = \langle(lbal, mc)\rangle;$
$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle;$
$s'_{1_b} = \langle(drib, io)\rangle;$
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle;$
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s'_2 = \langle(pass, io)\rangle;$
$s''_2 = \langle(cond, io), (pass, io)\rangle;$
$s'''_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle;$
$s'_3 = \langle(bpas, io)\rangle;$
$s''_3 = \langle(cond, io), (bpas, io)\rangle.$
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle.$
$s'_4 = \langle(cond, mc)\rangle;$
$s''_4 = \langle(rec, io), (cond, mc)\rangle.$

- **Instante 20:**

– Sequências recebidas:

$s_{1_a} = \langle(pass, mc), (lbal, mc)\rangle;$
$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle;$
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle;$
$s_4 = \langle(rec, io), (cond, mc), (\mathbf{cond}, \mathbf{ao})\rangle.$

– Tabela hash $L_\#$:



– Sequências retornadas:

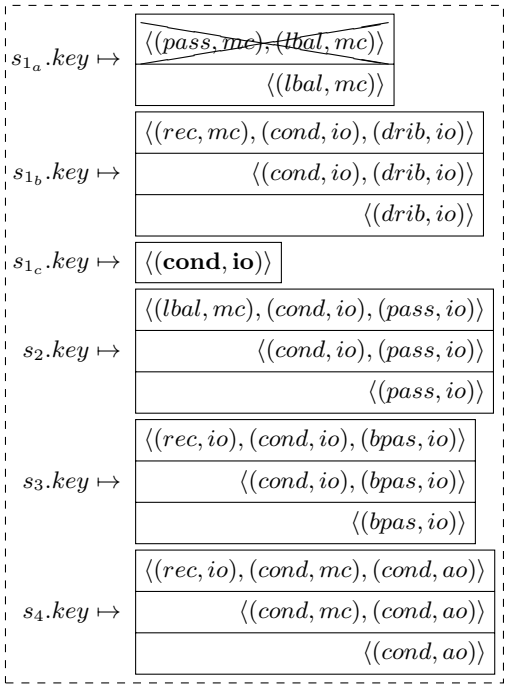
$s'_{1_a} = \langle(lbal, mc)\rangle;$
$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle;$
$s'_{1_b} = \langle(drib, io)\rangle;$
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle;$
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s'_2 = \langle(pass, io)\rangle;$
$s''_2 = \langle(cond, io), (pass, io)\rangle;$
$s'''_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle;$
$s'_3 = \langle(rec, io)\rangle;$
$s''_3 = \langle(rec, io), (cond, io)\rangle;$
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle;$
$s'_4 = \langle(cond, ao)\rangle;$
$s''_4 = \langle(cond, mc), (cond, ao)\rangle;$
$s'''_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle.$

- **Instante 21:**

– Sequências recebidas:

$s_{1_a} = \langle\cancel{(pass, mc)}, (lbal, mc)\rangle;$
$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s_{1_c} = \langle(\mathbf{cond}, \mathbf{io})\rangle;$
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle;$
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle;$
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle.$

– Tabela hash $L_{\#}$:



– Sequências retornadas:

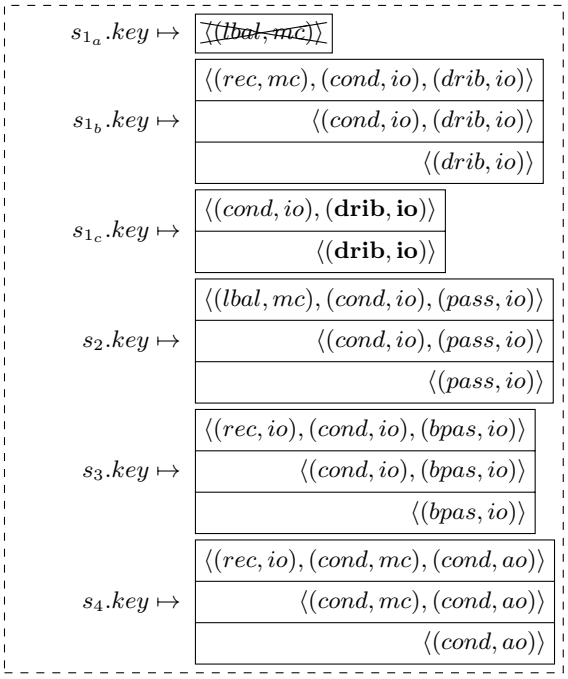
$s'_{1_a} = \langle(lbal, mc)\rangle;$
$s'_{1_b} = \langle(drib, io)\rangle;$
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle;$
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s'_{1_c} = \langle(cond, io)\rangle;$
$s'_2 = \langle(pass, io)\rangle;$
$s''_2 = \langle(cond, io), (pass, io)\rangle;$
$s'''_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle;$
$s'_3 = \langle(bpas, io)\rangle;$
$s''_3 = \langle(cond, io), (bpas, io)\rangle.$
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle.$
$s'_4 = \langle(cond, ao)\rangle;$
$s''_4 = \langle(cond, mc), (cond, ao)\rangle;$
$s'''_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle.$

- **Instante 22:**

– Sequências recebidas:

$s_{1_a} = \langle\cancel{(lbal, mc)}\rangle;$
$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s_{1_c} = \langle(cond, io), (\mathbf{drib}, \mathbf{io})\rangle;$
$s_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle;$
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle;$
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle.$

– Tabela hash $L_\#$:



– Sequências retornadas:

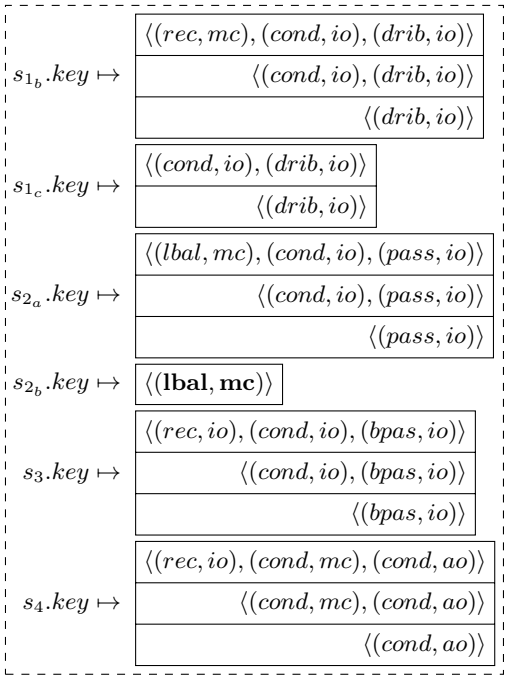
$s'_{1_b} = \langle(drib, io)\rangle;$
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle;$
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s'_{1_c} = \langle(drib, io)\rangle;$
$s''_{1_c} = \langle(cond, io), (drib, io)\rangle;$
$s'_2 = \langle(pass, io)\rangle;$
$s''_2 = \langle(cond, io), (pass, io)\rangle;$
$s'''_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle;$
$s'_3 = \langle(bpas, io)\rangle;$
$s''_3 = \langle(cond, io), (bpas, io)\rangle.$
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle.$
$s'_4 = \langle(cond, ao)\rangle;$
$s''_4 = \langle(cond, mc), (cond, ao)\rangle;$
$s'''_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle.$

- **Instante 23:**

– Sequências recebidas:

$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s_{1_c} = \langle(cond, io), (drib, io)\rangle;$
$s_{2_a} = \langle(lbal, mc), (cond, io), (pass, io)\rangle;$
$s_{2_b} = \langle(\mathbf{lbal}, \mathbf{mc})\rangle;$
$s_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle;$
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle.$

– Tabela hash $L_{\#}$:



– Sequências retornadas:

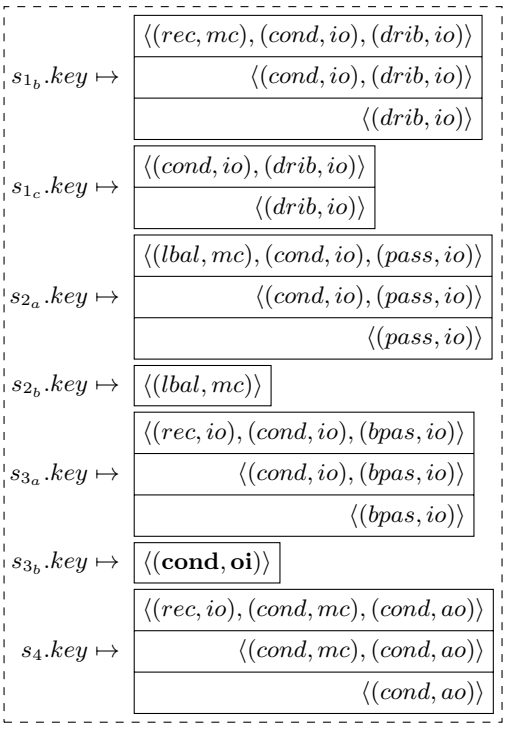
$s'_{1_b} = \langle(drib, io)\rangle;$
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle;$
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s'_{1_c} = \langle(drib, io)\rangle;$
$s''_{1_c} = \langle(cond, io), (drib, io)\rangle;$
$s'_{2_a} = \langle(pass, io)\rangle;$
$s''_{2_a} = \langle(cond, io), (pass, io)\rangle;$
$s'''_{2_a} = \langle(lbal, mc), (cond, io), (pass, io)\rangle;$
$s'_{2_b} = \langle(lbal, mc)\rangle;$
$s'_3 = \langle(bpas, io)\rangle;$
$s''_3 = \langle(cond, io), (bpas, io)\rangle.$
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle.$
$s'_4 = \langle(cond, ao)\rangle;$
$s''_4 = \langle(cond, mc), (cond, ao)\rangle;$
$s'''_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle.$

- **Instante 24:**

– Sequências recebidas:

$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle;$
$s_{1_c} = \langle(cond, io), (drib, io)\rangle;$
$s_{2_a} = \langle(lbal, mc), (cond, io), (pass, io)\rangle;$
$s_{2_b} = \langle(lbal, mc)\rangle;$
$s_{3_a} = \langle(rec, io), (cond, io), (bpas, io)\rangle;$
$s_{3_b} = \langle(\mathbf{cond}, \mathbf{io})\rangle;$
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle.$

– Tabela hash $L_{\#}$:



– Sequências retornadas:

$s'_{1_b} = \langle(drib, io)\rangle$;
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s'_{1_c} = \langle(drib, io)\rangle$;
$s''_{1_c} = \langle(cond, io), (drib, io)\rangle$;
$s'_{2_a} = \langle(pass, io)\rangle$;
$s''_{2_a} = \langle(cond, io), (pass, io)\rangle$;
$s'''_{2_a} = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s'_{2_b} = \langle(lbal, mc)\rangle$;
$s'_{3_a} = \langle(bpas, io)\rangle$;
$s''_{3_a} = \langle(cond, io), (bpas, io)\rangle$;
$s'''_{3_a} = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s'_{3_b} = \langle(cond, io)\rangle$;
$s'_4 = \langle(cond, ao)\rangle$;
$s''_4 = \langle(cond, mc), (cond, ao)\rangle$;
$s'''_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

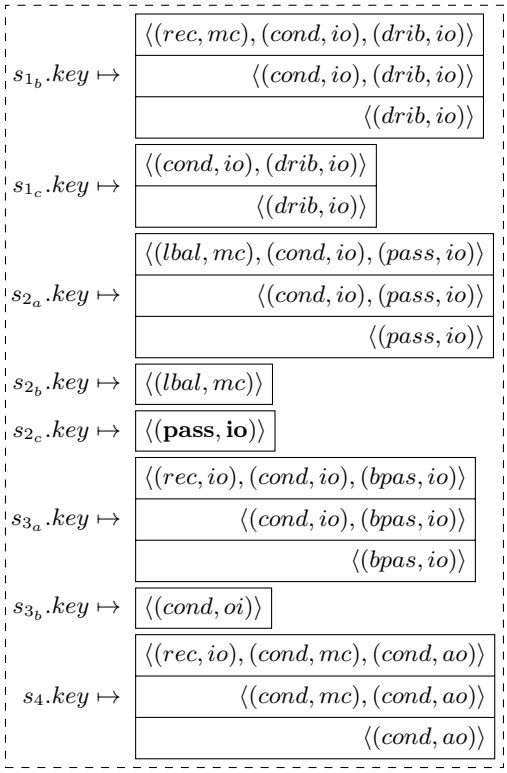
- **Instante 25:**

– Sequências recebidas:

$s_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s_{1_c} = \langle(cond, io), (drib, io)\rangle$;
$s_{2_a} = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s_{2_b} = \langle(lbal, mc)\rangle$;

$s_{2_c} = \langle(\mathbf{pass}, \mathbf{io})\rangle$;
$s_{3_a} = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s_{3_b} = \langle(cond, io)\rangle$;
$s_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

– Tabela hash $L_{\#}$:



– Sequências retornadas:

$s'_{1_b} = \langle(drib, io)\rangle$;
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s'_{1_c} = \langle(drib, io)\rangle$;
$s''_{1_c} = \langle(cond, io), (drib, io)\rangle$;
$s'_{2_a} = \langle(pass, io)\rangle$;
$s''_{2_a} = \langle(cond, io), (pass, io)\rangle$;
$s'''_{2_a} = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s'_{2_b} = \langle(lbal, mc)\rangle$;
$s'_{2_c} = \langle(pass, io)\rangle$;
$s'_{3_a} = \langle(bpas, io)\rangle$;
$s''_{3_a} = \langle(cond, io), (bpas, io)\rangle$;
$s'''_{3_a} = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s'_{3_b} = \langle(cond, io)\rangle$;
$s'_4 = \langle(cond, ao)\rangle$;
$s''_4 = \langle(cond, mc), (cond, ao)\rangle$;
$s'''_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

B.4 Exemplo de execução do algoritmo *GetBestSeq*

Esta seção apresenta a execução do algoritmo *GetBestSeq* sobre as sequências retornadas pela operação $\mathbf{MINSEQ}_2(\mathbf{ENDSEQ}(\mathbf{CONSEQ}(\mathbf{SEQ}_{\{\texttt{jid}\},20,1}(\texttt{evento}))))$ durante 25 iterações. A teoria-pct usada pelo algoritmo é $\Phi' = \{\varphi_4, \varphi_5, \varphi_6, \varphi_7\}$, onde:

φ_4 : $\mathbf{First} \rightarrow (\texttt{jogada} = rec) \succ (\texttt{jogada} = lbal)$;
φ_5 : $\mathbf{Prev}(\texttt{jogada} = cond) \rightarrow (\texttt{jogada} = drib) \succ (\texttt{jogada} = pass)[\texttt{local}]$;
φ_6 : $\rightarrow (\texttt{jogada} = pass) \succ (\texttt{jogada} = bpas)[\texttt{local}]$;
φ_7 : $\mathbf{AllPrev}(\texttt{local} = io) \rightarrow (\texttt{local} = io) \succ (\texttt{local} = mc)$.

- **Instante 0 e 1:**
  - Sequências recebidas: nenhuma;
  - Comparações: nenhuma;
  - Sequências retornadas: nenhuma.
- **Instante 2:**
  - Sequências recebidas:

    $s_1'' = \langle (pass, mc), (lbal, mc) \rangle$;
  - Comparações: nenhuma;
  - Sequências retornadas:

    $s_1'' = \langle (pass, mc), (lbal, mc) \rangle$.
- **Instantes 3, 4, 5 e 6:** Mesmo resultado do instante 2.
- **Instante 7:**
  - Sequências recebidas:

    $s_1'' = \langle (pass, mc), (lbal, mc) \rangle$;
    $s_3'' = \langle (rec, io), (cond, io) \rangle$;
  - Comparações: nenhuma;
  - Sequências retornadas:

    $s_1'' = \langle (pass, mc), (lbal, mc) \rangle$;
    $s_3'' = \langle (rec, io), (cond, io) \rangle$;
- **Instante 8:**
  - Sequências recebidas:

    $s_1'' = \langle (pass, mc), (lbal, mc) \rangle$;
    $s_3'' = \langle (cond, io), (bpas, io) \rangle$;
    $s_3''' = \langle (rec, io), (cond, io), (bpas, io) \rangle$;
  - Comparações: nenhuma;
  - Sequências retornadas:

    $s_1'' = \langle (pass, mc), (lbal, mc) \rangle$;

$s''_3 = \langle(cond, io), (bpas, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

- **Instantes 9, 10, 11 e 12:** Mesmo resultado do instante 8.
- **Instante 13:**
  - Sequências recebidas:

    $s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
    $s''_{1_b} = \langle(rec, mc), (cond, io)\rangle$;
    $s''_3 = \langle(cond, io), (bpas, io)\rangle$;
    $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

  - Comparações: nenhuma;
  - Sequências retornadas:

    $s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
    $s''_{1_b} = \langle(rec, mc), (cond, io)\rangle$;
    $s''_3 = \langle(cond, io), (bpas, io)\rangle$;
    $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

- **Instante 14:**
  - Sequências recebidas:

    $s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
    $s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
    $s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
    $s''_3 = \langle(cond, io), (bpas, io)\rangle$;
    $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

  - Comparações: $s''_{1_b} \succ_{\varphi_5} ... \succ_{\varphi_6} s''_3$;
  - Sequências retornadas:

    $s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
    $s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
    $s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
    $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

- **Instante 15:** Mesmo resultado do instante 14.
- **Instante 16:**
  - Sequências recebidas:

    $s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
    $s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
    $s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
    $s''_2 = \langle(lbal, mc), (cond, io)\rangle$;
    $s''_3 = \langle(cond, io), (bpas, io)\rangle$;
    $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

– Comparações: $s''_{1_b} \succ_{\varphi_5} ... \succ_{\varphi_6} s''_3$, $s'''_{1_b} \succ_{\varphi_4} s''_2$;
– Sequências retornadas:

$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

- **Instante 17:**

– Sequências recebidas:

$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s''_2 = \langle(cond, io), (pass, io)\rangle$;
$s'''_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s''_3 = \langle(cond, io), (bpas, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

– Comparações: $s'''_{1_b} \succ_{\varphi_4} s'''_2$, $s''_{1_b} \succ_{\varphi_5} s''_2$, $s''_{1_b} \succ_{\varphi_5} ... \succ_{\varphi_6} s''_3$, $s''_2 \succ_{\varphi_6} s''_3$;
– Sequências retornadas:

$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

- **Instante 18:** Mesmo resultado do instante 17.
- **Instante 19:**

– Sequências recebidas:

$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s''_2 = \langle(cond, io), (pass, io)\rangle$;
$s'''_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s''_3 = \langle(cond, io), (bpas, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s''_4 = \langle(rec, io), (cond, mc)\rangle$.

– Comparações: $s'''_{1_b} \succ_{\varphi_4} s'''_2$, $s''_{1_b} \succ_{\varphi_5} s''_2$, $s''_{1_b} \succ_{\varphi_5} ... \succ_{\varphi_6} s''_3$, $s''_2 \succ_{\varphi_6} s''_3$, $s'''_3 \succ_{\varphi_7} s''_4$;
– Sequências retornadas:

$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

- **Instante 20:**
  - Sequências recebidas:

    $s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
    $s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
    $s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
    $s''_2 = \langle(cond, io), (pass, io)\rangle$;
    $s'''_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
    $s''_3 = \langle(cond, io), (bpas, io)\rangle$;
    $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
    $s''_4 = \langle(cond, mc), (cond, ao)\rangle$;
    $s'''_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.
  - Comparações: $s'''_{1_b} \succ_{\varphi_4} s'''_2$, $s''_{1_b} \succ_{\varphi_5} s''_2$, $s''_{1_b} \succ_{\varphi_5} ... \succ_{\varphi_6} s''_3$, $s''_2 \succ_{\varphi_6} s''_3$, $s'''_3 \succ_{\varphi_7} s'''_4$;
  - Sequências retornadas:

    $s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
    $s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
    $s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
    $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
    $s''_4 = \langle(cond, mc), (cond, ao)\rangle$;

- **Instante 21:**
  - Sequências recebidas:

    $s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
    $s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
    $s''_2 = \langle(cond, io), (pass, io)\rangle$;
    $s'''_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
    $s''_3 = \langle(cond, io), (bpas, io)\rangle$;
    $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
    $s''_4 = \langle(cond, mc), (cond, ao)\rangle$;
    $s'''_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.
  - Comparações: $s'''_{1_b} \succ_{\varphi_4} s'''_2$, $s''_{1_b} \succ_{\varphi_5} s''_2$, $s''_{1_b} \succ_{\varphi_5} ... \succ_{\varphi_6} s''_3$, $s''_2 \succ_{\varphi_6} s''_3$, $s'''_3 \succ_{\varphi_7} s'''_4$;
  - Sequências retornadas:

    $s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
    $s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
    $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
    $s''_4 = \langle(cond, mc), (cond, ao)\rangle$;

- **Instante 22:**
  - Sequências recebidas:

    $s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
    $s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;

$s''_{1_c} = \langle(cond, io), (drib, io)\rangle$;
$s''_2 = \langle(cond, io), (pass, io)\rangle$;
$s'''_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s''_3 = \langle(cond, io), (bpas, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s''_4 = \langle(cond, mc), (cond, ao)\rangle$;
$s'''_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

– Comparações: $s'''_{1_b} \succ_{\varphi_4} s'''_2$, $s''_{1_b} \succ_{\varphi_5} s''_2$, $s''_{1_b} \succ_{\varphi_5} ... \succ_{\varphi_6} s''_3$, $s''_{1_c} \succ_{\varphi_5} s''_2$, $s''_{1_c} \succ_{\varphi_5} ... \succ_{\varphi_6} s''_3$, $s''_2 \succ_{\varphi_6} s''_3$, $s'''_3 \succ_{\varphi_7} s'''_4$;
– Sequências retornadas:

$s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s''_{1_c} = \langle(cond, io), (drib, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s''_4 = \langle(cond, mc), (cond, ao)\rangle$;

- **Instantes 23, 24 e 25:** Mesmo resultado do instante 22.

B.5 Exemplo de execução do algoritmo *IncBestSeq*

Esta seção apresenta a execução do algoritmo *IncBestSeq* sobre as sequências retornadas pela operação $\mathbf{MINSEQ}_2(\mathbf{ENDSEQ}(\mathbf{CONSEQ}(\mathbf{SEQ}_{\{\texttt{jid}\},20,1}(\texttt{evento}))))$ durante 25 iterações. A teoria-pct usada pelo algoritmo é $\Phi' = \{\varphi_4, \varphi_5, \varphi_6, \varphi_7\}$, onde:

φ_4 : $\mathbf{First} \rightarrow (\texttt{jogada} = rec) \succ (\texttt{jogada} = lbal)$;
φ_5 : $\mathbf{Prev}(\texttt{jogada} = cond) \rightarrow (\texttt{jogada} = drib) \succ (\texttt{jogada} = pass)[\texttt{local}]$;
φ_6 : $\rightarrow (\texttt{jogada} = pass) \succ (\texttt{jogada} = bpas)[\texttt{local}]$;
φ_7 : $\mathbf{AllPrev}(\texttt{local} = io) \rightarrow (\texttt{local} = io) \succ (\texttt{local} = mc)$.

Os nós dominantes da árvore de sequências mantida pelo algoritmo estão representados na cor cinza.

- **Instante 0 e 1:**

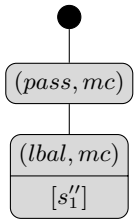
  – Sequências recebidas: nenhuma;
  – Árvore de sequências: vazia;
  – Sequências retornadas: nenhuma.

- **Instante 2:**

  – Sequências recebidas:

  $s''_1 = \langle(pass, mc), (lbal, mc)\rangle$;

  – Árvore de sequências:



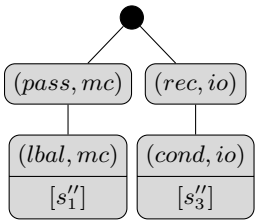
– Sequências retornadas:

$s_1'' = \langle (pass, mc), (lbal, mc) \rangle$.

- **Instantes 3, 4, 5 e 6:** Mesmo resultado do instante 2.
- **Instante 7:**
  - Sequências recebidas:

    $s_1'' = \langle (pass, mc), (lbal, mc) \rangle$;
    $s_3'' = \langle (rec, io), (cond, io) \rangle$;

  - Árvore de sequências:



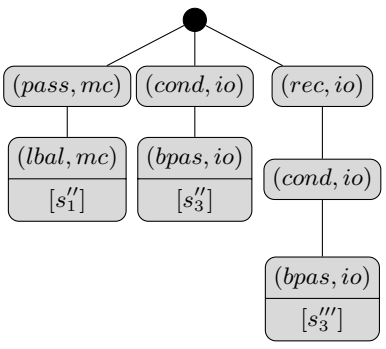
  - Sequências retornadas:

    $s_1'' = \langle (pass, mc), (lbal, mc) \rangle$;
    $s_3'' = \langle (rec, io), (cond, io) \rangle$;

- **Instante 8:**
  - Sequências recebidas:

    $s_1'' = \langle (pass, mc), (lbal, mc) \rangle$;
    $s_3'' = \langle (cond, io), (bpas, io) \rangle$;
    $s_3''' = \langle (rec, io), (cond, io), (bpas, io) \rangle$;

  - Árvore de sequências:



  - Sequências retornadas:

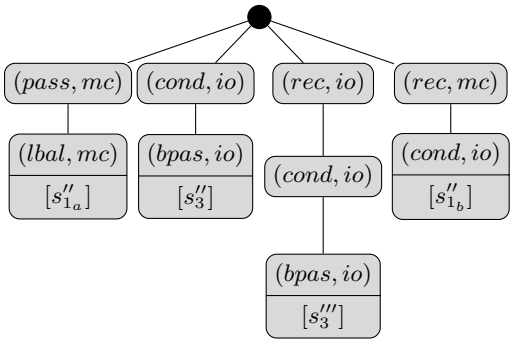
$s''_1 = \langle(pass, mc), (lbal, mc)\rangle$;
$s''_3 = \langle(cond, io), (bpas, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

- **Instantes 9, 10, 11 e 12:** Mesmo resultado do instante 8.
- **Instante 13:**

  – Sequências recebidas:

  $s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
  $s''_{1_b} = \langle(rec, mc), (cond, io)\rangle$;
  $s''_3 = \langle(cond, io), (bpas, io)\rangle$;
  $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

  – Árvore de sequências:



  – Sequências retornadas:

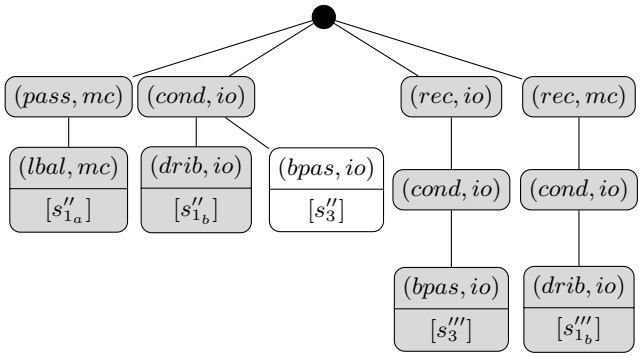
  $s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
  $s''_{1_b} = \langle(rec, mc), (cond, io)\rangle$;
  $s''_3 = \langle(cond, io), (bpas, io)\rangle$;
  $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

- **Instante 14:**

  – Sequências recebidas:

  $s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
  $s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
  $s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
  $s''_3 = \langle(cond, io), (bpas, io)\rangle$;
  $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

  – Árvore de sequências:



– Sequências retornadas:

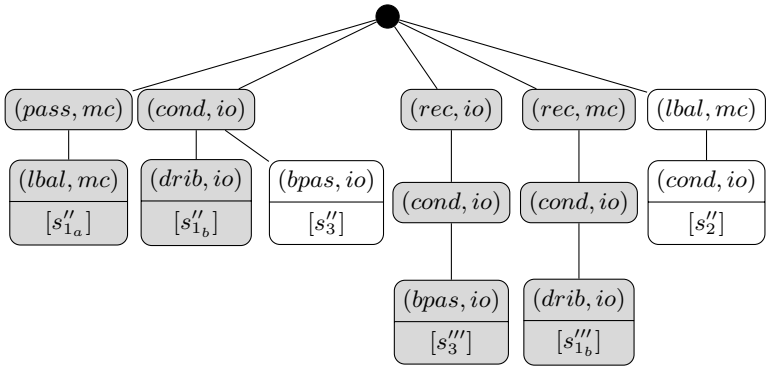
$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

- **Instante 15:** Mesmo resultado do instante 14.
- **Instante 16:**

  – Sequências recebidas:

  $s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
  $s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
  $s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
  $s''_2 = \langle(lbal, mc), (cond, io)\rangle$;
  $s''_3 = \langle(cond, io), (bpas, io)\rangle$;
  $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

  – Árvore de sequências:



  – Sequências retornadas:

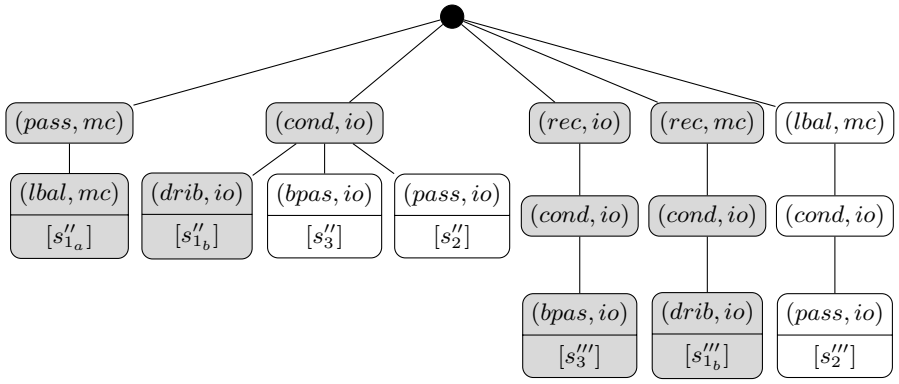
  $s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
  $s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
  $s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
  $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

- **Instante 17:**

  – Sequências recebidas:

$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s''_2 = \langle(cond, io), (pass, io)\rangle$;
$s'''_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s''_3 = \langle(cond, io), (bpas, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

– Árvore de sequências:



– Sequências retornadas:

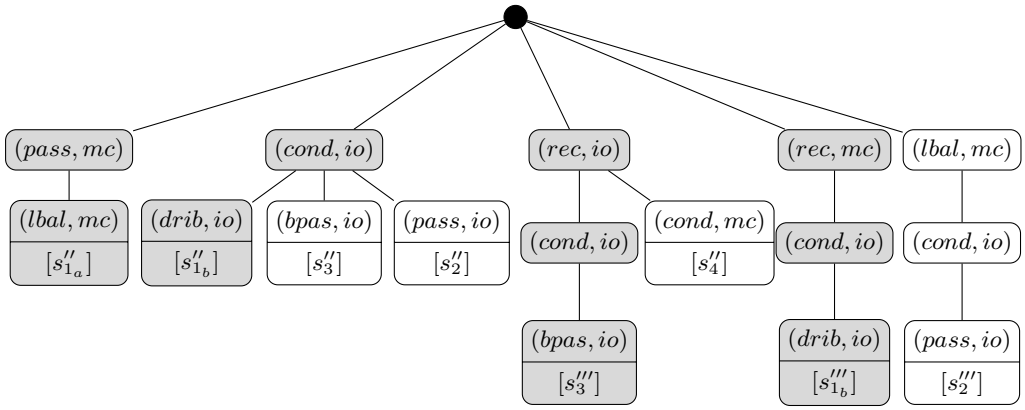
$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

- **Instante 18:** Mesmo resultado do instante 17.
- **Instante 19:**

– Sequências recebidas:

$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s''_2 = \langle(cond, io), (pass, io)\rangle$;
$s'''_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s''_3 = \langle(cond, io), (bpas, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s''_4 = \langle(rec, io), (cond, mc)\rangle$.

– Árvore de sequências:



– Sequências retornadas:

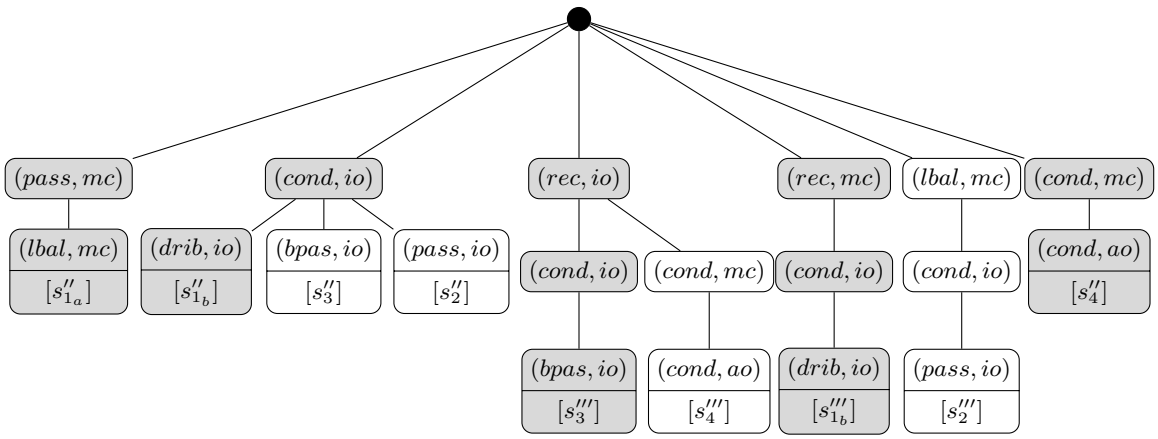
$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

- **Instante 20:**

– Sequências recebidas:

$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;
$s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s''_2 = \langle(cond, io), (pass, io)\rangle$;
$s'''_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
$s''_3 = \langle(cond, io), (bpas, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s''_4 = \langle(cond, mc), (cond, ao)\rangle$;
$s'''_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

– Árvore de sequências:



– Sequências retornadas:

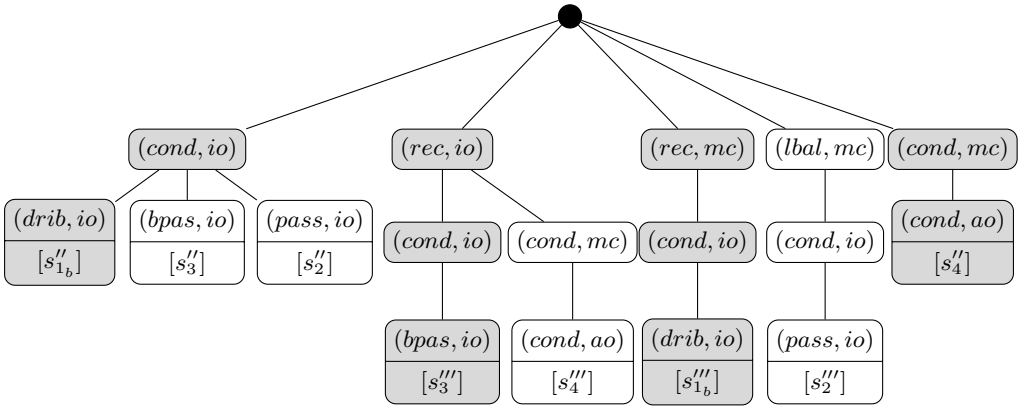
$s''_{1_a} = \langle(pass, mc), (lbal, mc)\rangle$;

$s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s''_4 = \langle(cond, mc), (cond, ao)\rangle$;

- **Instante 21:**
  - Sequências recebidas:

    $s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
    $s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
    $s''_2 = \langle(cond, io), (pass, io)\rangle$;
    $s'''_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
    $s''_3 = \langle(cond, io), (bpas, io)\rangle$;
    $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
    $s''_4 = \langle(cond, mc), (cond, ao)\rangle$;
    $s'''_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

  - Árvore de sequências:



  - Sequências retornadas:

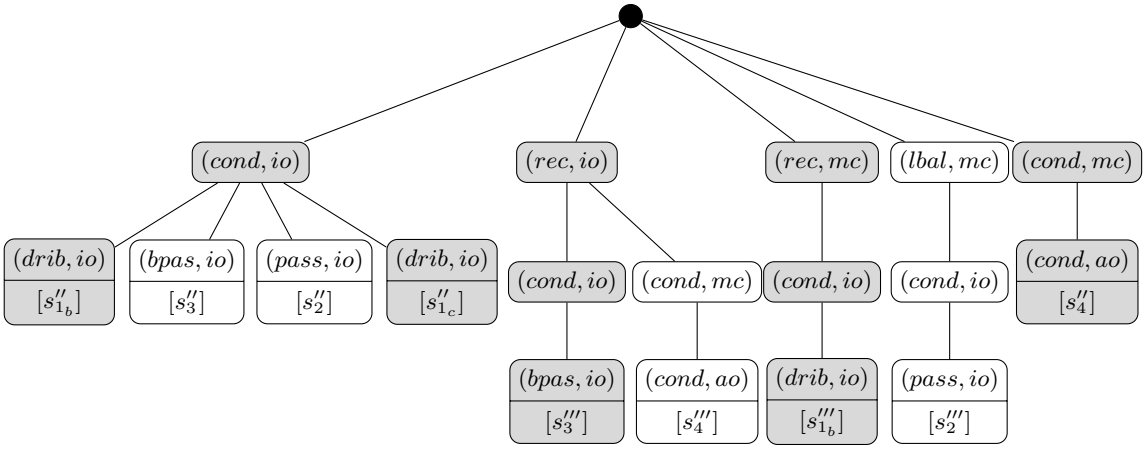
    $s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
    $s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
    $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
    $s''_4 = \langle(cond, mc), (cond, ao)\rangle$;

- **Instante 22:**
  - Sequências recebidas:

    $s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
    $s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
    $s''_{1_c} = \langle(cond, io), (drib, io)\rangle$;
    $s''_2 = \langle(cond, io), (pass, io)\rangle$;
    $s'''_2 = \langle(lbal, mc), (cond, io), (pass, io)\rangle$;
    $s''_3 = \langle(cond, io), (bpas, io)\rangle$;
    $s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;

$s''_4 = \langle(cond, mc), (cond, ao)\rangle$;
$s'''_4 = \langle(rec, io), (cond, mc), (cond, ao)\rangle$.

– Árvore de sequências:



– Sequências retornadas:

$s''_{1_b} = \langle(cond, io), (drib, io)\rangle$;
$s'''_{1_b} = \langle(rec, mc), (cond, io), (drib, io)\rangle$;
$s''_{1_c} = \langle(cond, io), (drib, io)\rangle$;
$s'''_3 = \langle(rec, io), (cond, io), (bpas, io)\rangle$;
$s''_4 = \langle(cond, mc), (cond, ao)\rangle$;

- **Instantes 23, 24 e 25:** Mesmo resultado do instante 22.

APÊNDICE C

Intervalos de confiança dos experimentos da primeira etapa

Este apêndice apresenta os intervalos de confiança dos experimentos da primeira etapa do trabalho descrito nesta tese apresentados no Capítulo 8. Os experimentos foram realizados com os algoritmos *BNL***, *BNL-KB* e *PartitionBest* para processamento de consultas CPrefSQL em bancos de dados tradicionais. O tempo de execução dos algoritmos está expresso em segundos. Os intervalos foram calculados considerando uma confiança de 95% e uma população de três experimentos.

Experimento com variação no nível de preferência máximo (LEV)

LEV	*BNL***	*BNL-KB*	*PartitionBest*
1	$1.069,653 \pm 2,922$	$680,663 \pm 2,184$	$1,041 \pm 0,011$
2	$1.104,639 \pm 2,752$	$842,360 \pm 10,065$	$1,022 \pm 0,004$
3	$1.110,389 \pm 0,624$	$1.268,848 \pm 5,516$	$1,170 \pm 0,009$
4	$1.074,991 \pm 4,064$	$1.353,111 \pm 5,876$	$1,193 \pm 0,004$
5	$1.060,352 \pm 3,917$	$1.653,870 \pm 7,760$	$1,256 \pm 0,007$
6	$1.026,744 \pm 7,004$	$2.181,297 \pm 6,161$	$1,365 \pm 0,004$

Experimento com variação no número de top-k tuplas (TOP)

TOP	*BNL***	*BNL-KB*	*PartitionBest*
-1	$1.104,808 \pm 1,341$	$838,396 \pm 5,100$	$1,021 \pm 0,005$
10	$4.251,406 \pm 21,868$	$3.363,171 \pm 47,216$	$0,601 \pm 0,004$
100	$4.228,253 \pm 17,979$	$3.338,754 \pm 3,513$	$0,612 \pm 0,004$
1000	$4.226,529 \pm 4,554$	$3.348,353 \pm 32,183$	$0,675 \pm 0,007$
5000	$4.217,235 \pm 1,633$	$3.332,862 \pm 18,806$	$0,929 \pm 0,005$

Experimento com variação no número de regras (RUL)

RUL	*BNL***	*BNL-KB*	*PartitionBest*
6	$1.104,339 \pm 2,792$	$839,671 \pm 8,491$	$1,018 \pm 0,007$
10	$1.679,384 \pm 5,187$	$1.283,432 \pm 4,063$	$1,167 \pm 0,011$
20	$2.584,421 \pm 8,912$	$2.887,289 \pm 2,508$	$8,283 \pm 0,022$
30	$3.132,036 \pm 12,614$	$2.821,999 \pm 12,473$	$15,453 \pm 0,056$
40	$3.862,413 \pm 6,525$	$3.232,172 \pm 9,452$	$19,942 \pm 0,089$

Experimento com variação no tamanho do banco de dados (DB)

DB	*BNL***	*BNL-KB*	*PartitionBest*
16	$274,357 \pm 7,686$	$206,392 \pm 5,658$	$0,506 \pm 0,011$
32	$1.110,447 \pm 16,438$	$788,055 \pm 13,050$	$0,888 \pm 0,041$
64	$4.214,288 \pm 5,688$	$3.118,726 \pm 11,614$	$1,784 \pm 0,058$
128	$17.043,363 \pm 10,536$	$12.664,997 \pm 28,657$	$3,489 \pm 0,091$
256	$71.516,269 \pm 9,516$	$51.100,896 \pm 18,697$	$6,905 \pm 0,173$
512	$283.362,450 \pm 11,886$	$203.720,272 \pm 25,580$	$14,529 \pm 0,364$
1024	$1.167.779,990 \pm 29,732$	$795.288,955 \pm 23,745$	$35,357 \pm 1,132$

Experimento com variação nas consultas (QUE)

QUE	*BNL***	*BNL-KB*	*PartitionBest*
Q3	$8,656 \pm 0,044$	$7,591 \pm 0,127$	$0,238 \pm 0,003$
Q5	$1.105,133 \pm 2,363$	$844,584 \pm 5,451$	$1,025 \pm 0,010$
Q10	$5.666,081 \pm 8,573$	$4.454,036 \pm 9,814$	$1,587 \pm 0,006$
Q18	$19.314,154 \pm 13,180$	$16.179,238 \pm 128,325$	$2,626 \pm 0,015$

APÊNDICE D

Intervalos de confiança dos experimentos da segunda etapa

Este apêndice apresenta os intervalos de confiança do tempo de execução e do consumo de memória dos experimentos da segunda etapa do trabalho descrito nesta tese apresentados no Capítulo 8. O tempo de execução foi medido em segundos e o consumo de memória foi contabilizado em megabytes (MB). Tais experimentos foram realizados com os algoritmos *PartitionBest*, *IncAncestorsBest*, *IncGraphBest* e `IncPartitionBest` para processamento de consultas contínuas contendo preferências condicionais. Os intervalos foram calculados considerando uma confiança de 95% e uma população de cinco experimentos.

D.1 Experimentos com dados sintéticos

D.1.1 Intervalo de confiança do tempo de execução

Experimento com variação no número de atributos (ATT)

ATT	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
8	$321,335 \pm 1,708$	$317,812 \pm 1,581$	$7,452 \pm 0,016$	$52,998 \pm 0,130$
16	$349,197 \pm 6,860$	$347,030 \pm 3,096$	$14,301 \pm 0,097$	$114,153 \pm 0,811$
32	$388,968 \pm 2,338$	$389,478 \pm 1,292$	$28,001 \pm 0,171$	$227,358 \pm 0,903$
64	$481,754 \pm 2,470$	$477,312 \pm 4,720$	$54,306 \pm 0,837$	$467,863 \pm 1,719$

Experimento com variação no número de tuplas (TUP)

TUP	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
500	150,452 ± 2,046	151,794 ± 1,594	4,771 ± 0,047	27,174 ± 0,165
1000	321,335 ± 1,708	317,812 ± 1,581	7,452 ± 0,016	52,998 ± 0,130
2000	719,983 ± 15,751	716,748 ± 4,154	13,276 ± 0,123	107,904 ± 1,058
4000	1.673,498 ± 15,946	1.679,100 ± 30,005	24,710 ± 0,161	215,583 ± 1,203
8000	4.241,614 ± 37,379	4.187,711 ± 25,348	47,978 ± 0,369	437,657 ± 1,194

Experimento com variação no número de inserções por instante (INS)

INS	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
50	321,335 ± 1,708	317,812 ± 1,581	7,452 ± 0,016	52,998 ± 0,130
100	2.071,463 ± 14,336	2.075,513 ± 23,416	22,443 ± 0,119	188,715 ± 1,556
200	9.859,642 ± 70,799	9.796,075 ± 53,618	53,927 ± 0,596	472,659 ± 2,440
400	42.950,443 ± 485,395	42.669,288 ± 666,737	127,657 ± 0,578	1.076,360 ± 7,880

Experimento com variação no número de remoções por instante (DEL)

DEL	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
50	4.241,614 ± 37,379	4.187,711 ± 25,348	47,978 ± 0,369	437,657 ± 1,194
100	3.508,797 ± 27,892	3.540,474 ± 56,886	35,385 ± 0,241	300,625 ± 3,825
200	2.553,538 ± 10,220	2.518,982 ± 32,524	17,948 ± 0,082	119,842 ± 1,273
400	2.172,040 ± 43,360	2.172,401 ± 26,962	11,553 ± 0,058	54,763 ± 0,271

Experimento com variação no número de regras (RUL)

RUL	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
2	114,116 ± 1,264	114,142 ± 1,137	6,236 ± 0,021	17,536 ± 0,158
4	185,859 ± 2,488	185,631 ± 1,857	6,652 ± 0,022	29,500 ± 0,410
8	321,335 ± 1,708	317,812 ± 1,581	7,452 ± 0,016	52,998 ± 0,130
16	594,006 ± 4,781	584,879 ± 11,828	9,325 ± 0,119	102,060 ± 0,948
32	1.128,701 ± 19,009	1.118,748 ± 9,241	12,868 ± 0,168	199,934 ± 2,242

Experimento com variação no nível de preferência máximo (LEV)

LEV	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
1	325,650 ± 3,704	321,333 ± 2,195	6,984 ± 0,077	37,878 ± 0,213
2	321,335 ± 1,708	317,812 ± 1,581	7,452 ± 0,016	52,998 ± 0,130
4	323,089 ± 4,138	326,784 ± 5,643	8,753 ± 0,078	85,779 ± 1,417
8	324,584 ± 2,711	323,589 ± 2,896	11,095 ± 0,100	150,603 ± 2,338

Experimento com variação no número de atributos indiferentes (IND)

IND	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
0	$331,191 \pm 4,353$	$327,721 \pm 8,140$	$7,983 \pm 0,026$	$73,909 \pm 0,230$
1	$328,019 \pm 2,984$	$326,640 \pm 3,264$	$7,864 \pm 0,042$	$68,403 \pm 0,270$
2	$323,765 \pm 3,589$	$324,047 \pm 5,087$	$7,701 \pm 0,062$	$63,377 \pm 0,135$
4	$321,335 \pm 1,708$	$317,812 \pm 1,581$	$7,452 \pm 0,016$	$52,998 \pm 0,130$

D.1.2 Intervalo de confiança do consumo de memória

Experimento com variação no número de atributos (ATT)

ATT	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
8	$19,499 \pm 0,407$	$18,990 \pm 0,000$	$18,990 \pm 0,000$	$18,950 \pm 0,000$
16	$19,970 \pm 0,000$	$19,960 \pm 0,000$	$19,970 \pm 0,000$	$18,970 \pm 0,000$
32	$34,950 \pm 0,000$	$34,933 \pm 0,004$	$34,950 \pm 0,000$	$33,885 \pm 0,005$
64	$37,461 \pm 0,261$	$37,059 \pm 0,252$	$36,937 \pm 0,005$	$36,861 \pm 0,000$

Experimento com variação no número de tuplas (TUP)

TUP	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
500	$16,000 \pm 0,000$	$16,000 \pm 0,000$	$16,000 \pm 0,000$	$16,000 \pm 0,000$
1000	$19,499 \pm 0,407$	$18,990 \pm 0,000$	$18,990 \pm 0,000$	$18,950 \pm 0,000$
2000	$26,980 \pm 0,000$	$25,980 \pm 0,000$	$25,980 \pm 0,000$	$24,921 \pm 0,000$
4000	$41,580 \pm 0,005$	$40,653 \pm 0,000$	$40,594 \pm 0,000$	$37,842 \pm 0,000$
8000	$71,905 \pm 0,005$	$69,743 \pm 0,000$	$69,901 \pm 0,000$	$63,673 \pm 0,000$

Experimento com variação no número de inserções por instante (INS)

INS	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
50	$19,499 \pm 0,407$	$18,990 \pm 0,000$	$18,990 \pm 0,000$	$18,950 \pm 0,000$
100	$37,483 \pm 0,016$	$36,618 \pm 0,005$	$36,689 \pm 0,005$	$34,291 \pm 0,008$
200	$74,176 \pm 0,007$	$72,232 \pm 0,005$	$72,366 \pm 0,012$	$66,105 \pm 0,005$
400	$150,143 \pm 0,013$	$145,364 \pm 0,004$	$145,743 \pm 0,000$	$129,285 \pm 0,077$

Experimento com variação no número de remoções por instante (DEL)

DEL	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
50	$71,905 \pm 0,005$	$69,743 \pm 0,000$	$69,901 \pm 0,000$	$63,673 \pm 0,000$
100	$65,693 \pm 0,024$	$63,626 \pm 0,007$	$63,626 \pm 0,038$	$53,624 \pm 0,000$
200	$59,842 \pm 0,000$	$58,396 \pm 0,000$	$58,594 \pm 0,143$	$49,493 \pm 0,038$
400	$56,525 \pm 0,000$	$55,030 \pm 0,000$	$55,386 \pm 0,000$	$47,158 \pm 0,000$

Experimento com variação no número de regras (RUL)

RUL	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
2	$19,145 \pm 0,185$	$18,990 \pm 0,000$	$18,990 \pm 0,000$	$18,459 \pm 0,158$
4	$19,578 \pm 0,242$	$18,990 \pm 0,000$	$18,990 \pm 0,000$	$18,711 \pm 0,073$
8	$19,499 \pm 0,407$	$18,990 \pm 0,000$	$18,990 \pm 0,000$	$18,950 \pm 0,000$
16	$19,501 \pm 0,409$	$18,990 \pm 0,000$	$18,990 \pm 0,000$	$18,503 \pm 0,045$
32	$19,877 \pm 0,005$	$18,990 \pm 0,000$	$19,974 \pm 0,005$	$18,980 \pm 0,000$

Experimento com variação no nível de preferência máximo (LEV)

LEV	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
1	$18,990 \pm 0,000$	$18,990 \pm 0,000$	$18,990 \pm 0,000$	$18,903 \pm 0,023$
2	$19,499 \pm 0,407$	$18,990 \pm 0,000$	$18,990 \pm 0,000$	$18,950 \pm 0,000$
4	$19,327 \pm 0,404$	$18,990 \pm 0,000$	$18,990 \pm 0,000$	$18,679 \pm 0,087$
8	$19,655 \pm 0,327$	$18,990 \pm 0,000$	$18,990 \pm 0,000$	$18,703 \pm 0,035$

Experimento com variação no número de atributos indiferentes (IND)

IND	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
0	$19,335 \pm 0,414$	$18,990 \pm 0,000$	$18,990 \pm 0,000$	$18,947 \pm 0,008$
1	$19,602 \pm 0,356$	$19,168 \pm 0,349$	$18,990 \pm 0,000$	$18,931 \pm 0,000$
2	$19,525 \pm 0,428$	$18,990 \pm 0,000$	$18,990 \pm 0,000$	$18,929 \pm 0,043$
4	$19,499 \pm 0,407$	$18,990 \pm 0,000$	$18,990 \pm 0,000$	$18,950 \pm 0,000$

D.2 Experimentos com dados reais

D.2.1 Intervalo de confiança do tempo de execução

Experimento com variação na abrangência temporal (RAN)

RAN	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
2	$4.205,314 \pm 115,487$	$4.246,316 \pm 95,506$	$43,554 \pm 1,119$	$88,379 \pm 2,373$
3	$6.997,778 \pm 231,004$	$6.914,252 \pm 151,714$	$45,837 \pm 0,883$	$129,739 \pm 2,405$
4	$9.698,066 \pm 227,910$	$9.676,780 \pm 125,448$	$47,985 \pm 1,092$	$170,080 \pm 3,855$
5	$12.392,240 \pm 326,190$	$12.371,158 \pm 379,854$	$50,609 \pm 1,164$	$210,788 \pm 5,811$
6	$15.037,118 \pm 353,225$	$15.047,429 \pm 270,442$	$53,682 \pm 0,996$	$249,797 \pm 7,268$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
1	$9.698,066 \pm 227,910$	$9.676,780 \pm 125,448$	$47,985 \pm 1,092$	$170,080 \pm 3,855$
2	$8.348,115 \pm 248,209$	$8.490,721 \pm 161,303$	$48,205 \pm 0,812$	$151,317 \pm 3,809$
3	$7.052,859 \pm 186,853$	$7.089,983 \pm 163,614$	$46,901 \pm 1,138$	$132,168 \pm 2,742$
4	$5.626,363 \pm 109,825$	$5.670,524 \pm 178,374$	$45,730 \pm 1,248$	$110,641 \pm 2,730$

D.2.2 Intervalo de confiança do consumo de memória

Experimento com variação na abrangência temporal (RAN)

RAN	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
2	$20,418 \pm 0,096$	$20,197 \pm 0,102$	$20,921 \pm 0,081$	$20,753 \pm 0,091$
3	$21,785 \pm 0,066$	$21,617 \pm 0,105$	$21,821 \pm 0,096$	$21,148 \pm 0,062$
4	$23,239 \pm 0,505$	$23,367 \pm 0,390$	$24,082 \pm 0,061$	$21,562 \pm 0,078$
5	$25,809 \pm 0,106$	$25,397 \pm 0,076$	$25,739 \pm 0,087$	$22,024 \pm 0,219$
6	$27,292 \pm 0,064$	$26,947 \pm 0,094$	$27,310 \pm 0,093$	$23,193 \pm 0,162$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	*IncAncestorsBest*	*IncGraphBest*	*IncPartitionBest*	*PartitionBest*
1	$23,239 \pm 0,505$	$23,367 \pm 0,390$	$24,082 \pm 0,061$	$21,562 \pm 0,078$
2	$22,352 \pm 0,078$	$22,356 \pm 0,364$	$22,518 \pm 0,117$	$21,312 \pm 0,037$
3	$21,870 \pm 0,126$	$21,553 \pm 0,122$	$22,319 \pm 0,517$	$21,335 \pm 0,087$
4	$22,018 \pm 0,151$	$21,705 \pm 0,533$	$22,000 \pm 0,116$	$21,252 \pm 0,078$

APÊNDICE E

Intervalos de confiança dos experimentos da terceira etapa

Este apêndice apresenta os intervalos de confiança do tempo de execução e do consumo de memória dos experimentos da terceira etapa do trabalho descrito nesta tese apresentados no Capítulo 8. O tempo de execução foi medido em segundos e o consumo de memória foi contabilizado em megabytes (MB). Os intervalos foram calculados considerando uma confiança de 95% e uma população de cinco experimentos.

E.1 Experimentos com o operador SEQ

Esta seção apresenta os intervalos de confiança dos experimentos com o operador **SEQ**. Os experimentos compararam o tempo de execução e o consumo de memória do operador **SEQ** (processado pelo algoritmo *IncSeq*) e sua equivalência em CQL.

E.1.1 Experimentos com dados sintéticos

E.1.1.1 Intervalo de confiança do tempo de execução

Experimento com variação no número de atributos (ATT)

ATT	CQL	**SEQ** (*IncSeq*)
8	$46,005 \pm 11,408$	$0,239 \pm 0,028$
10	$60,022 \pm 2,044$	$0,239 \pm 0,010$
12	$65,764 \pm 2,173$	$0,257 \pm 0,012$
14	$72,133 \pm 1,193$	$0,271 \pm 0,008$
16	$78,339 \pm 1,178$	$0,287 \pm 0,009$

Experimento com variação no número de sequências (NSQ)

NSQ	CQL	**SEQ** (*IncSeq*)
4	$20,977 \pm 1,095$	$0,139 \pm 0,016$
8	$32,266 \pm 2,676$	$0,156 \pm 0,002$
16	$46,005 \pm 11,408$	$0,239 \pm 0,028$
24	$76,669 \pm 4,452$	$0,287 \pm 0,015$
32	$100,133 \pm 4,907$	$0,351 \pm 0,006$

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	**SEQ** (*IncSeq*)
10	$6,273 \pm 1,752$	$0,192 \pm 0,032$
20	$22,828 \pm 5,959$	$0,211 \pm 0,025$
30	$46,005 \pm 11,408$	$0,239 \pm 0,028$
40	$102,821 \pm 13,341$	$0,323 \pm 0,042$
50	$157,886 \pm 19,028$	$0,317 \pm 0,015$
60	$280,670 \pm 18,799$	$0,390 \pm 0,017$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	**SEQ** (*IncSeq*)
1	$64,467 \pm 0,969$	$0,229 \pm 0,004$
10	$46,005 \pm 11,408$	$0,239 \pm 0,028$
20	$48,532 \pm 1,067$	$0,220 \pm 0,005$
30	$40,103 \pm 0,468$	$0,204 \pm 0,006$
40	$37,721 \pm 0,308$	$0,206 \pm 0,009$

E.1.1.2 Intervalo de confiança do consumo de memória

Experimento com variação no número de atributos (ATT)

ATT	CQL	**SEQ** (*IncSeq*)
8	$47,765 \pm 0,074$	$21,725 \pm 0,067$
10	$48,321 \pm 0,066$	$21,734 \pm 0,064$
12	$48,689 \pm 0,032$	$21,779 \pm 0,038$
14	$48,885 \pm 0,070$	$21,752 \pm 0,063$
16	$49,569 \pm 0,095$	$21,790 \pm 0,075$

Experimento com variação no número de sequências (NSQ)

NSQ	CQL	**SEQ** (*IncSeq*)
4	$33,975 \pm 0,023$	$21,697 \pm 0,033$
8	$37,405 \pm 0,034$	$21,657 \pm 0,057$
16	$47,765 \pm 0,074$	$21,725 \pm 0,067$
24	$59,311 \pm 0,098$	$21,816 \pm 0,030$
32	$70,933 \pm 0,023$	$21,873 \pm 0,041$

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	**SEQ** (*IncSeq*)
10	$29,008 \pm 0,024$	$21,716 \pm 0,054$
20	$35,962 \pm 0,020$	$21,759 \pm 0,048$
30	$47,765 \pm 0,074$	$21,725 \pm 0,067$
40	$62,568 \pm 0,029$	$21,734 \pm 0,061$
50	$79,471 \pm 0,084$	$21,754 \pm 0,044$
60	$98,138 \pm 0,026$	$21,741 \pm 0,063$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	**SEQ** (*IncSeq*)
1	$47,413 \pm 0,081$	$21,659 \pm 0,064$
10	$47,765 \pm 0,074$	$21,725 \pm 0,067$
20	$47,816 \pm 0,067$	$21,709 \pm 0,089$
30	$44,843 \pm 0,064$	$21,791 \pm 0,033$
40	$40,967 \pm 0,042$	$21,743 \pm 0,046$

E.1.2 Experimentos com dados reais

E.1.2.1 Intervalo de confiança do tempo de execução

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	**SEQ** (*IncSeq*)
10	$20,295 \pm 0,125$	$0,473 \pm 0,034$
20	$56,460 \pm 0,491$	$0,550 \pm 0,025$
30	$110,196 \pm 0,838$	$0,619 \pm 0,031$
40	$179,196 \pm 1,449$	$0,662 \pm 0,020$
50	$263,284 \pm 1,453$	$0,713 \pm 0,015$
60	$368,378 \pm 2,689$	$0,771 \pm 0,022$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	**SEQ** (*IncSeq*)
1	$115,176 \pm 0,732$	$0,622 \pm 0,013$
10	$110,196 \pm 0,838$	$0,619 \pm 0,031$
20	$104,219 \pm 0,927$	$0,577 \pm 0,035$
30	$98,252 \pm 0,837$	$0,542 \pm 0,037$
40	$92,334 \pm 0,319$	$0,493 \pm 0,035$

E.1.2.2 Intervalo de confiança do consumo de memória

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	**SEQ** (*IncSeq*)
10	$27,304 \pm 0,032$	$21,904 \pm 0,083$
20	$29,519 \pm 0,057$	$21,973 \pm 0,035$
30	$31,760 \pm 0,080$	$21,946 \pm 0,074$
40	$34,228 \pm 0,028$	$21,962 \pm 0,074$
50	$37,053 \pm 0,065$	$21,980 \pm 0,029$
60	$39,081 \pm 0,098$	$21,996 \pm 0,026$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	**SEQ** (*IncSeq*)
1	$31,760 \pm 0,087$	$21,938 \pm 0,090$
10	$31,760 \pm 0,080$	$21,946 \pm 0,074$
20	$31,758 \pm 0,039$	$21,908 \pm 0,066$
30	$31,687 \pm 0,066$	$21,916 \pm 0,068$
40	$31,743 \pm 0,016$	$21,942 \pm 0,040$

E.2 Experimentos com o operador CONSEQ

Esta seção apresenta os intervalos de confiança dos experimentos com o operador **CONSEQ**. Os experimentos compararam o tempo de execução e o consumo de memória do operador **CONSEQ** (processado pelos algoritmos *NaiveSubseq* e *IncConseq*) e sua equivalência em CQL.

E.2.1 Experimentos com dados sintéticos

E.2.1.1 Intervalo de confiança do tempo de execução

Experimento com variação no número de atributos (ATT)

ATT	CQL	*NaiveSubseq*	*IncConseq*
8	$12,227 \pm 3,575$	$0,366 \pm 0,021$	$0,314 \pm 0,017$
10	$16,883 \pm 1,711$	$0,375 \pm 0,012$	$0,350 \pm 0,023$
12	$18,786 \pm 1,660$	$0,392 \pm 0,010$	$0,361 \pm 0,016$
14	$20,890 \pm 1,755$	$0,415 \pm 0,020$	$0,370 \pm 0,021$
16	$22,868 \pm 1,468$	$0,434 \pm 0,028$	$0,389 \pm 0,022$

Experimento com variação no número de sequências (NSQ)

NSQ	CQL	*NaiveSubseq*	*IncConseq*
4	$3,942 \pm 0,705$	$0,168 \pm 0,028$	$0,156 \pm 0,012$
8	$7,370 \pm 1,287$	$0,224 \pm 0,019$	$0,207 \pm 0,013$
16	$12,227 \pm 3,575$	$0,366 \pm 0,021$	$0,314 \pm 0,017$
24	$21,677 \pm 3,268$	$0,500 \pm 0,016$	$0,433 \pm 0,022$
32	$29,943 \pm 3,460$	$0,639 \pm 0,026$	$0,549 \pm 0,027$

Experimento com variação no porcentagem de tuplas consecutivas (PCT)

PCT	CQL	*NaiveSubseq*	*IncConseq*
0,00	$19,359 \pm 0,942$	$0,337 \pm 0,021$	$0,252 \pm 0,014$
0,25	$12,156 \pm 0,591$	$0,285 \pm 0,019$	$0,250 \pm 0,017$
0,50	$12,227 \pm 3,575$	$0,366 \pm 0,021$	$0,314 \pm 0,017$
0,75	$16,222 \pm 0,692$	$0,374 \pm 0,021$	$0,351 \pm 0,016$
1,00	$15,805 \pm 0,601$	$0,372 \pm 0,018$	$0,373 \pm 0,021$

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	*NaiveSubseq*	*IncConseq*
10	$2,451 \pm 0,742$	$0,157 \pm 0,050$	$0,158 \pm 0,046$
20	$5,981 \pm 1,826$	$0,224 \pm 0,067$	$0,214 \pm 0,056$
30	$12,227 \pm 3,575$	$0,366 \pm 0,021$	$0,314 \pm 0,017$
40	$18,518 \pm 4,282$	$0,406 \pm 0,095$	$0,340 \pm 0,079$
50	$30,135 \pm 6,702$	$0,592 \pm 0,022$	$0,489 \pm 0,022$
60	$37,371 \pm 9,315$	$0,591 \pm 0,149$	$0,513 \pm 0,085$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	*NaiveSubseq*	*IncConseq*
1	$25,359 \pm 1,741$	$0,393 \pm 0,031$	$0,334 \pm 0,016$
10	$12,227 \pm 3,575$	$0,366 \pm 0,021$	$0,314 \pm 0,017$
20	$13,651 \pm 0,659$	$0,344 \pm 0,035$	$0,305 \pm 0,024$
30	$10,671 \pm 0,554$	$0,299 \pm 0,018$	$0,274 \pm 0,016$
40	$10,108 \pm 0,493$	$0,287 \pm 0,023$	$0,268 \pm 0,017$

E.2.1.2 Intervalo de confiança do consumo de memória

Experimento com variação no número de atributos (ATT)

ATT	CQL	*NaiveSubseq*	*IncConseq*
8	$33,427 \pm 0,086$	$21,793 \pm 0,019$	$21,753 \pm 0,074$
10	$33,972 \pm 0,089$	$21,755 \pm 0,062$	$21,733 \pm 0,079$
12	$34,829 \pm 0,045$	$21,807 \pm 0,015$	$21,738 \pm 0,087$
14	$37,904 \pm 0,048$	$21,776 \pm 0,062$	$21,829 \pm 0,024$
16	$38,660 \pm 0,046$	$21,754 \pm 0,087$	$21,788 \pm 0,033$

Experimento com variação no número de sequências (NSQ)

NSQ	CQL	*NaiveSubseq*	*IncConseq*
4	$25,830 \pm 0,033$	$21,737 \pm 0,019$	$21,674 \pm 0,063$
8	$27,645 \pm 0,049$	$21,768 \pm 0,073$	$21,752 \pm 0,081$
16	$33,427 \pm 0,086$	$21,793 \pm 0,019$	$21,753 \pm 0,074$
24	$39,612 \pm 0,070$	$22,206 \pm 0,016$	$22,204 \pm 0,030$
32	$46,280 \pm 0,092$	$22,557 \pm 0,027$	$22,557 \pm 0,038$

Experimento com variação no porcentagem de tuplas consecutivas (PCT)

PCT	CQL	*NaiveSubseq*	*IncConseq*
0,00	$33,278 \pm 0,073$	$21,818 \pm 0,034$	$21,814 \pm 0,028$
0,25	$29,228 \pm 0,022$	$21,862 \pm 0,040$	$21,780 \pm 0,082$
0,50	$33,427 \pm 0,086$	$21,793 \pm 0,019$	$21,753 \pm 0,074$
0,75	$33,014 \pm 0,065$	$21,932 \pm 0,064$	$21,949 \pm 0,033$
1,00	$31,229 \pm 0,090$	$22,043 \pm 0,028$	$21,966 \pm 0,078$

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	*NaiveSubseq*	*IncConseq*
10	$25,683 \pm 0,016$	$21,709 \pm 0,019$	$21,745 \pm 0,054$
20	$28,432 \pm 0,040$	$21,704 \pm 0,103$	$21,738 \pm 0,055$
30	$33,427 \pm 0,086$	$21,793 \pm 0,019$	$21,753 \pm 0,074$
40	$39,116 \pm 0,071$	$21,840 \pm 0,099$	$21,849 \pm 0,106$
50	$45,338 \pm 0,036$	$22,011 \pm 0,061$	$22,024 \pm 0,066$
60	$51,511 \pm 0,133$	$22,084 \pm 0,048$	$22,122 \pm 0,070$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	*NaiveSubseq*	*IncConseq*
1	$36,394 \pm 0,030$	$21,739 \pm 0,069$	$21,759 \pm 0,043$
10	$33,427 \pm 0,086$	$21,793 \pm 0,019$	$21,753 \pm 0,074$
20	$30,411 \pm 0,027$	$21,783 \pm 0,019$	$21,765 \pm 0,017$
30	$29,589 \pm 0,024$	$21,740 \pm 0,055$	$21,762 \pm 0,076$
40	$29,016 \pm 0,055$	$21,728 \pm 0,060$	$21,739 \pm 0,063$

E.2.2 Experimentos com dados reais

E.2.2.1 Intervalo de confiança do tempo de execução

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	*NaiveSubseq*	*IncConseq*
10	$9,090 \pm 0,069$	$0,576 \pm 0,016$	$0,588 \pm 0,013$
20	$16,944 \pm 0,064$	$0,799 \pm 0,017$	$0,826 \pm 0,023$
30	$24,835 \pm 0,201$	$0,986 \pm 0,019$	$1,020 \pm 0,014$
40	$32,743 \pm 0,419$	$1,191 \pm 0,038$	$1,245 \pm 0,020$
50	$40,469 \pm 0,361$	$1,373 \pm 0,018$	$1,444 \pm 0,032$
60	$48,702 \pm 0,323$	$1,550 \pm 0,015$	$1,627 \pm 0,015$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	*NaiveSubseq*	*IncConseq*
1	$28,581 \pm 0,626$	$1,081 \pm 0,019$	$1,111 \pm 0,012$
10	$24,835 \pm 0,201$	$0,986 \pm 0,019$	$1,020 \pm 0,014$
20	$20,546 \pm 0,188$	$0,915 \pm 0,035$	$0,936 \pm 0,028$
30	$16,900 \pm 0,082$	$0,800 \pm 0,011$	$0,826 \pm 0,021$
40	$13,628 \pm 0,063$	$0,689 \pm 0,020$	$0,710 \pm 0,015$

E.2.2.2 Intervalo de confiança do consumo de memória

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	*NaiveSubseq*	*IncConseq*
10	$24,997 \pm 0,005$	$21,972 \pm 0,054$	$21,908 \pm 0,081$
20	$24,995 \pm 0,080$	$21,950 \pm 0,071$	$21,957 \pm 0,077$
30	$24,992 \pm 0,085$	$21,961 \pm 0,020$	$21,925 \pm 0,066$
40	$25,003 \pm 0,064$	$21,984 \pm 0,023$	$21,961 \pm 0,039$
50	$25,082 \pm 0,030$	$21,960 \pm 0,063$	$21,991 \pm 0,025$
60	$25,143 \pm 0,008$	$21,973 \pm 0,078$	$21,970 \pm 0,064$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	*NaiveSubseq*	*IncConseq*
1	$25,036 \pm 0,052$	$21,985 \pm 0,040$	$21,920 \pm 0,026$
10	$24,992 \pm 0,085$	$21,961 \pm 0,020$	$21,925 \pm 0,066$
20	$24,978 \pm 0,040$	$21,972 \pm 0,035$	$21,951 \pm 0,033$
30	$24,972 \pm 0,041$	$21,966 \pm 0,030$	$21,976 \pm 0,043$
40	$24,939 \pm 0,062$	$21,947 \pm 0,040$	$21,955 \pm 0,033$

E.3 Experimentos com o operador ENDSEQ

Esta seção apresenta os intervalos de confiança dos experimentos com o operador **ENDSEQ**. Os experimentos compararam o tempo de execução e o consumo de memória do operador **ENDSEQ** (processado pelos algoritmos *NaiveSubseq* e *IncEndseq*) e sua equivalência em CQL.

E.3.1 Experimentos com dados sintéticos

E.3.1.1 Intervalo de confiança do tempo de execução

Experimento com variação no número de atributos (ATT)

ATT	CQL	*NaiveSubseq*	*IncEndseq*
8	$29,046 \pm 8,435$	$1,055 \pm 0,038$	$0,902 \pm 0,013$
10	$47,052 \pm 2,198$	$1,143 \pm 0,028$	$1,049 \pm 0,046$
12	$54,803 \pm 2,141$	$1,179 \pm 0,046$	$1,041 \pm 0,035$
14	$62,886 \pm 2,090$	$1,214 \pm 0,071$	$1,066 \pm 0,038$
16	$71,093 \pm 1,792$	$1,264 \pm 0,104$	$1,109 \pm 0,062$

Experimento com variação no número de sequências (NSQ)

NSQ	CQL	*NaiveSubseq*	*IncEndseq*
4	$9,668 \pm 0,919$	$0,334 \pm 0,038$	$0,293 \pm 0,012$
8	$19,059 \pm 1,397$	$0,565 \pm 0,034$	$0,502 \pm 0,018$
16	$29,046 \pm 8,435$	$1,055 \pm 0,038$	$0,902 \pm 0,013$
24	$57,034 \pm 3,847$	$1,563 \pm 0,070$	$1,397 \pm 0,061$
32	$77,943 \pm 4,106$	$2,122 \pm 0,066$	$1,877 \pm 0,077$

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	*NaiveSubseq*	*IncEndseq*
10	$2,079 \pm 0,554$	$0,202 \pm 0,077$	$0,191 \pm 0,059$
20	$9,914 \pm 2,761$	$0,429 \pm 0,118$	$0,394 \pm 0,086$
30	$29,046 \pm 8,435$	$1,055 \pm 0,038$	$0,902 \pm 0,013$
40	$55,931 \pm 10,897$	$1,639 \pm 0,346$	$1,447 \pm 0,312$
50	$115,833 \pm 14,868$	$3,113 \pm 0,101$	$2,818 \pm 0,064$
60	$158,561 \pm 39,579$	$4,105 \pm 1,060$	$3,859 \pm 0,832$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	*NaiveSubseq*	*IncEndseq*
1	$67,561 \pm 1,637$	$1,253 \pm 0,028$	$1,108 \pm 0,016$
10	$29,046 \pm 8,435$	$1,055 \pm 0,038$	$0,902 \pm 0,013$
20	$32,483 \pm 0,812$	$0,876 \pm 0,030$	$0,753 \pm 0,009$
30	$24,284 \pm 0,719$	$0,682 \pm 0,030$	$0,587 \pm 0,007$
40	$22,836 \pm 0,697$	$0,657 \pm 0,025$	$0,572 \pm 0,007$

E.3.1.2 Intervalo de confiança do consumo de memória

Experimento com variação no número de atributos (ATT)

ATT	CQL	*NaiveSubseq*	*IncEndseq*
8	$46,075 \pm 0,062$	$24,851 \pm 0,051$	$24,854 \pm 0,074$
10	$46,699 \pm 0,075$	$25,740 \pm 0,044$	$25,950 \pm 0,056$
12	$47,897 \pm 0,041$	$25,830 \pm 0,021$	$25,857 \pm 0,049$
14	$47,863 \pm 0,068$	$25,695 \pm 0,059$	$25,840 \pm 0,044$
16	$48,115 \pm 0,024$	$25,917 \pm 0,052$	$25,982 \pm 0,031$

Experimento com variação no número de sequências (NSQ)

NSQ	CQL	*NaiveSubseq*	*IncEndseq*
4	$25,218 \pm 0,037$	$21,887 \pm 0,038$	$21,895 \pm 0,071$
8	$31,979 \pm 0,038$	$22,886 \pm 0,038$	$22,868 \pm 0,057$
16	$46,075 \pm 0,062$	$24,851 \pm 0,051$	$24,854 \pm 0,074$
24	$61,323 \pm 0,038$	$27,288 \pm 0,051$	$27,301 \pm 0,029$
32	$80,207 \pm 0,014$	$30,559 \pm 0,060$	$30,504 \pm 0,073$

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	*NaiveSubseq*	*IncEndseq*
10	$23,116 \pm 0,058$	$21,681 \pm 0,101$	$21,716 \pm 0,026$
20	$30,842 \pm 0,205$	$22,883 \pm 0,070$	$22,882 \pm 0,066$
30	$46,075 \pm 0,062$	$24,851 \pm 0,051$	$24,854 \pm 0,074$
40	$68,403 \pm 0,141$	$28,067 \pm 0,045$	$28,131 \pm 0,026$
50	$94,558 \pm 0,036$	$32,590 \pm 0,083$	$32,519 \pm 0,046$
60	$123,046 \pm 0,935$	$37,845 \pm 0,088$	$37,820 \pm 0,137$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	*NaiveSubseq*	*IncEndseq*
1	$53,304 \pm 0,032$	$24,943 \pm 0,026$	$24,962 \pm 0,015$
10	$46,075 \pm 0,062$	$24,851 \pm 0,051$	$24,854 \pm 0,074$
20	$43,257 \pm 0,077$	$24,711 \pm 0,066$	$24,843 \pm 0,062$
30	$40,526 \pm 0,015$	$24,679 \pm 0,062$	$24,831 \pm 0,061$
40	$35,603 \pm 0,079$	$24,661 \pm 0,013$	$24,379 \pm 0,073$

E.3.2 Experimentos com dados reais

E.3.2.1 Intervalo de confiança do tempo de execução

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	*NaiveSubseq*	*IncEndseq*
10	$5,087 \pm 0,071$	$0,619 \pm 0,019$	$0,640 \pm 0,035$
20	$15,930 \pm 0,179$	$0,893 \pm 0,022$	$0,922 \pm 0,013$
30	$32,911 \pm 0,524$	$1,181 \pm 0,036$	$1,230 \pm 0,035$
40	$55,281 \pm 0,728$	$1,461 \pm 0,026$	$1,505 \pm 0,010$
50	$83,803 \pm 0,938$	$1,759 \pm 0,040$	$1,816 \pm 0,022$
60	$116,802 \pm 1,736$	$2,057 \pm 0,036$	$2,129 \pm 0,017$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	*NaiveSubseq*	*IncEndseq*
1	$38,686 \pm 0,685$	$1,307 \pm 0,022$	$1,369 \pm 0,035$
10	$32,911 \pm 0,524$	$1,181 \pm 0,036$	$1,230 \pm 0,035$
20	$27,322 \pm 0,453$	$1,043 \pm 0,035$	$1,076 \pm 0,040$
30	$22,471 \pm 0,298$	$0,918 \pm 0,036$	$0,942 \pm 0,024$
40	$17,661 \pm 0,272$	$0,789 \pm 0,035$	$0,802 \pm 0,035$

E.3.2.2 Intervalo de confiança do consumo de memória

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	*NaiveSubseq*	*IncEndseq*
10	$22,832 \pm 0,049$	$21,931 \pm 0,072$	$21,956 \pm 0,035$
20	$22,978 \pm 0,020$	$21,912 \pm 0,085$	$21,928 \pm 0,054$
30	$23,025 \pm 0,053$	$22,001 \pm 0,033$	$21,918 \pm 0,061$
40	$23,275 \pm 0,035$	$22,003 \pm 0,034$	$22,001 \pm 0,014$
50	$23,627 \pm 0,046$	$21,988 \pm 0,085$	$21,973 \pm 0,079$
60	$24,117 \pm 0,076$	$22,027 \pm 0,045$	$21,938 \pm 0,073$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	*NaiveSubseq*	*IncEndseq*
1	$23,060 \pm 0,043$	$21,994 \pm 0,036$	$21,959 \pm 0,082$
10	$23,025 \pm 0,053$	$22,001 \pm 0,033$	$21,918 \pm 0,061$
20	$22,968 \pm 0,080$	$21,959 \pm 0,065$	$21,947 \pm 0,063$
30	$22,945 \pm 0,026$	$21,986 \pm 0,051$	$21,972 \pm 0,040$
40	$22,923 \pm 0,081$	$21,946 \pm 0,076$	$21,959 \pm 0,066$

E.4 Experimentos com o operador MAXSEQ

Esta seção apresenta os intervalos de confiança dos experimentos com o operador **MAXSEQ**. Os experimentos compararam o tempo de execução e o consumo de memória do operador **MAXSEQ** (processado pelo algoritmo *FilterByLength*) e sua equivalência em CQL.

E.4.1 Experimentos com dados sintéticos

E.4.1.1 Intervalo de confiança do tempo de execução

Experimento com variação no número de atributos (ATT)

ATT	CQL	**MAXSEQ**
8	$3,839 \pm 0,889$	$0,206 \pm 0,012$
10	$5,038 \pm 0,735$	$0,207 \pm 0,021$
12	$5,939 \pm 0,806$	$0,235 \pm 0,008$
14	$6,735 \pm 0,798$	$0,250 \pm 0,011$
16	$7,857 \pm 0,913$	$0,257 \pm 0,010$

Experimento com variação no número de sequências (NSQ)

NSQ	CQL	**MAXSEQ**
4	$1,143 \pm 0,209$	$0,125 \pm 0,014$
8	$2,083 \pm 0,370$	$0,151 \pm 0,015$
16	$3,839 \pm 0,889$	$0,206 \pm 0,012$
24	$6,091 \pm 1,054$	$0,266 \pm 0,009$
32	$8,405 \pm 1,325$	$0,327 \pm 0,012$

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	**MAXSEQ**
10	$0,995 \pm 0,191$	$0,131 \pm 0,035$
20	$2,097 \pm 0,459$	$0,153 \pm 0,039$
30	$3,839 \pm 0,889$	$0,206 \pm 0,012$
40	$4,919 \pm 1,006$	$0,198 \pm 0,041$
50	$7,962 \pm 1,581$	$0,280 \pm 0,012$
60	$8,912 \pm 2,467$	$0,282 \pm 0,067$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	**MAXSEQ**
1	$7,209 \pm 0,741$	$0,213 \pm 0,006$
10	$3,839 \pm 0,889$	$0,206 \pm 0,012$
20	$4,033 \pm 0,436$	$0,204 \pm 0,012$
30	$3,276 \pm 0,348$	$0,198 \pm 0,011$
40	$3,016 \pm 0,271$	$0,190 \pm 0,010$

Experimento com variação no comprimento máximo (MAX)

MAX	CQL	**MAXSEQ**
2	$4,716 \pm 0,430$	$0,210 \pm 0,005$
4	$4,730 \pm 0,434$	$0,204 \pm 0,005$
6	$4,624 \pm 0,420$	$0,205 \pm 0,009$
8	$4,795 \pm 0,363$	$0,209 \pm 0,012$
10	$3,839 \pm 0,889$	$0,206 \pm 0,012$

E.4.1.2 Intervalo de confiança do consumo de memória

Experimento com variação no número de atributos (ATT)

ATT	CQL	**MAXSEQ**
8	$25,471 \pm 0,031$	$21,763 \pm 0,050$
10	$25,955 \pm 0,066$	$21,722 \pm 0,066$
12	$26,284 \pm 0,039$	$21,760 \pm 0,066$
14	$26,975 \pm 0,047$	$21,771 \pm 0,055$
16	$27,153 \pm 0,031$	$21,798 \pm 0,029$

Experimento com variação no número de sequências (NSQ)

NSQ	CQL	**MAXSEQ**
4	$23,381 \pm 0,084$	$21,708 \pm 0,079$
8	$24,090 \pm 0,027$	$21,781 \pm 0,048$
16	$25,471 \pm 0,031$	$21,763 \pm 0,050$
24	$27,379 \pm 0,010$	$21,931 \pm 0,040$
32	$29,038 \pm 0,025$	$21,889 \pm 0,064$

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	**MAXSEQ**
10	$23,431 \pm 0,094$	$21,745 \pm 0,049$
20	$24,339 \pm 0,124$	$21,720 \pm 0,037$
30	$25,471 \pm 0,031$	$21,763 \pm 0,050$
40	$26,479 \pm 0,053$	$21,713 \pm 0,064$
50	$27,291 \pm 0,078$	$21,763 \pm 0,050$
60	$28,223 \pm 0,061$	$21,706 \pm 0,057$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	**MAXSEQ**
1	$26,166 \pm 0,065$	$21,752 \pm 0,029$
10	$25,471 \pm 0,031$	$21,763 \pm 0,050$
20	$24,862 \pm 0,009$	$21,797 \pm 0,022$
30	$24,602 \pm 0,098$	$21,756 \pm 0,079$
40	$24,212 \pm 0,036$	$21,716 \pm 0,053$

Experimento com variação no comprimento máximo (MAX)

MAX	CQL	**MAXSEQ**
2	$25,564 \pm 0,099$	$21,764 \pm 0,047$
4	$25,637 \pm 0,038$	$21,806 \pm 0,024$
6	$25,670 \pm 0,032$	$21,728 \pm 0,076$
8	$25,579 \pm 0,031$	$21,734 \pm 0,040$
10	$25,471 \pm 0,031$	$21,763 \pm 0,050$

E.4.2 Experimentos com dados reais

E.4.2.1 Intervalo de confiança do tempo de execução

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	**MAXSEQ**
10	$3,104 \pm 0,040$	$0,510 \pm 0,012$
20	$5,899 \pm 0,155$	$0,652 \pm 0,028$
30	$8,627 \pm 0,206$	$0,789 \pm 0,030$
40	$11,080 \pm 0,029$	$0,879 \pm 0,024$
50	$14,125 \pm 0,365$	$0,982 \pm 0,021$
60	$16,498 \pm 0,424$	$1,076 \pm 0,026$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	**MAXSEQ**
1	$9,885 \pm 0,203$	$0,814 \pm 0,025$
10	$8,627 \pm 0,206$	$0,789 \pm 0,030$
20	$7,272 \pm 0,185$	$0,694 \pm 0,017$
30	$5,944 \pm 0,099$	$0,653 \pm 0,020$
40	$4,732 \pm 0,116$	$0,589 \pm 0,021$

Experimento com variação no comprimento máximo (MAX)

MAX	CQL	**MAXSEQ**
2	$8,066 \pm 0,038$	$0,740 \pm 0,020$
4	$8,505 \pm 0,151$	$0,736 \pm 0,004$
6	$8,647 \pm 0,174$	$0,743 \pm 0,002$
8	$8,617 \pm 0,181$	$0,758 \pm 0,025$
10	$8,627 \pm 0,206$	$0,789 \pm 0,030$

E.4.2.2 Intervalo de confiança do consumo de memória

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	**MAXSEQ**
10	$23,607 \pm 0,072$	$21,988 \pm 0,020$
20	$23,592 \pm 0,072$	$21,946 \pm 0,059$
30	$23,664 \pm 0,028$	$21,987 \pm 0,020$
40	$23,650 \pm 0,065$	$21,942 \pm 0,092$
50	$23,670 \pm 0,070$	$21,960 \pm 0,063$
60	$23,695 \pm 0,025$	$22,014 \pm 0,026$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	**MAXSEQ**
1	$23,657 \pm 0,024$	$21,944 \pm 0,097$
10	$23,664 \pm 0,028$	$21,987 \pm 0,020$
20	$23,605 \pm 0,072$	$21,942 \pm 0,051$
30	$23,630 \pm 0,044$	$21,951 \pm 0,086$
40	$23,615 \pm 0,054$	$21,872 \pm 0,071$

Experimento com variação no comprimento máximo (MAX)

MAX	CQL	**MAXSEQ**
2	$23,610 \pm 0,059$	$21,956 \pm 0,025$
4	$23,652 \pm 0,040$	$21,967 \pm 0,032$
6	$23,657 \pm 0,038$	$21,978 \pm 0,040$
8	$23,603 \pm 0,085$	$21,986 \pm 0,055$
10	$23,664 \pm 0,028$	$21,987 \pm 0,020$

E.5 Experimentos com o operador MINSEQ

Esta seção apresenta os intervalos de confiança dos experimentos com o operador **MINSEQ**. Os experimentos compararam o tempo de execução e o consumo de memória do operador **MINSEQ** (processado pelo algoritmo *FilterByLength*) e sua equivalência em CQL.

E.5.1 Experimentos com dados sintéticos

E.5.1.1 Intervalo de confiança do tempo de execução

Experimento com variação no número de atributos (ATT)

ATT	CQL	**MAXSEQ**
8	$3,849 \pm 0,639$	$0,237 \pm 0,023$
10	$4,949 \pm 0,760$	$0,254 \pm 0,024$
12	$5,860 \pm 0,846$	$0,267 \pm 0,023$
14	$6,720 \pm 0,918$	$0,278 \pm 0,024$
16	$7,988 \pm 0,955$	$0,292 \pm 0,026$

Experimento com variação no número de sequências (NSQ)

NSQ	CQL	**MAXSEQ**
4	$1,092 \pm 0,188$	$0,123 \pm 0,022$
8	$2,012 \pm 0,346$	$0,161 \pm 0,024$
16	$3,849 \pm 0,639$	$0,237 \pm 0,023$
24	$5,904 \pm 0,987$	$0,315 \pm 0,029$
32	$8,186 \pm 1,271$	$0,397 \pm 0,035$

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	**MAXSEQ**
10	$0,865 \pm 0,111$	$0,127 \pm 0,034$
20	$2,100 \pm 0,333$	$0,164 \pm 0,039$
30	$3,849 \pm 0,639$	$0,237 \pm 0,023$
40	$5,373 \pm 0,805$	$0,260 \pm 0,052$
50	$7,839 \pm 1,387$	$0,360 \pm 0,034$
60	$10,115 \pm 2,771$	$0,417 \pm 0,107$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	**MAXSEQ**
1	$6,363 \pm 0,717$	$0,246 \pm 0,022$
10	$3,849 \pm 0,639$	$0,237 \pm 0,023$
20	$3,961 \pm 0,425$	$0,222 \pm 0,024$
30	$3,038 \pm 0,269$	$0,202 \pm 0,024$
40	$2,943 \pm 0,296$	$0,195 \pm 0,021$

Experimento com variação no comprimento máximo (MIN)

MIN	CQL	**MAXSEQ**
2	$3,849 \pm 0,639$	$0,237 \pm 0,023$
3	$4,721 \pm 0,466$	$0,238 \pm 0,027$
4	$4,731 \pm 0,435$	$0,238 \pm 0,025$
5	$4,755 \pm 0,402$	$0,240 \pm 0,022$
6	$4,797 \pm 0,437$	$0,232 \pm 0,019$

E.5.1.2 Intervalo de confiança do consumo de memória

Experimento com variação no número de atributos (ATT)

ATT	CQL	**MAXSEQ**
8	$25,660 \pm 0,016$	$21,780 \pm 0,038$
10	$26,057 \pm 0,045$	$21,789 \pm 0,003$
12	$26,617 \pm 0,025$	$21,759 \pm 0,055$
14	$26,842 \pm 0,091$	$21,806 \pm 0,045$
16	$27,568 \pm 0,025$	$21,778 \pm 0,066$

Experimento com variação no número de sequências (NSQ)

NSQ	CQL	**MAXSEQ**
4	$23,392 \pm 0,083$	$21,740 \pm 0,026$
8	$24,127 \pm 0,023$	$21,781 \pm 0,078$
16	$25,660 \pm 0,016$	$21,780 \pm 0,038$
24	$27,437 \pm 0,064$	$22,141 \pm 0,035$
32	$29,572 \pm 0,086$	$22,469 \pm 0,030$

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	**MAXSEQ**
10	$23,442 \pm 0,063$	$21,737 \pm 0,042$
20	$24,441 \pm 0,042$	$21,709 \pm 0,107$
30	$25,660 \pm 0,016$	$21,780 \pm 0,038$
40	$26,649 \pm 0,025$	$21,821 \pm 0,068$
50	$27,616 \pm 0,042$	$21,959 \pm 0,093$
60	$28,725 \pm 0,139$	$22,076 \pm 0,037$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	**MAXSEQ**
1	$26,222 \pm 0,041$	$21,781 \pm 0,035$
10	$25,660 \pm 0,016$	$21,780 \pm 0,038$
20	$25,075 \pm 0,060$	$21,713 \pm 0,088$
30	$25,040 \pm 0,087$	$21,787 \pm 0,019$
40	$24,576 \pm 0,045$	$21,780 \pm 0,021$

Experimento com variação no comprimento máximo (MIN)

MIN	CQL	**MAXSEQ**
2	$25,660 \pm 0,016$	$21,780 \pm 0,038$
3	$25,651 \pm 0,075$	$21,745 \pm 0,082$
4	$25,656 \pm 0,036$	$21,713 \pm 0,060$
5	$25,543 \pm 0,045$	$21,745 \pm 0,028$
6	$25,502 \pm 0,062$	$21,707 \pm 0,080$

E.5.2 Experimentos com dados reais

E.5.2.1 Intervalo de confiança do tempo de execução

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	**MAXSEQ**
10	$2,480 \pm 0,032$	$0,508 \pm 0,018$
20	$4,754 \pm 0,066$	$0,623 \pm 0,014$
30	$6,980 \pm 0,029$	$0,741 \pm 0,025$
40	$9,197 \pm 0,073$	$0,852 \pm 0,024$
50	$11,420 \pm 0,090$	$0,946 \pm 0,015$
60	$13,622 \pm 0,077$	$1,035 \pm 0,025$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	**MAXSEQ**
1	$8,057 \pm 0,057$	$0,793 \pm 0,023$
10	$6,980 \pm 0,029$	$0,741 \pm 0,025$
20	$5,838 \pm 0,047$	$0,679 \pm 0,021$
30	$4,784 \pm 0,034$	$0,638 \pm 0,034$
40	$3,826 \pm 0,029$	$0,558 \pm 0,015$

Experimento com variação no comprimento máximo (MIN)

MIN	CQL	**MAXSEQ**
2	$6,980 \pm 0,029$	$0,741 \pm 0,025$
3	$5,900 \pm 0,012$	$0,685 \pm 0,021$
4	$5,645 \pm 0,062$	$0,657 \pm 0,012$
5	$5,305 \pm 0,028$	$0,640 \pm 0,015$
6	$5,257 \pm 0,050$	$0,633 \pm 0,014$

E.5.2.2 Intervalo de confiança do consumo de memória

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	**MAXSEQ**
10	$23,590 \pm 0,070$	$21,976 \pm 0,022$
20	$23,642 \pm 0,035$	$21,937 \pm 0,066$
30	$23,643 \pm 0,036$	$21,959 \pm 0,065$
40	$23,698 \pm 0,023$	$21,962 \pm 0,033$
50	$23,630 \pm 0,066$	$22,006 \pm 0,034$
60	$23,623 \pm 0,041$	$21,934 \pm 0,091$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	**MAXSEQ**
1	$23,659 \pm 0,031$	$21,961 \pm 0,083$
10	$23,643 \pm 0,036$	$21,959 \pm 0,065$
20	$23,668 \pm 0,059$	$21,969 \pm 0,012$
30	$23,656 \pm 0,021$	$21,965 \pm 0,041$
40	$23,627 \pm 0,043$	$21,955 \pm 0,049$

Experimento com variação no comprimento máximo (MIN)

MIN	CQL	**MAXSEQ**
2	$23,643 \pm 0,036$	$21,959 \pm 0,065$
3	$23,576 \pm 0,094$	$21,947 \pm 0,064$
4	$23,655 \pm 0,052$	$21,937 \pm 0,073$
5	$23,588 \pm 0,056$	$21,956 \pm 0,038$
6	$23,571 \pm 0,096$	$21,916 \pm 0,074$

E.6 Experimentos com o operador BESTSEQ

Esta seção apresenta os intervalos de confiança dos experimentos com o operador **BESTSEQ**. Os experimentos compararam o tempo de execução e o consumo de memória do operador **BESTSEQ** (processado pelos algoritmos *GetBestSeq*, *IncBestSeq* e *IncBestSeq* com poda) e sua equivalência em CQL.

E.6.1 Experimentos com dados sintéticos

E.6.1.1 Intervalo de confiança do tempo de execução

Experimento com variação no número de atributos (ATT)

ATT	CQL	*GetBestSeq*	*IncBestSeq*	
			Sem poda	Com poda
8	$2.794,023 \pm 78,990$	$2,670 \pm 0,157$	$1,468 \pm 0,130$	$0,456 \pm 0,027$
10	$3.109,364 \pm 158,100$	$2,772 \pm 0,115$	$1,560 \pm 0,072$	$0,506 \pm 0,059$
12	$3.371,343 \pm 176,615$	$2,660 \pm 0,205$	$1,725 \pm 0,074$	$0,521 \pm 0,046$
14	$3.971,764 \pm 211,072$	$2,832 \pm 0,058$	$1,822 \pm 0,046$	$0,560 \pm 0,068$
16	$4.287,593 \pm 204,990$	$2,829 \pm 0,071$	$1,930 \pm 0,079$	$0,595 \pm 0,058$

Experimento com variação no número de sequências (NSQ)

NSQ	CQL	*GetBestSeq*	*IncBestSeq*	
			Sem poda	Com poda
4	$164,351 \pm 2,781$	$0,316 \pm 0,034$	$0,400 \pm 0,014$	$0,255 \pm 0,022$
8	$684,028 \pm 12,913$	$0,715 \pm 0,041$	$0,635 \pm 0,077$	$0,379 \pm 0,054$
16	$2.794,023 \pm 78,990$	$2,670 \pm 0,157$	$1,468 \pm 0,130$	$0,456 \pm 0,027$
24	$7.265,717 \pm 138,827$	$4,584 \pm 0,219$	$1,544 \pm 0,071$	$0,739 \pm 0,039$
32	$13.919,172 \pm 436,292$	$8,190 \pm 0,395$	$2,051 \pm 0,143$	$0,946 \pm 0,029$

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	*GetBestSeq*	*IncBestSeq*	
			Sem poda	Com poda
10	$1.807,453 \pm 361,452$	$1,997 \pm 0,363$	$0,561 \pm 0,074$	$0,345 \pm 0,068$
20	$2.570,784 \pm 510,191$	$2,500 \pm 0,277$	$1,073 \pm 0,159$	$0,442 \pm 0,126$
30	$2.794,023 \pm 78,990$	$2,670 \pm 0,157$	$1,468 \pm 0,130$	$0,456 \pm 0,027$
40	$3.596,439 \pm 674,712$	$3,548 \pm 0,560$	$2,248 \pm 0,417$	$0,615 \pm 0,096$
50	$3.854,861 \pm 171,380$	$3,409 \pm 0,211$	$2,668 \pm 0,177$	$0,644 \pm 0,023$
60	$4.959,318 \pm 960,600$	$4,328 \pm 0,826$	$3,788 \pm 0,731$	$0,900 \pm 0,230$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	*GetBestSeq*	*IncBestSeq*	
			Sem poda	Com poda
1	$3.156,650 \pm 112,095$	$2,635 \pm 0,109$	$4,880 \pm 0,229$	$0,837 \pm 0,085$
10	$2.794,023 \pm 78,990$	$2,670 \pm 0,157$	$1,468 \pm 0,130$	$0,456 \pm 0,027$
20	$2.810,530 \pm 111,979$	$2,820 \pm 0,237$	$0,997 \pm 0,028$	$0,386 \pm 0,007$
30	$2.632,411 \pm 73,046$	$2,656 \pm 0,114$	$0,895 \pm 0,119$	$0,366 \pm 0,040$
40	$2.264,720 \pm 66,225$	$2,230 \pm 0,066$	$0,742 \pm 0,038$	$0,324 \pm 0,017$

Experimento com variação no número de regras (RUL)

RUL	CQL	*GetBestSeq*	*IncBestSeq*	
			Sem poda	Com poda
4	$1.413,677 \pm 86,752$	$1,539 \pm 0,076$	$0,983 \pm 0,043$	$0,412 \pm 0,024$
8	$2.794,023 \pm 78,990$	$2,670 \pm 0,157$	$1,468 \pm 0,130$	$0,456 \pm 0,027$
16	$5.478,527 \pm 185,682$	$5,277 \pm 0,273$	$2,407 \pm 0,146$	$0,559 \pm 0,009$
24	$8.069,105 \pm 377,306$	$8,768 \pm 0,366$	$3,368 \pm 0,305$	$0,803 \pm 0,110$
32	$10.691,300 \pm 471,847$	$11,611 \pm 0,796$	$4,556 \pm 0,543$	$1,024 \pm 0,036$

Experimento com variação no nível de preferência máximo (LEV)

LEV	CQL	*GetBestSeq*	*IncBestSeq*	
			Sem poda	Com poda
1	$2.862,854 \pm 115,072$	$2,594 \pm 0,088$	$1,325 \pm 0,091$	$0,479 \pm 0,106$
2	$2.794,023 \pm 78,990$	$2,670 \pm 0,157$	$1,468 \pm 0,130$	$0,456 \pm 0,027$
3	$2.844,337 \pm 62,429$	$2,676 \pm 0,056$	$1,535 \pm 0,058$	$0,446 \pm 0,036$
4	$2.826,283 \pm 57,311$	$2,726 \pm 0,153$	$1,724 \pm 0,116$	$0,429 \pm 0,020$
5	$2.811,237 \pm 68,812$	$2,760 \pm 0,162$	$1,699 \pm 0,079$	$0,471 \pm 0,057$
6	$2.843,170 \pm 77,634$	$2,739 \pm 0,150$	$1,686 \pm 0,079$	$0,450 \pm 0,024$

E.6.1.2 Intervalo de confiança do consumo de memória

Experimento com variação no número de atributos (ATT)

ATT	CQL	*GetBestSeq*	*IncBestSeq*	
			Sem poda	Com poda
8	$4.321,996 \pm 0,699$	$21,728 \pm 0,037$	$22,347 \pm 0,068$	$21,742 \pm 0,071$
10	$4.354,756 \pm 0,523$	$21,711 \pm 0,049$	$22,480 \pm 0,026$	$21,887 \pm 0,041$
12	$4.422,206 \pm 0,920$	$21,739 \pm 0,024$	$22,632 \pm 0,032$	$21,889 \pm 0,081$
14	$7.241,272 \pm 1,782$	$21,731 \pm 0,042$	$22,741 \pm 0,052$	$21,952 \pm 0,017$
16	$7.245,365 \pm 0,659$	$21,716 \pm 0,077$	$22,832 \pm 0,033$	$22,018 \pm 0,074$

Experimento com variação no número de sequências (NSQ)

NSQ	CQL	*GetBestSeq*	*IncBestSeq*	
			Sem poda	Com poda
4	$266,802 \pm 0,079$	$21,648 \pm 0,066$	$21,695 \pm 0,039$	$21,706 \pm 0,035$
8	$1.092,044 \pm 0,875$	$21,691 \pm 0,064$	$21,956 \pm 0,057$	$21,719 \pm 0,039$
16	$4.321,996 \pm 0,699$	$21,728 \pm 0,037$	$22,347 \pm 0,068$	$21,742 \pm 0,071$
24	$10.164,135 \pm 7,137$	$21,910 \pm 0,022$	$24,161 \pm 0,065$	$22,944 \pm 0,101$
32	$18.215,845 \pm 2,852$	$22,270 \pm 0,089$	$25,328 \pm 0,058$	$23,672 \pm 0,066$

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	*GetBestSeq*	*IncBestSeq*	
			Sem poda	Com poda
10	$4.123,012 \pm 0,136$	$21,694 \pm 0,077$	$21,737 \pm 0,033$	$21,677 \pm 0,087$
20	$4.268,186 \pm 2,584$	$21,750 \pm 0,012$	$21,902 \pm 0,028$	$21,740 \pm 0,037$
30	$4.321,996 \pm 0,699$	$21,728 \pm 0,037$	$22,347 \pm 0,068$	$21,742 \pm 0,071$
40	$4.370,814 \pm 1,397$	$21,692 \pm 0,073$	$22,798 \pm 0,064$	$22,011 \pm 0,073$
50	$4.415,445 \pm 0,698$	$21,700 \pm 0,060$	$23,210 \pm 0,079$	$22,310 \pm 0,032$
60	$4.460,747 \pm 1,779$	$21,702 \pm 0,069$	$23,535 \pm 0,073$	$22,447 \pm 0,075$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	*GetBestSeq*	*IncBestSeq*	
			Sem poda	Com poda
1	$4.419,719 \pm 0,151$	$21,723 \pm 0,047$	$22,331 \pm 0,035$	$21,810 \pm 0,027$
10	$4.321,996 \pm 0,699$	$21,728 \pm 0,037$	$22,347 \pm 0,068$	$21,742 \pm 0,071$
20	$4.288,416 \pm 1,884$	$21,678 \pm 0,095$	$22,181 \pm 0,028$	$21,743 \pm 0,046$
30	$4.249,170 \pm 0,398$	$21,706 \pm 0,084$	$22,076 \pm 0,013$	$21,754 \pm 0,035$
40	$3.985,081 \pm 53,069$	$21,621 \pm 0,066$	$22,019 \pm 0,073$	$21,746 \pm 0,021$

Experimento com variação no número de regras (RUL)

RUL	CQL	*GetBestSeq*	*IncBestSeq*	
			Sem poda	Com poda
4	$2.204,532 \pm 0,621$	$21,700 \pm 0,074$	$22,303 \pm 0,082$	$21,789 \pm 0,023$
8	$4.321,996 \pm 0,699$	$21,728 \pm 0,037$	$22,347 \pm 0,068$	$21,742 \pm 0,071$
16	$8.565,940 \pm 0,279$	$21,678 \pm 0,092$	$22,541 \pm 0,015$	$21,872 \pm 0,048$
24	$12.860,535 \pm 45,110$	$21,747 \pm 0,026$	$22,674 \pm 0,085$	$21,936 \pm 0,020$
32	$17.092,581 \pm 8,752$	$21,750 \pm 0,058$	$22,834 \pm 0,026$	$21,993 \pm 0,035$

Experimento com variação no nível de preferência máximo (LEV)

LEV	CQL	*GetBestSeq*	*IncBestSeq*	
			Sem poda	Com poda
1	$4.322,042 \pm 1,109$	$21,676 \pm 0,098$	$22,324 \pm 0,065$	$21,799 \pm 0,035$
2	$4.321,996 \pm 0,699$	$21,728 \pm 0,037$	$22,347 \pm 0,068$	$21,742 \pm 0,071$
3	$4.321,718 \pm 1,840$	$21,694 \pm 0,079$	$22,372 \pm 0,030$	$21,818 \pm 0,048$
4	$4.322,481 \pm 0,094$	$21,705 \pm 0,052$	$22,389 \pm 0,052$	$21,750 \pm 0,063$
5	$4.325,345 \pm 1,171$	$21,696 \pm 0,064$	$22,442 \pm 0,056$	$21,824 \pm 0,024$
6	$4.321,897 \pm 0,837$	$21,683 \pm 0,067$	$22,425 \pm 0,025$	$21,816 \pm 0,031$

E.6.2 Experimentos com dados reais

Os resultados sobre dados reais para a equivalência CQL considerando abrangências temporais maiores do que 30 apresentam o valor NaN ± NaN porque não foi possível executar a equivalência para estas variações do parâmetro RAN. A equivalência CQL do operador **BESTSEQ** é muito complexa e contém diversas relações temporárias. As abrangências temporais maiores geram um número maior de tuplas por instante que causa estouro de memória na execução da equivalência CQL, impossibilitando assim a execução da equivalência CQL nestes cenários.

E.6.2.1 Intervalo de confiança do tempo de execução

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	*GetBestSeq*	*IncBestSeq*	
			Sem poda	Com poda
10	$1.236,975 \pm 6,775$	$1,062 \pm 0,008$	$0,926 \pm 0,020$	$0,775 \pm 0,021$
20	$12.138,424 \pm 44,064$	$2,852 \pm 0,020$	$1,220 \pm 0,022$	$0,967 \pm 0,024$
30	$38.635,184 \pm 183,564$	$5,302 \pm 0,042$	$1,414 \pm 0,014$	$1,143 \pm 0,025$
40	NaN ± NaN	$8,020 \pm 0,042$	$1,635 \pm 0,024$	$1,294 \pm 0,024$
50	NaN ± NaN	$10,964 \pm 0,021$	$1,832 \pm 0,022$	$1,437 \pm 0,024$
60	NaN ± NaN	$13,959 \pm 0,106$	$2,030 \pm 0,042$	$1,583 \pm 0,044$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	*GetBestSeq*	*IncBestSeq*	
			Sem poda	Com poda
1	$56.606,232 \pm 455,166$	$6,391 \pm 0,047$	$1,690 \pm 0,034$	$1,326 \pm 0,018$
10	$38.635,184 \pm 183,564$	$5,302 \pm 0,042$	$1,414 \pm 0,014$	$1,143 \pm 0,025$
20	$26.617,184 \pm 71,631$	$4,059 \pm 0,035$	$1,258 \pm 0,020$	$1,013 \pm 0,021$
30	$19.102,213 \pm 117,160$	$3,113 \pm 0,027$	$1,144 \pm 0,031$	$0,947 \pm 0,029$
40	$15.240,338 \pm 82,815$	$2,485 \pm 0,035$	$0,955 \pm 0,016$	$0,814 \pm 0,039$

E.6.2.2 Intervalo de confiança do consumo de memória

Experimento com variação na abrangência temporal (RAN)

RAN	CQL	*GetBestSeq*	*IncBestSeq*	
			Sem poda	Com poda
10	$166,716 \pm 1,738$	$21,912 \pm 0,063$	$22,013 \pm 0,026$	$21,998 \pm 0,060$
20	$3.449,007 \pm 185,846$	$21,905 \pm 0,023$	$21,971 \pm 0,074$	$22,042 \pm 0,043$
30	$2.982,869 \pm 268,072$	$21,877 \pm 0,092$	$22,021 \pm 0,083$	$22,049 \pm 0,027$
40	NaN ± NaN	$21,884 \pm 0,063$	$22,013 \pm 0,052$	$22,052 \pm 0,017$
50	NaN ± NaN	$21,860 \pm 0,068$	$22,070 \pm 0,052$	$22,015 \pm 0,072$
60	NaN ± NaN	$21,929 \pm 0,045$	$22,087 \pm 0,024$	$22,087 \pm 0,037$

Experimento com variação no intervalo de deslocamento (SLI)

SLI	CQL	*GetBestSeq*	*IncBestSeq*	
			Sem poda	Com poda
1	$4.111,043 \pm 52,112$	$21,874 \pm 0,076$	$21,986 \pm 0,065$	$22,018 \pm 0,047$
10	$2.982,869 \pm 268,072$	$21,877 \pm 0,092$	$22,021 \pm 0,083$	$22,049 \pm 0,027$
20	$3.811,677 \pm 0,467$	$21,866 \pm 0,087$	$22,016 \pm 0,062$	$22,026 \pm 0,045$
30	$3.038,921 \pm 364,411$	$21,915 \pm 0,037$	$22,012 \pm 0,035$	$22,042 \pm 0,023$
40	$1.711,432 \pm 300,067$	$21,890 \pm 0,037$	$22,011 \pm 0,066$	$21,944 \pm 0,075$