Temporal Conditional Preference Queries on Streams

Marcos Roberto Ribeiro, Maria Camila, N Barioni, Sandra de Amo, Claudia Roncancio, Cyril Labbé

To cite this version:

Marcos Roberto Ribeiro, Maria Camila, N Barioni, Sandra de Amo, Claudia Roncancio, et al.. Temporal Conditional Preference Queries on Streams. 28th International Conference, DEXA 2017 Database and Expert Systems Applications, 2017, Lyon, France. pp.143-158. ⟨hal-01658631⟩

HAL Id: hal-01658631
https://hal.science/hal-01658631v1

Submitted on 7 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Temporal Conditional Preference Queries on Streams

Marcos Roberto Ribeiro[1,2], Maria Camila N. Barioni[2], Sandra de Amo[2], Claudia Roncancio[3], and Cyril Labbé[3]

[1] Instituto Federal de Minas Gerais, Bambuí, Brazil
marcos.ribeiro@ifmg.edu.br
[2] Universidade Federal de Uberlândia, Uberlândia, Brazil
{camila.barioni, deamo}@ufu.br
[3] Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000 Grenoble, France
{claudia.roncancio, cyril.labbe}@imag.fr

Abstract. Preference queries on data streams have been proved very useful for many application areas. Despite of the existence of research studies dedicated to this issue, they lack to support the use of an important implicit information of data streams, the temporal preferences. In this paper we define new operators and an algorithm for the efficient evaluation of temporal conditional preference queries on data streams. We also demonstrate how the proposed operators can be translated to the Continuous Query Language (CQL). The experiments performed show that our proposed operators have considerably superior performance when compared to the equivalent operations in CQL.

Keywords: Data streams, Preference queries, Temporal preferences

1 Introduction

There is a variety of application domains which data naturally occur in the form of a sequence of values, such as financial applications, sport players monitoring, telecommunications, web applications, sensor networks, among others. An especially useful model explored by many research works to deal with this type of data is the data stream model [2, 6, 10, 5]. Great part of these research works has focused on the development of new techniques to answer continuous queries efficiently [3, 9]. Other research works have been concerned with the evaluation of preferences in continuous queries to monitor for information that best fit the users wishes when processing data streams [7].

The research literature regarding this later issue is rich in works dealing with processing of continuous skyline queries where the preferences are simple independent preferences for minimum or maximum values over attributes [4, 8]. However, these works do not meet the needs of many domain applications that require the users to express conditional preferences. That is, those applications where the preferences over a data attribute can be affected by values of another data attribute. Moreover, they do not take advantage of the implicit temporal information of data streams to deal with temporal preferences.

Temporal preferences may allow users to express how an instant of time may influence his preferences at another time moment. Therefore, it allows an application to employ continuous queries to find sequences of patterns in data according to user preferences. For example, considering a soccer game where players are monitored, it is possible for a coach to check if some player behavior matches certain preferences before making an intervention in the game. Thus, it is possible to evaluate queries such as "Which are the best players considering that if a player was at defensive intermediary, then I prefer that this player go to the middle-field instead of staying in the same place?".

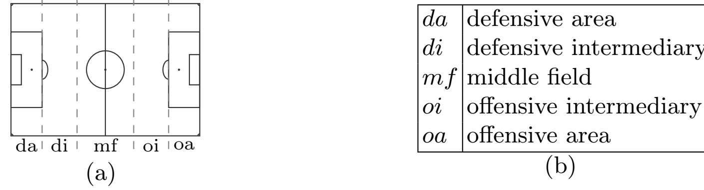
The evaluation of continuous queries with *conditional preference* has already been explored in previous works [1, 12]. Nevertheless, to the best of our knowledge, the support for temporal conditional preference queries on data streams began to be exploited recently in our previous paper [14]. The main goal herein is to present an extension for the *Continuous Query Language (CQL)* that is specially tailored to efficiently process temporal conditional preference queries on data streams. Although CQL is an expressive SQL-based declarative language [3, 2], it was not designed to deal with temporal preferences. In order to cope with this issue, we define appropriate data structures for keeping the temporal order of tuples and new specific operators for selecting the best sequences according to users preferences. These new features allow our approach to achieve a considerable better performance.

Main Contributions. The main contributions of this paper can be summarized as follows: **(1)** The definition of new operators for the evaluation of continuous queries containing temporal conditional preferences; **(2)** A new and efficient incremental method for the evaluation of the proposed preference operator; **(3)** A detailed demonstration of the CQL equivalences for the proposed operators; **(4)** An extensive set of experiments showing that our proposed operators have better performance than their equivalent operations in CQL.

In the following sections, we introduce a motivating example. In addition, we present the fundamental concepts regarding the temporal conditional preferences and describe the operators and algorithms proposed for the evaluation of temporal conditional preference queries on data streams. We also discuss the equivalent operations in CQL and the experimental results. Finally, at the end we give the conclusions of this paper.

2 A Motivating Example

Philip is a soccer coach who uses technology to make decisions. He has access to an information system that provides real-time data about players during a match. The information available is the stream `Event(PID, PC, PE)` containing the match events along with an identification of the involved players. The attributes of the stream `Event` are: player identification (`PID`), current place (`PC`) and the match event (`PE`). The values for `PC` are the regions showed in Fig. 1. The match events (`PE`) are: carrying the ball (*ca*), completed pass (*cp*), dribble (*dr*), losing the ball (*lb*), non-completed pass (*ncp*) and pass reception (*re*).



da	defensive area
di	defensive intermediary
mf	middle field
oi	offensive intermediary
oa	offensive area

(b)

Fig. 1. Values for attribute `PC`: (a) Field division; (b) Values description.

Based on his experience, Philip has the following preferences: **[P1]** If the previous in-game event was a pass reception then I prefer a dribble than a completed pass, independent of the place; **[P2]** Completed passes are better than non-completed passes; **[P3]** If all previous in-game events were in the middle-field then I prefer events in the middle-field than events in the defensive intermediary.

These preferences can be used by Philip to submit the following continuous query to the information system: **[Q1]** Every instant, give me the in-game event sequences that best fit my preferences in the last six seconds. When the coach says "best fit my preferences" it means that if a data item X is in the query result then it is not possible to find another response better than X according to his preferences. The query answers could help the coach to give special attention to a particular player having behavior fitting the coach preferences.

3 Background: Temporal Conditional Preferences

Our proposed language, called *StreamPref*, uses the formalism introduced in our previous work [14] to compare sequences of tuples. Let $\mathbf{Dom}(A)$ be the domain of the attribute A. Let $R(A_1, ..., A_l)$ be a relational schema. The set of all tuples over R is denoted by $\mathbf{Tup}(R) = \mathbf{Dom}(A_1) \times ... \times \mathbf{Dom}(A_l)$. A sequence $s = \langle t_1, ..., t_n \rangle$ over R is an ordered set of tuples, such that $t_i \in \mathbf{Tup}(R)$ for all $i \in \{1, ..., n\}$. The length of a sequence s is denoted by $|s|$. A tuple in the position i of a sequence s is denoted by $s[i]$ and $s[i].A$ represents the attribute A in the position i of s. We use $s[i, j]$ to denote the subsequence $s' = \langle t_i, ..., t_j \rangle$ of $s = \langle t_1, ..., t_n \rangle$ such that $1 \leq i \leq n$ and $i \leq j \leq n$. The concatenation s'' of two sequences $s = \langle t_1, ..., t_n \rangle$ and $s' = \langle t'_1, ..., t'_{n'} \rangle$, denoted by $s + s'$, is $s'' = \langle t_1, ..., t_n, t'_1, ..., t'_{n'} \rangle$. We denote by $\mathbf{Seq}(R)$ the set of all possible sequences over R.

Our preference model uses StreamPref Temporal Logic (STL) formulas composed by propositions in the form $A\theta a$, where $a \in \mathbf{Dom}(A)$ and $\theta \in \{<, \leq, =, \neq, \geq, >\}$ (see Definition 1). Let $Q(A)$ be a proposition, $S_{Q(A)} = \{a \in \mathbf{Dom}(A) \mid a \models Q(A)\}$ denotes the set of values satisfying $Q(A)$.

Definition 1 (STL Formulas). *The STL formulas are defined as follows: **(1) true** and **false** are STL formulas; **(2)** If F is a proposition then F is a STL formula; **(3)** If F and G are STL formulas then $(F \wedge G)$, $(F \vee G)$, $(F$ **Since** $G)$, $\neg F$ and $\neg G$ are STL formulas.*

A STL formula F is satisfied by a sequence $s = \langle t_1, ..., t_n \rangle$ at a position $i \in \{1, ..., n\}$, denoted by $(s, i) \models F$, according to the following conditions: **(1)** $(s, i) \models Q(A)$ if and only if $s[i].A \models Q(A)$; **(2)** $(s, i) \models F \wedge G$ if and only

if $(s,i) \models F$ and $(s,i) \models G$; **(3)** $(s,i) \models F \vee G$ if and only if $(s,i) \models F$ or $(s,i) \models G$; **(4)** $(s,i) \models \neg F$ if and only if $(s,i) \not\models F$; **(5)** $(s,i) \models (F\ \textbf{since}\ G)$ if and only if there exists j where $1 \leq j < i$ and $(s,j) \models G$ and $(s,k) \models F$ for all $k \in \{j+1,...,i\}$. The **true** formula is always satisfied and the **false** formula is never satisfied. We also define the following derived formulas:

Prev $Q(A)$**:** Equivalent to $(\textbf{false since}\ Q(A))$, $(s,i) \models$ **Prev** $Q(A)$ if and only if $i > 1$ and $(s,i-1) \models F$;

SomePrev $Q(A)$**:** Equivalent to $(\textbf{true since}\ Q(A))$, $(s,i) \models$ **SomePrev** $Q(A)$ if and only if there exists j such that $1 \leq j < i$ and $(s,i) \models Q(A)$;

AllPrev $Q(A)$**:** Equivalent to $\neg(\textbf{SomePrev}\neg Q(A))$, $(s,i) \models$ **AllPrev** $Q(A)$ if and only if $(s,j) \models F$ for all $j \in \{1,...,i-1\}$;

First: Equivalent to $\neg(\textbf{Prev}(\textbf{true}))$, $(s,i) \models$ **First** if and only if $i = 1$.

The Definition 2 formalizes the *temporal conditions* used by Definition 3 (*tcp-rules* and *tcp-theories*).

Definition 2 (Temporal Conditions). *A temporal condition is a formula $F = F_1 \wedge ... \wedge F_n$, where $F_1,...,F_n$ are propositions or derived formulas. The temporal components of F, denoted by $F^{\leftarrow}$, is the conjunction of all derived formulas in F. The non-temporal components of F, denoted by $F^{\bullet}$, is the conjunction of all propositions in F and not present in $F^{\leftarrow}$. We use $\textbf{\textit{Att}}(F)$ to denote the attributes appearing in F.*

Definition 3 (TCP-Rules and TCP-Theories). *Let R be a relational schema. A temporal conditional preference rule, or tcp-rule, is an expression in the format $\varphi : C_\varphi \to Q^+_\varphi \succ Q^-_\varphi[W_\varphi]$, where: **(1)** The propositions Q^+_φ and Q^-_φ represent the preferred values and non-preferred values for the preference attribute A_φ, respectively, such that $S_{Q^+_\varphi} \cap S_{Q^-_\varphi} = \emptyset$; **(2)** $W_\varphi \subset R$ is the set of indifferent attributes such that $A_\varphi \notin W_\varphi$; **(3)** C_φ is a temporal condition such that $\textbf{\textit{Att}}(C^\bullet_\varphi) \cap (\{A_\varphi\} \cup W_\varphi) = \emptyset$. A temporal conditional preference theory, or tcp-theory, is a finite set of tcp-rules.*

Example 1. Consider the coach preferences of Section 2. We can express them by the tcp-theory $\Phi = \{\varphi_1, \varphi_2, \varphi_3\}$, where $\varphi_1 : \textbf{Prev}(\texttt{PE} = re) \to (\texttt{PE} = dr) \succ (\texttt{PE} = cp)[\texttt{PC}]$; $\varphi_2 : \to (\texttt{PE} = cp) \succ (\texttt{PE} = ncp)$; $\varphi_3 : \textbf{AllPrev}(\texttt{PC} = mf) \to (\texttt{PC} = mf) \succ (\texttt{PC} = di)$.

Given a tcp-rule φ and two sequences s, s'. We say that s is preferred to s' (or s dominates s') according to φ, denoted by $s \succ_\varphi s'$ if and only if there exists a position i such that: **(1)** $s[j] = s'[j]$ for all $j \in \{1,...,i-1\}$; **(2)** $(s,i) \models C_\varphi$ and $(s',i) \models C_\varphi$; **(3)** $s[i].A_\varphi \models Q^+_\varphi$ and $s'[i].A_\varphi \models Q^-_\varphi$; **(4)** $s[i].A' = s'[i].A'$ for all $A' \notin (\{A_\varphi\} \cup W_\varphi)$ (*ceteris paribus* semantic).

The notation $\succ_\Phi$ represents the transitive closure of $\bigcup_{\varphi \in \Phi} \succ_\varphi$. The notation $s \succ_\Phi s'$ means that s is preferred to s' according to Φ. When two sequences cannot be compared, we say that they are incomparable. We also must consider consistency issues when dealing with order induced by rules to avoid inferences like “a sequence is preferred to itself”. So, we check the consistency of tcp-theories using the test proposed in [14] before the query execution.

4 Proposed Operators

Our StreamPref language introduces the operators **SEQ** and **BESTSEQ**. The **SEQ** operator extracts sequences from data streams preserving the temporal order of tuples and the **BESTSEQ** operator selects the best extracted sequences according to the defined temporal preferences. If it is necessary, our operators can be combined with the existing CQL operators to create more sophisticated queries. As we will see in the next section, our operators can be processed by equivalent CQL operations. However, the definition of these equivalences is not trivial and our operators have better performance than their CQL equivalent operations.

*The **SEQ** Operator.* The **SEQ** operator retrieves identified sequences (Definition 4) over a data stream according to: a set of identifier attributes (X), a temporal range (n) and a slide interval (d). The parameters n and d are used to select a portion of tuples from a data stream analogous to the selection performed by the *sliding window* approach [3, 13]. The parameter X is used to group the tuples with the same identifier in a sequence. It is important to note that the values for the identifier attributes must be unique at every instant to keep a relation one-to-one between tuples and sequences.

Definition 4 (Identified Sequences). *Let $S(A_1, ..., A_l)$ be a stream. Let Y and X be two disjoint sets such that $X \cup Y = \{A_1, ..., A_l\}$. An identified sequence $s_x = \langle t_1, ..., t_n \rangle$ from S is a sequence where $t_i \in \boldsymbol{Tup}(Y)$ for all $i \in \{1, ..., n\}$ and $x \in \boldsymbol{Tup}(X)$.*

TS	PID	PC	PE
1	1	mf	re
2	1	oi	dr
3	1	oi	cp

TS	PID	PC	PE
3	2	mf	re
4	2	oi	cp
5	2	oi	lb

TS	PID	PC	PE
6	3	mf	ca
7	3	mf	dr
8	3	di	ncp

TS	PID	PC	PE
8	4	mf	ca
9	4	mf	dr
10	4	mf	cp

Fig. 2. `Event` stream

Example 2. Consider the `Event` stream of Fig. 2 where `TS` is the timestamp (instant). The sequence extraction needed by query **Q1** presented in the motivating example is performed by operation $\mathbf{SEQ}_{\{\texttt{PID}\},6,1}$ (`Event`) as follows: `TS` **1:** $s_1 = \langle (mf, re) \rangle$; `TS` **2:** $s_1 = \langle (mf, re), (oi, dr) \rangle$, `TS` **3:** $s_1 = \langle (mf, re), (oi, dr), (oi, cp) \rangle$, $s_2 = \langle (mf, re) \rangle$; `TS` **4:** $s_1 = \langle (mf, re), (oi, dr), (oi, cp) \rangle$, $s_2 = \langle (mf, re), (oi, cp) \rangle$; `TS` **5:** $s_1 = \langle (mf, re), (oi, dr), (oi, cp) \rangle$, $s_2 = \langle (mf, re), (oi, cp), (oi, lb) \rangle$; `TS` **6:** $s_1 = \langle (mf, re), (oi, dr), (oi, cp) \rangle$, $s_2 = \langle (mf, re), (oi, cp), (oi, lb) \rangle$, $s_3 = \langle (mf, ca) \rangle$; `TS` **7:** $s_1 = \langle (oi, dr), (oi, cp) \rangle$, $s_3 = \langle (mf, ca), (mf, dr) \rangle$; `TS` **8:** $s_1 = \langle (oi, cp) \rangle$, $s_2 = \langle (mf, re), (oi, cp), (oi, lb) \rangle$, $s_3 = \langle (mf, ca), (mf, dr), (di, ncp) \rangle$, $s_4 = \langle (mf, ca) \rangle$; `TS` **9:** $s_2 = \langle (oi, cp), (oi, lb) \rangle$, $s_3 = \langle (mf, ca), (mf, dr), (di, ncp) \rangle$, $s_4 = \langle (mf, ca), (mf, dr) \rangle$; `TS` **10:** $s_2 = \langle (oi, lb) \rangle$, $s_3 = \langle (mf, ca), (mf, dr), (di, ncp) \rangle$, $s_4 = \langle (mf, ca), (mf, dr), (mf, cp) \rangle$. Note that from `TS` 7 the **SEQ** operator appends the new tuples and drops the expired positions in the beginning of the sequences.

*The **BESTSEQ** Operator.* Let Z be a set of sequences and Φ be a tcp-theory. The operation $\mathbf{BESTSEQ}_{\Phi}(Z)$ returns the *dominant* sequences in Z according to Φ. A sequence $s \in Z$ is dominant according to Φ, if $\nexists s' \in Z$ such that $s' \succ_{\Phi} s$.

Example 3. Let Z be the extracted sequences of Example 2 and Φ be the tcp-theory of Example 1. The query **Q1** is computed by the operation $\mathbf{BESTSEQ}_{\Phi}(\mathbf{SEQ}_{\{\texttt{PID}\},6,1}(\texttt{Event}))$ as follows: `TS` **1:** $\{s_1\}$ (the unique input sequence); `TS` **2:** $\{s_1\}$ (same result of `TS` 1); `TS` **3:** $\{s_1, s_2\}$ (incomparable sequences); `TS` **4:** $\{s_1\}$ ($s_1 \succ_{\varphi_1} s_2$); `TS` **5:** $\{s_1\}$ (same result of `TS` 4); `TS` **6:** $\{s_1, s_3\}$ ($s_1 \succ_{\varphi_1} s_2$ and s_3 is incomparable); `TS` **7:** $\{s_1, s_2, s_3\}$ (incomparable sequences); `TS` **8:** $\{s_1, s_2, s_3, s_4\}$ (incomparable sequences); `TS` **9:** $\{s_2, s_3, s_4\}$ (incomparable sequences); `TS` **10:** $\{s_2, s_4\}$ ($s_4 \succ_{\varphi_2} ... \succ_{\varphi_3} s_3$ and s_2 is incomparable).

5 CQL Equivalences

This section demonstrates how our StreamPref operators can be translated to CQL equivalent operations. It is worth noting that although this means the StreamPref does not increase the expression power of CQL, the equivalences are not trivial. Moreover, the evaluation of the StreamPref operators are more efficient than their CQL equivalent operations (see Section 7). The equivalences consider a stream $S(A_1, ..., A_l)$ and the identifier $X = \{A_1\}$. Although, we can use any subset of $\{A_1, ..., A_l\}$ as identifier without lost of generality. In addition, the CQL equivalences represent the sequences using relations containing the attribute `POS` to identify the position of the tuples.

We use the symbols π, $-$, $\bowtie$, γ, σ and $\cup$ for the CQL operators that are equivalent to the traditional operations: projection, set difference, join, aggregation function, selection and union, respectively. The **RSTREAM** is a CQL operator to convert a relation to a stream and the symbol $\boxplus$ is the CQL sliding window operator. The notation `TS`() returns the original timestamp of the tuple. We rename an attribute A to A' by using the notation $A \mapsto A'$ in the projection operator.

*Equivalence for the **SEQ** Operator.* Equation (1) establishes the CQL equivalence for the **SEQ** operator such that $P_0 = \{\}$ and $i \in \{1, ..., n\}$.

$$W_0 = \pi_{\texttt{POS},A_1}(\boxplus_{n,d}(\mathbf{RSTREAM}(\pi_{\texttt{TS}()\mapsto\texttt{POS},A_1,...,A_l}(\boxplus_{1,1}(S))))) \quad (1a)$$
$$W_i = W_{i-1} - P_{i-1} \quad (1b)$$
$$P_i = \gamma_{A_1,\min(\texttt{POS})\mapsto\texttt{POS}}(W_i) \bowtie_{A_1,\texttt{POS}} W \quad (1c)$$
$$\mathbf{SEQ}_{\{A_1\},n,d}(S) = \pi_{1\mapsto\texttt{POS},A_1,...A_l}(P_1) \cup ... \cup \pi_{n\mapsto\texttt{POS},A_1,...A_l}(P_n) \quad (1d)$$

*Equivalence for the **BESTSEQ** Operator.* The CQL equivalence for the **BESTSEQ** operator is computed over a relation $Z(\texttt{POS}, A_1, ..., A_l)$ containing the input sequences where $\{A_1\}$ is the identifier and `POS` is the position attribute. First, Equation (2) calculates the position to compare every pair of sequences.

$$Z' = \pi_{\texttt{POS},A_1\mapsto B',A_2\mapsto A_2',...,A_l\mapsto A_l'}(Z) \quad (2a)$$
$$P_{nc} = \sigma_{A_2\neq A_2'\vee...\vee A_l\neq A_l'}(\pi_{\texttt{POS},A_1\mapsto B,A_2,...,A_l}(Z) \bowtie_{\texttt{POS}} Z') \quad (2b)$$
$$P = \gamma_{B,B',\min(\texttt{POS})\mapsto\texttt{POS}}(P_{nc}) \quad (2c)$$

The next step is to identify the sequence positions satisfying the temporal components of the rule conditions. Equation (3) calculates the positions satisfied by every derived formula.

$$P^{\mathbf{First}} = \pi_{\mathtt{POS},B}(\sigma_{\mathtt{POS}=1}(P)) \tag{3a}$$
$$P^{\mathbf{Prev}}_{Q(A)} = \pi_{\mathtt{POS},B}(P \bowtie_{\mathtt{POS},B} (\pi_{(\mathtt{POS}+1)\mapsto\mathtt{POS},A_1\mapsto B}(\sigma_{Q(A)}(Z)))) \tag{3b}$$
$$P^{\mathbf{SomePrev}}_{Q(A)} = \pi_{\mathtt{POS},B}(\sigma_{\mathtt{POS}>\mathtt{POS'}}(P \bowtie_{B} (\gamma_{A_1\mapsto B,\min(\mathtt{POS})\mapsto\mathtt{POS'}}(\sigma_{Q(A)}(Z))))) \tag{3c}$$
$$P^{\max} = \gamma_{A_1\mapsto B,\max(\mathtt{POS})\mapsto\mathtt{POS}}(P) \tag{3d}$$
$$P'_{\neg Q(A)} = \gamma_{B,\min(\mathtt{POS})\mapsto\mathtt{POS'}}(\pi_{\mathtt{POS},A_1\mapsto B}(\sigma_{\neg Q(A)}(Z)) \cup P^{\max}) \tag{3e}$$
$$P^{\mathbf{AllPrev}}_{Q(A)} = \pi_{\mathtt{POS},B}(\sigma_{\mathtt{POS}\leq\mathtt{POS'}\wedge\mathtt{POS}>1}(P \bowtie_{B} (P'_{\neg Q(A)}))) \tag{3f}$$

Next, Equation (4) computes the relation R_i containing the positions satisfied by condition $C^{\leftarrow}_{\varphi_i} = F_1 \wedge ... \wedge F_p$ for every tcp-rule $\varphi_i \in \Phi$.

$$P_j = \begin{cases} P^{\mathbf{First}}, \mathbf{if}\ F_j = \mathbf{First} \\ P^{\mathbf{Prev}}_{Q(A)}, \mathbf{if}\ F_j = \mathbf{Prev}(Q(A)) \\ P^{\mathbf{SomePrev}}_{Q(A)}, \mathbf{if}\ F_j = \mathbf{SomePrev}(Q(A)) \\ P^{\mathbf{AllPrev}}_{Q(A)}, \mathbf{if}\ F_j = \mathbf{AllPrev}(Q(A)) \end{cases} \tag{4a}$$
$$R_i = (P_1) \bowtie_{\mathtt{POS},B} ... \bowtie_{\mathtt{POS},B} (P_p) \tag{4b}$$

Equation (5) performs the direct comparisons for every $\varphi_i \in \Phi$. This equation also consider the tuples of $\mathbf{Tup}(S)$ for posterior computation of the transitive closure. The relations D^+_i and D^-_i represent the tuples satisfying respectively the preferred values and the non-preferred values of φ_i. These tuples include *original tuples* (from existing positions) and *fake tuples* from $\mathbf{Tup}(S)$. We use the attribute A_t to separate these tuples ($A_t = 1$ for original tuples and $A_t = 0$ for fake tuples). Note that the Equation (5e) applies the filter $E_{\varphi_i} : (A_{i_1} = A'_{i_1}) \wedge ... \wedge (A_{i_j} = A'_{i_j})$ such that $\{A_{i_1}, ..., A_{i_j}\} = (\{A_1, ..., A_l\} - \{A_{\varphi_i}, B, B'\} - W_{\varphi_i})$. This filter is required to follow the *ceteris paribus* semantic.

$$Z^+_i = \pi_{\mathtt{POS},B}(R_i) \bowtie_{\mathtt{POS},B} (\sigma_{C^{\bullet}_{\varphi_i} \wedge Q^+_{\varphi_i}}(\pi_{\mathtt{POS},A_1\mapsto B,...,A_l,1\mapsto A_t}(Z))) \tag{5a}$$
$$D^+_i = Z^+_i \cup (\pi_{\mathtt{POS},B}(R_i) \bowtie_{B} (\pi_{A_1\mapsto B,...,A_l,0\mapsto A_t}(\sigma_{C^{\bullet}_{\varphi_i} \wedge Q^+_{\varphi_i}}(\mathbf{Tup}(S))))) \tag{5b}$$
$$Z^-_i = \pi_{\mathtt{POS},B'}(R_i) \bowtie_{\mathtt{POS},B'} (\sigma_{C^{\bullet}_{\varphi_i} \wedge Q^-_{\varphi_i}}(\pi_{\mathtt{POS},A_1\mapsto B',...,A_l,1\mapsto A_t}(Z))) \tag{5c}$$
$$D^-_i = Z^-_i \cup (\pi_{\mathtt{POS},B'}(R_i) \bowtie_{B'} (\pi_{A_1\mapsto B',...,A_l,0\mapsto A_t}(\sigma_{C^{\bullet}_{\varphi_i} \wedge Q^-_{\varphi_i}}(\mathbf{Tup}(S))))) \tag{5d}$$
$$D_i = \sigma_{E_{\varphi_i}}(P \bowtie_{\mathtt{POS},B,B'} (D^+_i \bowtie_{\mathtt{POS}} (\pi_{\mathtt{POS},B',A_2\mapsto A'_2,...,A_l\mapsto A'_l,A_t\mapsto A'_t}(D^-_i)))) \tag{5e}$$

Equation (6) calculates the transitive closure. The relations T'_i, T''_i and T_i are computed for $i \in \{2, ..., m\}$. In the end, T_m has all comparisons imposed by Φ where $m = |\Phi|$ is the number of tcp-rules.

$$T_1 = D_1 \cup ... \cup D_m \tag{6a}$$
$$T'_i = \pi_{(\mathtt{POS},B,B',A_2,...,A_l,A_t,A'_2\mapsto A''_2,...,A'_l\mapsto A''_l,A'_t\mapsto A''_t)}(T_{i-1}) \tag{6b}$$
$$T''_i = \pi_{(\mathtt{POS},B,B',A_2\mapsto A''_2,...,A_l\mapsto A''_l,A_t\mapsto A''_t,A'_2,...,A'_l,A'_t)}(T_{i-1}) \tag{6c}$$
$$T_i = \pi_{\mathtt{POS},B,B',A_2,...,A_l,A_t,A'_2,...,A'_l,A'_t}(T'_i \bowtie_{\mathtt{POS},B,B',A''_2,...,A''_l} T''_i) \cup T_{i-1} \tag{6d}$$

Equation (7) calculates the dominant sequences. Observe that just comparisons between original tuples are considered ($A_t = 1 \wedge A'_t = 1$).

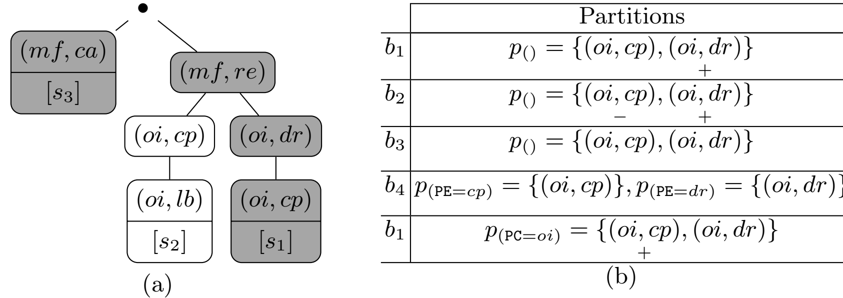
$$\mathbf{BESTSEQ}_{\Phi}(Z) = Z \ltimes_{A_1} (\pi'_{A_1}(Z) - \pi'_{B' \mapsto A_1}(\sigma_{A_t=1 \wedge A'_t=1}(T_m))) \qquad (7)$$

6 Data Structures and Algorithms

Our previous work [14] proposed the algorithm *ExtractSeq* for extracting sequences and the algorithm *BestSeq* for computing the dominant sequences. The StreamPref operators **SEQ** and **BESTSEQ** can be evaluated by the algorithms *ExtractSeq* and *BestSeq*, respectively. However, only the algorithm *ExtractSeq* uses an incremental method suitable for data streams scenarios. In this paper we propose a new incremental method to evaluate the **BESTSEQ** operator.

Index Structure. The main idea of our incremental method is to keep an index tree built using the sequence tuples. Given a sequence $s = \langle t_1, ..., t_n \rangle$, every tuple t_i is represented by a node in the tree. The tuple t_1 is a child of the root node. For the remaining tuples, every t_i is a father of t_{i+1}. The sequence s is stored in the node t_n.

Example 4. Consider the sequences of the Example 2 at `TS` 6. Fig. 3(a) shows how these sequences are stored in the index tree. The root node, represented by black circle, is an empty node without an associated tuple.



	Partitions
b_1	$p_{()} = \{(oi, cp), (oi, dr)\}$ (+ under (oi, dr))
b_2	$p_{()} = \{(oi, cp), (oi, dr)\}$ (− under (oi, cp), + under (oi, dr))
b_3	$p_{()} = \{(oi, cp), (oi, dr)\}$
b_4	$p_{(\texttt{PE}=cp)} = \{(oi, cp)\}, p_{(\texttt{PE}=dr)} = \{(oi, dr)\}$
b_1	$p_{(\texttt{PC}=oi)} = \{(oi, cp), (oi, dr)\}$ (+ under (oi, cp))

Fig. 3. Preference hierarchy: (a) Index tree; (b) Partitions imposed by K_Γ of $(oi, 1, la)$.

Starting from the root node, it is possible to find the position where two sequences must be compared. For instance, consider the sequences s_1 and s_2 in the tree of Fig. 3(a). The paths from the root to these sequences are different in the second node. Thus, the comparison of s_1 and s_2 happens in the position 2.

The index is updated only for changed sequences and new sequences. The new sequences are just inserted in the tree. When positions are deleted from a sequence s (and s is still no empty), we reinsert s in the tree. The empty sequences are dropped from the tree. If a sequence s has new tuples (and no expired tuples), we move s to a child branch of its current node.

Given a node nd, $nd.t$ is the tuple associated to nd and $nd.Z$ represents the set of sequences stored in nd. The children of nd are stored in a hash-table $nd.Ch$ mapping the associated tuples to the respective child nodes. In addition, each

node nd stores a preference hierarchy $nd.H$ over the tuples of the child nodes. The preference hierarchy allows to determine if a child node is dominant or is dominated. Thus, it is possible to know if a sequence dominates another one.

Preference Hierarchy. Our preference hierarchy structure is based on the preference partition technique originally proposed in our previous work [15]. The main idea is to build a knowledge base for the preferences valid in a node and keep a structure containing the preferred and non-preferred tuples according to such preferences. Given a set of non-temporal preference rules Γ. The knowledge base K_Γ over Γ is a set of comparisons in the format $b : F_b^+ \succ F_b^- [W_b]$. The terms F_b^+ and F_b^- are formulas representing the preferred values and non-preferred values, respectively. The term W_b is the set of indifferent attributes of b. For more details about the construction of the knowledge base, please see [15].

After the construction of K_Γ, the preference hierarchy is built by grouping the child node tuples into subsets called partitions. For every comparison $b \in K_\Gamma$, we group the tuples into partitions according to the attribute values not in W_b. If a partition does not contain *preferred tuples* (those satisfying F_b^+), then all tuples of this partition are dominant. On the other hand, if a partition has at least one preferred tuple, then all *non-preferred tuples* (those satisfying F_b^-) are dominated. Therefore, a tuple t is dominant if t is not dominated in any partition.

Example 5. Consider again the tcp-theory Φ of the Example 2 and the tree of the Fig. 3(a). The preference hierarchy of the node (mf, re) is built using the non-temporal components of the rules temporally valid in the last position of the sequence $s = \langle (mf, re), t \rangle$, where t is any tuple (t is not used to temporally validate the rules). For this node, all rules are used. So, we have the knowledge base K_Γ over $\Gamma = \{\varphi_1^\bullet, \varphi_2^\bullet, \varphi_3^\bullet\}$ containing the comparisons $b_1 : (\texttt{PE} = dr) \succ (\texttt{PE} = ncp)$ [PC, PE]; $b_2 : (\texttt{PE} = dr) \succ (\texttt{PE} = cp)$ [PC, PE]; $b_3 : (\texttt{PC} = mf) \wedge (\texttt{PE} = dr) \succ (\texttt{PC} = di) \wedge (\texttt{PE} = ncp)$ [PC, PE]; $b_4 : (\texttt{PC} = mf) \succ (\texttt{PC} = di)$ [PC]; $b_5 : (\texttt{PE} = cp) \succ (\texttt{PE} = ncp)$ [PE]. Fig. 3(b) shows the partitions imposed by K_Γ. The symbols $+$ and $-$ indicate if the tuple is preferred or non-preferred, respectively. We can see that (oi, cp) is dominated because it is a non-preferred tuple in the partition of the comparison b_2 containing a preferred tuple.

Our technique uses just the essential information to update the index efficiently. For every partition p, we keep the mappings *Pref*(p) and *NonPref*(p) representing the number of preferred tuples in p and the set on non-preferred tuples in p, respectively. Thus, a node nd is dominated if there exists p such that *Pref*(p) > 0 and $nd.t \in$ *NonPref*(p).

The construction of the knowledge base is not a trivial task since we must compute comparisons representing the transitive closure imposed by the preferences [15]. Thus, we also use a pruning strategy to avoid the construction of unnecessary preference hierarchies. Nodes having a unique child, do not need preference hierarchy since this child is always dominant. In addition, dominated nodes and their descendants do not require preference hierarchy. For example, in the tree of Fig. 3(a), we know that (oi, dr) dominates (oi, cp) according to the

hierarchy of (mf, re). Thus, we do not need preference hierarchies for nodes in the branch starting at (oi, cp).

Algorithms. Our full index structure is composed by the tree nodes represented by its *root* and the mapping *SeqNod*. For every identified sequence s_x, $SeqNod(x)$ stores (s_x, nd) where nd is the node where s_x is stored. In addition, we use the sequence attributes *deleted* and *inserted* to keep the number of deletions and insertions in the last instant.

The algorithm *IndexUpdate* (see Algorithm 1) incrementally updates the index according to sequence changes. The first loop (lines 2-11) processes the changes for every sequence s_x already stored. If s_x has expired positions ($s_x.deleted > 0$), we remove s_x from the index. When s_x is not empty ($|s_x| > 0$), we add s_x into I to be reinserted later since s_x must be repositioned in the index tree. If s_x has no expired positions and has inserted positions, the routine *AddSeq* reallocates s_x from its current node. The second loop (lines 12-13) looks for sequences in Z not stored in the index and adds them into I. At the end, the algorithm stores the sequences of I and calls the routine *Clean* to remove empty nodes.

Algorithm 1: *IndexUpdate*(idx, Z)

```
1  I ← {};
2  foreach x ∈ idx.SeqNod do
3      (s_x, nd) ← idx.SeqNod(x);
4      if s_x.deleted > 0 then
5          nd.Z.Del(s_x);
6          idx.SeqNod.Del(x);
7          if |s_x| > 0 then I.add(s_x);
8      else if s_x.inserted > 0 then
9          nd.Z.Del(s_x);
10         new ← AddSeq(nd, s_x);
11         idx.SeqNod.Put(x ↦ (s_x, new));
12 foreach s_x ∈ Z do
13     if x ∉ idx.SeqNod then I.add(s_x);
14 foreach s_x ∈ I do
15     nd ← AddSeq(idx.root, s_x);
16     idx.SeqNod.Put(x ↦ (s_x, nd));
17 Clean(idx.root);
```

Algorithm 2: *AddSeq*(nd, s_x)

```
1  d ← Depth(nd);
2  if d = |s_x| then
3      nd.Z.Add(s_x);
4      return nd;
5  t ← s_x[d + 1];
6  if t ∈ nd.Ch then
7      child ← nd.Ch(t);
8  else
9      child ← NewChild(nd, t);
10 return AddSeq(child, s);
```

Algorithm 3: *IncBestSeq*(nd)

```
1  Z ← nd.Z;
2  foreach dominant child of nd do
3      Z ← Z ∪ IncBestSeq(child);
4  return Z;
```

The routine *AddSeq* (see Algorithm 2) performs the insertion of a sequence s_x in the index tree. First, the routine checks if the depth (d) of node nd is equal to the length of s_x ($d = |s_x|$). If true, s_x is stored into nd since the full path containing the tuples of s_x is already created. Otherwise, the routine selects an existing child or creates a new one. At the end, the routine makes a recursion over this *child* node.

The algorithm *IncBestSeq* (see Algorithm 3) employs the index tree to evaluate the **BESTSEQ** operator incrementally. The execution starts at *idx.root*.

The algorithm acquires the sequences of input node *nd* and uses the preference hierarchy to select the dominant children of *nd*. Thus, for every dominant child, the algorithm makes a recursive call to retrieve all dominant sequences.

Example 6. Consider again the index tree of Fig. 3(a) (the dominant nodes are in gray). The execution of *IncBestSeq* over this index tree works as follows:

(1) The execution starts at the root node. This node has no sequences and the algorithm makes a recursion over the dominant children (mf, ca) and (mf, re);

(2) At (mf, ca) the algorithm reaches the sequence s_3. So, $Z = \{s_3\}$;

(3) At (mf, re), the algorithm performs a recursion over (oi, dr);

(4) At (oi, dr), the algorithm starts a recursion over (oi, cp);

(5) The algorithm reaches the sequence s_1 and returns $Z = \{s_1, s_3\}$.

Complexity. The complexity analysis of the algorithms takes into account the number of input sequences (k), the length of the largest sequence (n) and the number of tcp-rules in Φ (m). We also assume a constant factor for the number of attributes. In the worst case, the insertion or the deletion of a node tuple in the preference hierarchy has the cost $O(m^4)$ (the size of K_Γ [15]). Moreover, in the worst case scenario, the degree of nodes is $O(k)$, the tree depth is $O(n)$ and the number of partitions associated to a child node is $O(|K_\Gamma|)$. Our mapping structures *SeqNod*, *Ch*, *Pref* and *NonPrefSet* are implemented using hash-tables. So, the retrieval and the storage of elements is performed with a cost of $O(1)$.

In the worst case, the routine *AddSeq* reaches a leaf node and the routine *Clean* scans all tree nodes. So, the costs of *AddSeq* and *Clean* are $O(nm^4)$ and $O(k^n m^4)$, respectively. The complexity of the algorithm *IndexUpdate* is $O(k^n m^4)$. Regarding the algorithm *IncBestSeq*, the selection of the dominant children has a cost of $O(km^4)$. Thus, the complexity of *IncBestSeq* is $O(k^n m^4)$.

7 Experimental Results

Our experiments confront our proposed operators against their CQL counterparts to analyze the performance (runtime) and memory usage of both approaches. All experiments were carried out on a machine with a 3.2 GHz twelve-core processor and 32 GB of main memory, running Linux. The algorithms and all CQL operators were implemented in Python language.

Synthetic Datasets. Due to the nonexistence of data generators suitable for the experiment parameters employed herein, we designed our own generator of synthetic datasets[4]. The synthetic datasets are in the format of streams composed by integer attributes. Table 1 shows the parameters (with default values in bold).

Table 1(a) presents the parameters related to the dataset generation. The number of attributes (ATT) allows to evaluate the behavior of the algorithms according to data dimensionality. The number of sequences (NSQ) allows to

[4] http://streampref.github.io

Table 1. Parameters for the experiments over synthetic data: (a) Dataset generation; (b) Sequence extraction; (c) Preferences.

(a)

Param.	Variation
ATT	8, **10**, 12, 14, 16
NSQ	4, 8, **16**, 24, 32

(b)

Param.	Variation
RAN	10, 20, **40**, 60, 80, 100
SLI	1, **10**, 20, 30, 40

(c)

Param.	Variation
RUL	4, **8**, 16, 24, 32
LEV	1, **2**, 3, 4, 5, 6

evaluate how the number of tuples per instant (equal to NSQ $\times$ 0.5) affects the algorithms. Table 1(b) displays the parameters used for sequence extraction. These parameters are temporal range (RAN) and slide interval (SLI) and they allow to evaluate how the selection of the stream elements influences the algorithms. Table 1(c) shows the parameters number of rules (RUL) and maximum preference level (LEV) employed for the generation of the preferences. These parameters allow us to evaluate how different preferences affect the cost of the sequence comparison done by the algorithms. We use rules in the form φ_i : $\textbf{First} \wedge Q(A_3) \rightarrow Q^+(A_2) \succ Q^-(A_2)[A_4, A_5]$ and φ_{i+1} : $\textbf{Prev}Q(A_3) \wedge \textbf{SomePrev}Q(A_4) \wedge \textbf{AllPrev}Q(A_5) \wedge Q(A_3) \rightarrow Q^+(A_2) \succ Q^-(A_2)[A_4, A_5]$ having variations on propositions $Q^+(A_2)$, $Q^-(A_2)$, $Q(A_3)$, $Q(A_4)$, $Q(A_5)$. The number of iterations is RAN plus maximum slide interval and the sequence identifier is the attribute A_1. The definition of the parameter values was based on the experiments of related works [14, 15, 8, 11, 13]. For each experiment, we varied one parameter and fixed the default value for the others.

Real Datasets. We also used a real dataset containing play-by-play data of the 2014 soccer world cup[5]. This dataset contains 10,282 tuples from the last 4 matches. For this dataset we varied the parameters RAN and SLI. The values for RAN were 6, 12, 18, 24 and 30 seconds, where the default was 12 seconds. The values for SLI were 1, 3, 6, 9 and 12 seconds, where the default was 1 second. The experiments consider the average runtime per match which is the total runtime of all matches divided by the number of matches.

*Experiments with the **SEQ** Operator.* Fig. 4(a) and Fig. 4(b) show the experiment results with synthetic data confronting the **SEQ** operator (evaluated by the algorithm *ExtractSeq*) and its CQL equivalence. The first experiment considers the variation on the parameter ATT. Even for few attributes, the **SEQ** operator outperforms the CQL equivalence. Regarding the NSQ parameter, the **SEQ** operator has the best performance again. When there are more sequences, there are more tuples to be processed and the CQL operations are more expensive.

When examining the results obtained with different temporal ranges, it is possible to see that higher temporal ranges had greater impact on the CQL equivalence due to the generation of bigger sequences. Considering the results obtained for the SLI parameter, bigger slides caused more tuples expiration. So, once the sequences are smaller, the **SEQ** operator had the best performance. Finally, considering all experiments shown in Fig. 4(a), it is possible to see that the **SEQ** operator is more efficient than its CQL equivalence. Moreover, when comparing the memory usage displayed on Fig. 4(b), it is possible to verify that

[5] Extracted from data available in http://data.huffingtonpost.com/2014/world-cup

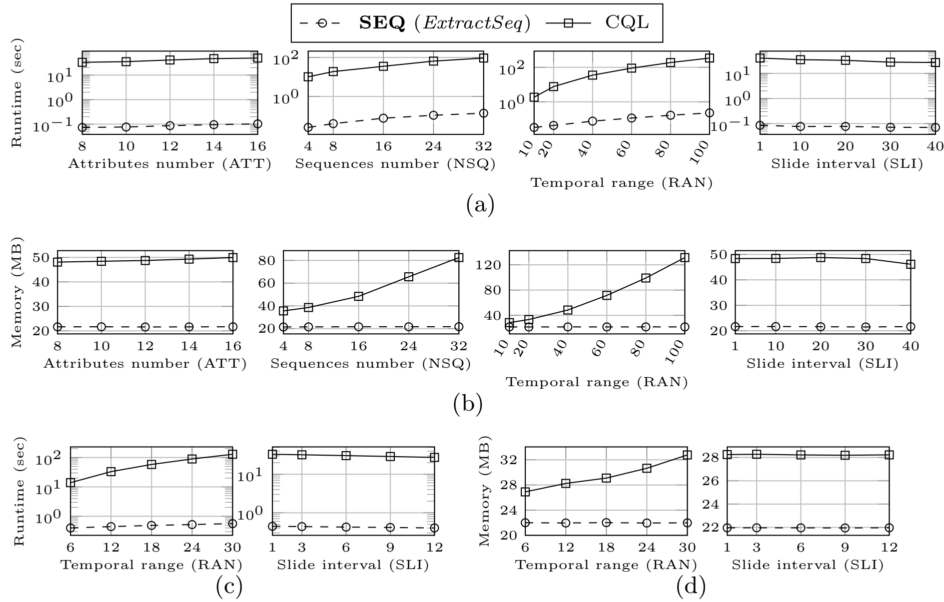


Fig. 4. Experiment results for the **SEQ** operator: (a) Synthetic data runtime; (b) Synthetic data memory usage; (c) Real data runtime; (d) Real data memory usage.

the CQL equivalence for the **SEQ** operator had a high memory usage in all experiments due to the extra tuples stored by the intermediary operations.

Fig. 4(d) and Fig. 4(c) present the results obtained with the real data. These results are analogous to the ones obtained with the synthetic data. Analyzing these figures, it is possible to see that the **SEQ** operator outperforms the CQL equivalence for all the experiments.

*Experiments with the **BESTSEQ** Operator.* Fig. 5(a) shows the runtime and Fig. 5(b) shows the memory usage for the experiments with the **BESTSEQ** operator with synthetic data. Notice that the runtime graphs are in logarithm scale. We can see that the CQL equivalence is slower than the remaining algorithms due to the processing of the intermediary operations. In addition, the incremental algorithm outperforms the naive algorithm due to the index tree and the pruning strategy. The same behavior is observed for the memory usage.

The results obtained for the parameters NSQ, RAN and RUL deserve to be highlighted. Considering the NSQ parameter, the behavior of the **BESTSEQ** algorithms is explained by the fact that when the number of sequences increases, we have more repetition of sequence identifiers and more chances for pruning. So, the updates in the index tree affect fewer branches and the incremental algorithm outperforms the naive algorithm.

Considering the RAN parameter, the incremental algorithm presents an advantage over the naive algorithm since longer sequences have more chances for overlapping. This behavior results in a more compact index tree and in a better performance for the incremental algorithm.

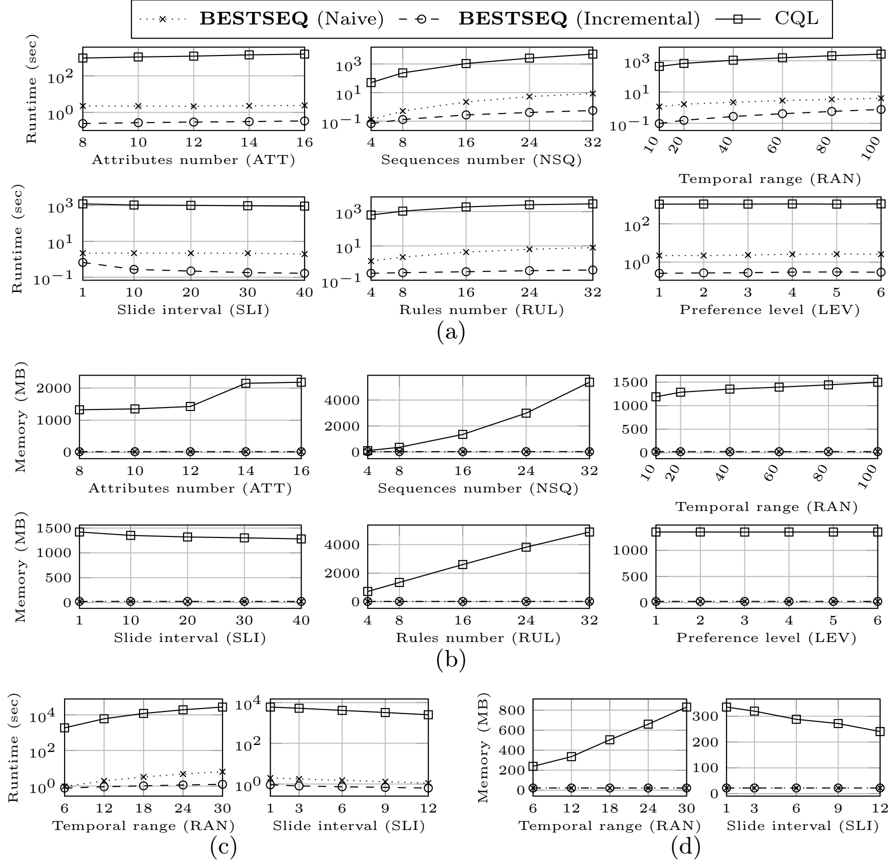


Fig. 5. Experiment results of **BESTSEQ** operator: (a) Synthetic data runtime; (b) Synthetic data memory usage; (c) Real data runtime. (d) Real data memory usage.

Regarding the results obtained by the naive algorithm considering the RUL parameter, it is worth noting that the number of rules has a great impact in its complexity [14]. Moreover, more rules means more intermediary relations in the CQL equivalence as addressed by Equations (4) and (5).

Fig. 5(c) and Fig. 5(d) show the results obtained for the **BESTSEQ** operator with the real data. Analyzing this figure, it is possible to see that the naive and the incremental algorithms outperform the CQL equivalence again. In addition, the CQL runtime cannot be applied in a real situation since a regular soccer match has duration of 5400 seconds. Regarding the memory usage, as expected, the CQL equivalence presented the greatest memory usage due to the storage of the intermediary relations. Both versions of **BESTSEQ** have a stable memory usage in all executions (around 20 MB).

8 Conclusion

In this paper we described the StreamPref query language presenting new operators to support temporal conditional preference queries on data streams. Stream-

Pref extends the CQL language including the **SEQ** operator for the sequence extraction and the **BESTSEQ** preference operator for the selection of dominant sequences. Regarding the evaluation of this later operator, a new incremental algorithm was proposed. In addition, we also demonstrated the CQL equivalences for the proposed operators. It is worth noting that these equivalences are not trivial since they involve many complex operations. In our experiments, we compared the previous algorithms proposed in [14], the new incremental algorithm and their CQL equivalences. The experimental results showed that our proposed operators outperform the equivalent operations in CQL. Furthermore, our incremental algorithm achieved the best performance for evaluating the **BESTSEQ** operator.

Acknowledgments. The authors thanks the Research Agencies CNPq, CAPES and FAPEMIG for supporting this work.

References

1. de Amo, S., Bueno, M.L.P.: Continuous processing of conditional preference queries. In: SBBD. Florianópolis, Brasil (2011)
2. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: STREAM: The Stanford Data Stream Management System, pp. 317–336. Springer, Berlin, Heidelberg (2016)
3. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. The VLDB Journal 15(2), 121–142 (2006)
4. Chomicki, J., Ciaccia, P., Meneghetti, N.: Skyline queries, front and back. ACM SIGMOD Record 42(3), 6–18 (2013)
5. Golab, L., Özsu, M.T.: Issues in data stream management. ACM SIGMOD Record 32(2), 5–14 (2003)
6. Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R.: A catalog of stream processing optimizations. ACM Computing Surveys 46(4), 46:1–46:34 (Mar 2014)
7. Kontaki, M., Papadopoulos, A.N., Manolopoulos, Y.: Continuous top-k dominating queries. IEEE Trans. on Knowledge and Data Eng. (TKDE) 24(5), 840–853 (2012)
8. Lee, Y.W., Lee, K.Y., Kim, M.H.: Efficient processing of multiple continuous skyline queries over a data stream. Information Sciences 221, 316–337 (2013)
9. Liu, W., Shen, Y.M., Wang, P.: An efficient approach of processing multiple continuous queries. J. of Computer Science and Technology 31(6), 1212–1227 (2016)
10. Margara, A., Urbani, J., van Harmelen, F., Bal, H.: Streaming the web: Reasoning over dynamic data. Web Semantics: Science, Services and Agents on the World Wide Web 25, 24–44 (2014)
11. Pereira, F.S.F., de Amo, S.: Evaluation of conditional preference queries. JIDM 1(3), 503–518 (2010)
12. Petit, L., de Amo, S., Roncancio, C., Labbé, C.: Top-k context-aware queries on streams. In: DEXA. pp. 397–411. Vienna, Austria (2012)
13. Petit, L., Labbé, C., Roncancio, C.: An algebric window model for data stream management. In: ACM MobiDE. pp. 17–24. Indianapolis, Indiana, USA (2010)
14. Ribeiro, M.R., Barioni, M.C.N., de Amo, S., Roncancio, C., Labbé, C.: Reasoning with temporal preferences over data streams. In: FLAIRS. Marco Island, USA (2017)
15. Ribeiro, M.R., Pereira, F.S.F., Dias, V.V.S.: Efficient algorithms for processing preference queries. In: ACM SAC. pp. 972–979. Pisa, Italy (2016)