

# Efficient Constraint Evaluation in Categorical Sequential Pattern Mining for Trajectory Databases

Leticia I. Gomez

Instituto Tecnológico de Buenos Aires  
lgomez@itba.edu.ar

Alejandro A. Vaisman

Universidad de Buenos Aires and  
University of Hasselt and  
Transnational University of Limburg, Belgium  
alejandro.vaisman@uhasselt.be

## ABSTRACT

The classic Generalized Sequential Patterns (GSP) algorithm returns all frequent sequences present in a database. However, usually a few ones are interesting from a user’s point of view. Thus, post-processing tasks are required in order to discard uninteresting sequences. To avoid this drawback, languages based on regular expressions (RE) were proposed to restrict frequent sequences to the ones that satisfy user-specified constraints. In all of these languages, REs are applied over items, which limits their applicability in complex real-world situations. We propose a much powerful language, based on regular expressions, denoted RE-SPaM, where the basic elements are constraints defined over the (temporal and non-temporal) attributes of the items to be mined. Expressions in this language may include attributes, functions over attributes, and variables. We specify the syntax and semantics of RE-SPaM, and present a comprehensive set of examples to illustrate its expressive power. We study in detail how the expressions can be used to prune the resulting sequences in the mining process. In addition, we introduce techniques that allow pruning sequences in the early stages of the process, reducing the need to access the database, making use of the categorization of the attributes that compose the items, and of the automaton that accepts the language generated by the RE. Finally, we present experimental results. Although in this paper we focus on trajectory databases, our approach is general enough for being applied to other settings.

## 1. INTRODUCTION

In many application domains, information is organized as ordered sequences. Usual examples are log analysis and retail market analysis. These applications can benefit from the discovery of hidden patterns in such sequences. The data mining community has contributed to solve these problems developing algorithms for efficiently discovering sequential patterns [3, 14, 10]. Some of these algorithms defined a pattern as *interesting* if this pattern is *frequent*, i.e., if it appears at least as many times as a user-specified threshold. The problem with this approach is that the user may obtain frequent sequential patterns not necessarily relevant enough from her point of view. Garofalakis *et al.* [6, 7] address this problem by pruning

the candidate patterns obtained during the mining process adding user-specified constraints in the form of regular expressions. The sequential pattern algorithm returns only the frequent patterns that satisfy these regular expressions.

In spite of the above, for many novel applications, more general forms of constraints are required. For example, when we need to discover patterns at different granularity levels involving some kind of aggregation, the task becomes more involved. This is the problem we study in this paper, where we introduce a language to express a wide range of constraints not supported by previous proposals, thus limiting their scope.

### 1.1 Motivation and Running Example

We motivate our work with an example from the field of moving object databases (MOD). Moving object data applications have been steadily gaining attention [8], in particular, the ones involving trajectory aggregation, mainly regarding traffic analysis. The behavior of moving objects is traceable by means of electronic devices (e.g., GPS, RFID). Since locations of a moving object are reported as a time-ordered sequence, sequential pattern algorithms appear as natural tools for querying and mining MODs. In what follows, we assume that MOD are captured at a given time interval, with a certain granularity. Thus, the trajectory of a moving object is given by samples composed of a finite number of tuples of the form  $\langle Oid, t, x, y \rangle$ , stating that at a certain point in time, namely  $t$ , the object  $Oid$  was located at coordinates  $(x, y)$ .

We address the problem of discovering patterns in a trajectory database, where each trajectory is represented as a time-ordered sequence of the geographic positions of an object. This trajectory database is transformed, using the ideas introduced by Spaccapietra *et al.* [13], and Alvares *et al.* [?], where it is assumed that objects move over a map that contains disjoint geometries, along with semantic information associated with them in the form of attributes. We denote these geometries *Places of Interest of the Application* (PoIs), because they depend on the application at hand, i.e., for the same map, two applications may have different PoIs defined for them. For example, in a tourist application PoIs can be restaurants, hotels and tourist attractions. When a moving object spends a sufficient amount of time within a PoI, the PoI is considered a *stop* of the trajectory, and all  $(x, y)$  points in the PoI are replaced by a data object representing the stop. This allows considering each object’s trajectory as a *sequence of stops* instead of a *sequence of points*. We apply sequential pattern algorithms to this approximation of trajectories (also called “semantic trajectories”).

Throughout the paper we refer to a tourist application in the city of Paris. The items to be mined are, precisely, sequences of stops, composed of the PoIs visited by tourists, the time interval spent for each moving object at each stop, and the attributes of the PoIs. The data model we present supports item categorization, meaning that

each item can be classified as belonging to a category, described by a set of attributes. Intuitively, this allows us to talk about *schema* (i.e., the structure), *occurrence* (a concrete instance of a category, e.g., a particular hotel), and *instances* (a set of occurrences) of categories. In our example we have four categories: hotels, restaurants, the Eiffel Tower and zoos, with different attributes and number of occurrences. Table 1 shows these categories and their schemas. An instance (i.e., a set of occurrences), is shown in Figure 1. Each time a tourist visits a PoI and stops at it, this information is recorded. More precisely, each stop of the moving object (i.e., an item) is a complex object composed of temporal attributes (indicating the duration of the stop), and category attributes. Each item can belong to exactly one category. We want to discover frequent sequential patterns for moving objects (although the approach could be used in any application domain), restricting these patterns to the ones that satisfy a set of constraints, specified by means of regular expressions over the attributes of the objects. These constraints are of the form “trajectories that first visit cheap restaurants, then go to a 3-star hotel, and finish at the first restaurant”.

## 1.2 Related Work

Two main approaches had been followed in the field of pattern discovery in sequences: (a) The classic Agrawal and Srikant [3] proposal, and (b) The approach of Mannila *et al.* [10]. The former is aimed at discovering inter-transactions patterns, based on previous work [1, 2] dealing with detecting intra-transactions patterns. The information to be mined is organized in transactions, and the system returns the frequent sequential patterns among them. The latter, instead, considers the information to be mined as a large single sequence. The choice of the algorithm depends on the application domain. Our work is based on (a) above, and its variants, i.e., we focus on the problem of mining sequential patterns in transactions. In that work, data is pre-processed in a way such that each customer (in a market basket analysis scenario) is associated with all her transactions ordered by time of occurrence. The idea is to find inter-transaction patterns corresponding to the same customer with a certain support. An interesting sequential pattern is one that appears in the database at least as many times as a user-specified threshold. The support of a sequence is the fraction of the total number of transactions containing it. In further work, the authors extended their proposal [14] in order to support three kinds of constraints: (a) time-gap constraints; (b) taxonomies; (c) time windows. The resulting algorithm is called Generalized Sequential Patterns (GSP). Although many frequent sequential patterns could be obtained using GSP, it is likely that only a few of them could be relevant to the user. To avoid this situation, Garofalakis *et al.* [6, 7] propose a variation of the GSP algorithm, denoted SPIRIT, where user-defined regular expressions are used to prune the information obtained. For example, assume that in a Web portal log, each user’s transaction is recorded, and that the following frequent sequential patterns are obtained:

$\{\{URL\_B\}, \{URL\_M\}\}; \{\{URL\_B\}, \{URL\_R\}\}$   
 $\{\{URL\_A\}, \{URL\_D\}, \{URL\_M\}\}, \{\{URL\_A\}, \{URL\_D\}, \{URL\_Z\}\}$   
 The last pattern shows that users navigate from  $URL\_A$  to  $URL\_Z$ , passing through many pages on the way. Based on this information, a Web designer may want to place a link between these two pages. The first two patterns appear to be useless from this point of view. Thus, the designer may want to specify the following constraint in order to retrieve only the patterns of interest for her:

$URL\_A(URL\_A|URL\_B|URL\_D|URL\_M|URL\_R)^+URL\_Z$

A-priori like algorithms are based on the anti-monotony property. Although they take advantage of pruning, they generate a generally huge number of candidate frequent patterns. To improve this,

pattern-growth methods have been proposed to avoid the generation of candidate sequences: FreeSpan [9] and PrefixSpan [11]. In these methods, projected databases are built recursively, and these smaller databases are scanned to find locally frequent sequences, avoiding scanning the original sequences database. These methods find the full set of frequent subsequences. Then, constraint-based sequential pattern mining based on constructing projected databases have been studied [12]. Further, to avoid generating patterns that could be obtained from other ones, Yan *et al.* introduce the CloSpan algorithm [15], which reduces the number of generated patterns by mining only frequent closed subsequences, i.e., those containing no super-sequence with the same support.

In the field of trajectory analysis, Mouza and Rigaux [5] proposed a language based on regular expressions for querying mobility patterns in trajectories where each zone could be represented by its label (a constant) or by a variable (@x). In this language, each occurrence of a variable in the pattern is instantiated with the same value. The units of time spent by the moving object inside some zone are expressed with the symbol + (undetermined time) or via the temporal constraint boundaries {min, max}. The query “objects that started in zone A, visited another zone and five minutes later came back to A”, is expressed in this language as:  $A,7.@X^+A,12$ . Here, variables *can only be associated with places* (represented by labels or IDs) visited by objects. Thus, the language cannot deal with *time constraints* or categories. On the contrary, our approach allows variables associated with any attribute of an item.

## 1.3 Contribution and organization

Regular expressions have been used for mining sequential patterns, particularly in the field of trajectory databases. Proposals like SPIRIT [6, 7] are aimed at pruning uninteresting sequences, using regular expressions. However, when mining huge volumes of data, and complex constraints must be managed, existing approaches do not suffice for an effective pruning phase. With this in mind, we extend existing work in several ways: we propose a language based on regular expressions, called RE-SPaM, built on constraints (i.e., conditions over attributes of complex items) rather than over atomic items. Further, regular expressions in RE-SPaM can contain constants, attributes, and variables in a way that substantially improves earlier proposals. We also allow *functions over attributes*. These functions can be defined, for instance, in a relational database, a multidimensional database (in the form of a rollout function [4]), or as a Web Service. We remark that, in the work of Srikant *et al.* [14], only items (IDs) are allowed to participate on hierarchies. We extend this idea allowing any kind of attribute (including temporal ones) in such hierarchies. We show that supporting attribute, variables, and categorization, implies not merely an addition to previous proposals, but introduces new theoretical and practical problems, providing a powerful language for sequential pattern mining, relevant to many real-world applications.

We first introduce the formal data model (Section 2). Then, we define the syntax and semantics of RE-SPaM, along with a comprehensive set of examples (Section 3). In Section 4 we present an algorithm for sequential pattern mining supporting these new features, also providing an in-depth discussion of different implementation issues. Section 5 presents and discusses experimental results. We conclude in Section 6.

## 2. PRELIMINARIES AND DATA MODEL

Traditional algorithms for sequential pattern mining work over *atomic items*, i.e., literals. Each item has the time interval of the transaction associated with it, used to define an order among the itemsets. In this work we consider items as composed of attributes.

Category	Schema
hotels	[ID, categoryName, geom, star]
restaurants	[ID, categoryName, geom, typeOfFood, price]
Eiffel Tower	[ID, categoryName, geom]
zoos	[ID, categoryName, geom, price]

**Table 1: Schema of the categories in the running example**

We first introduce a formal model and then define the regular language we use in the pruning phase of the data mining algorithm of Section 4. As usual in databases, we work with the notion of category *schema* and its associated *instances*. We have a set of attribute names  $\mathbf{A}$ , and a set of identifier names  $\mathbf{I}$ . Each attribute  $attr \in \mathbf{A}$  is associated with a set of values in  $dom(attr)$ , and each identifier  $ID \in \mathbf{I}$  is associated with a set of values in  $dom(ID)$ .

**DEFINITION 1 (CATEGORY SCHEMA).** A category schema  $S$  is a pair  $(ID, A)$ , where  $ID \in \mathbf{I}$  is a distinguished attribute denoted identifier, and  $A = \{attr | attr \in \mathbf{A}\}$ . Without loss of generality, and for simplicity, in what follows we consider the set  $A$  ordered. Thus,  $S$  has the form  $(ID, attr_1, \dots, attr_n)$ .  $\square$

**DEFINITION 2 (CATEGORY OCCURRENCE).** Given a category schema  $S$ , a category occurrence for  $S$  is the pair  $(\langle ID, id \rangle, P)$ , where  $ID$  is the ID attribute of Definition 1 above,  $id \in dom(ID)$ , and  $P$  is the set of pairs  $\{(attr_1, v_1), \dots, (attr_n, v_n)\}$ , where: (a)  $attr_i = A(i)$  (remember that  $A$  is considered ordered); (b)  $v_i \in dom(attr_i), \forall i, i = 1..n$ ; (c) All the occurrences of the same category have the same set of attributes; (d)  $ID$  is unique for a category occurrence, meaning that no two occurrences of the same category can have the same value for  $ID$ . (see below)  $\square$

**REMARK 1.** In what follows, for clarity reasons, we assume that  $attr_0$  stands for  $ID$ . Thus, a category occurrence is the set of pairs  $[(attr_0, v_0), (attr_1, v_1), \dots, (attr_n, v_n)]$ .  $\square$

**DEFINITION 3 (CATEGORY INSTANCES).** A set of occurrences of the same category is denoted a category instance. Also, given set of category instances (see Figure 1), we extend the fourth condition in Definition 2 to hold for the whole set:  $ID$  is unique for a set of category instances, meaning that no two occurrences of categories in the set can have the same value for  $ID$ .  $\square$

**EXAMPLE 1.** The schemas of the four categories in our example are shown in Table 1. The corresponding set of category instances is shown in Figure 1 (for example, the category hotels has two occurrences).  $\square$

Adding a time interval to a category occurrence, produces an **item**. The time interval of an item is described by its initial and final instants, and denoted  $[ts, tf]$ . Definition 4 spells the above out.

**DEFINITION 4 (ITEM).** Let  $S$  be a category schema, and  $O(S)$  a category occurrence of the form  $[(ID, v), (attr_1, v_1), \dots, (attr_n, v_n)]$ . An item  $I$  associated with  $O(S)$  is the set of pairs:  $[(ts, v_{ts}), (tf, v_{tf}), (ID, v), (attr_1, v_1), (attr_n, v_n)]$ , where  $ts$  and  $tf$  are temporal attributes corresponding to the beginning and ending of the time interval of the occurrence, and  $v_{ts}$  and  $v_{tf}$  are actual values for these attributes.  $\square$

**REMARK 2.** When necessary, we decompose the temporal attributes  $ts$  and  $tf$  into date and time parts of the form  $ts\_date, ts\_time, tf\_date$  and  $tf\_time$ , respectively. This allows to talk easily about the different parts of the day, and an implementation can make use of the many features provided by DBMSs to handle temporal data types. Nevertheless, it must be clear that we can indistinctly use both forms of referring to these temporal attributes.  $\square$

**DEFINITION 5 (STRICT TIME INTERVAL OVERLAP).** Let  $I_1 = [ts_1, tf_1]$  and  $I_2 = [ts_2, tf_2]$ , be two time intervals. We say that there is a Strict Time Interval Overlap between them, if  $I_1 \neq I_2$  and neither  $tf_1$  precedes  $ts_2$  nor  $tf_2$  precedes  $ts_1$ .  $\square$

**DEFINITION 6 (TABLE OF ITEMS).** Given a finite set of items  $\mathcal{I}$ , the schema of a Table Of Items (ToI) for  $\mathcal{I}$  is the pair  $T = (OID, Items)$ . An instance of  $T$  is a finite set of tuples of the form  $\langle O_j, i_k \rangle$  where  $i_k \in \mathcal{I}$  is an item associated with the object  $O_j$ . Given  $\langle O_j, i_k \rangle$  and  $\langle O_j, i_m \rangle$ , two tuples corresponding to the same object, then, either both time intervals of  $i_k$  and  $i_m$  are the same, or there is no strict time interval overlap between them.  $\square$

**EXAMPLE 2.** Figure 2 shows an instance of a ToI corresponding to the category instances of Figure 1. Note that the first two items for  $OID = O_2$  have the same ID because they correspond to the same category occurrence:  $[(categoryName, zoo), (ID, Z), (geom, pol7), (price, cheap)]$ . For the attribute geom, we assume that  $pol7$  stores the geometric extension of  $Z$ .  $\square$

The Data Model includes a set of functions  $\mathcal{F} = \{f_1, f_2, \dots, f_n\}$  such that each  $f_i$  can be applied to an attribute of an item. A distinguished function  $Val$  returns the value of an attribute.

**DEFINITION 7 (VALUATION OF AN ATTRIBUTE).** Let  $(attr, v)$  be a pair in a category occurrence; a valuation of  $attr$  is obtained applying a function  $Val$  such that  $Val(attr) = (v)$ .  $\square$

**EXAMPLE 3.** When the last item in Figure 2 is instantiated, a valuation  $Val(ts\_date) = 19/08/2008$  is applied.  $\square$

Besides the  $Val$  function, other (application-dependant) functions can be defined ad-hoc. For instance, in a GIS environment we may have the function *Contains*, that, applied to an attribute of type geometry and a constant geometry, returns *true* if the first one completely contains the second one. We can take advantage of the fact that our model supports a family of functions defined above, to integrate an OLAP environment in this setting. In OLAP, data is aggregated along so-called dimensions, usually organized in hierarchies. Each hierarchy is composed of levels (categories). Each category is associated with instances, and between instances of two levels in a hierarchy, a function (denoted *rollup*) is defined. Thus, at the instance level, a dimension consists in a set of rollup functions that define how aggregation is performed [4]. The rollup functions can be part of the family of functions defined above, in order to add OLAP capabilities in our data model. For example, if an attribute of an item has a hierarchy associated with it, and such attribute is the bottom level of this hierarchy, we can apply the rollup function to an instance of the attribute as follows: for the last item in Figure 2,  $rollup(ts\_date, quarter, Time)$  returns  $Q3$ , meaning that the function is applied to an element ( $ts\_date$ ) in the bottom level of the Time dimension, and the range of the function is the dimension level “quarter”. We make extensive use of a Time dimension, which may have the levels *day, month, quarter* and *year* organized in the hierarchy  $date \rightarrow day \rightarrow month \rightarrow quarter \rightarrow year$ . A relational view of this dimension can contain, for instance, the tuple  $\langle 22/09/07, 22, September, Q3, 2007 \rangle$ . We omit further details for the sake of space, and, in Section 3, we give an example of the use of these rollup functions in our language.

**DEFINITION 8 (VALUATION OF AN ITEM).** Let  $I$  be an Item, and  $\mathcal{F}$  a set of functions  $\{f_1, f_2, \dots, f_n\}$ , such that each  $f_i$  maps the value  $v$  in a pair  $(attr, v) \in I$  to a single value. In addition to  $v$ ,  $f_i$  can have other constants as arguments. We generically

Category	Instance
hotels (2 occurrences)	$[(ID, H1), (categoryName, hotel), (geom, pol1), (star, 3)]$ $[(ID, H2), (categoryName, hotel), (geom, pol2), (star, 5)]$
restaurants (3 occurrences)	$[(ID, R1), (categoryName, restaurant), (geom, pol3), (typeOfFood, French), (price, cheap)]$ $[(ID, R2), (categoryName, restaurant), (geom, pol4), (typeOfFood, French), (price, expensive)]$ $[(ID, R3), (categoryName, restaurant), (geom, pol5), (typeOfFood, Italian), (price, cheap)]$
Eiffel Tower (1 occurrence)	$[(ID, E), (categoryName, EiffelTower), (geom, pol6)]$
zoos (1 occurrence)	$[(ID, Z), (categoryName, zoo), (geom, pol7), (price, cheap)]$

Figure 1: Set of instances for the categories in Table 1

OID	Items
$O_1$	$[(ts\_date, 04/08/2008), (ts\_time, 14:05), (tf\_date, 04/08/2008), (tf\_time, 14:33), (ID, R2), (categoryName, restaurant), (geom, pol4), (typeOfFood, French), (price, expensive)]$ $[(ts\_date, 04/08/2008), (ts\_time, 15:10), (tf\_date, 04/08/2008), (tf\_time, 16:05), (ID, E), (geom, pol6)]$ $[(ts\_date, 04/08/2008), (ts\_time, 17:30), (tf\_date, 04/08/2008), (tf\_time, 18:48), (ID, R3), (categoryName, restaurant), (geom, pol5), (typeOfFood, Italian), (price, cheap)]$ $[(ts\_date, 08/08/2008), (ts\_time, 06:22), (tf\_date, 08/08/2008), (tf\_time, 07:05), (ID, R1), (categoryName, restaurant), (geom, pol3), (typeOfFood, French), (price, cheap)]$ $[(ts\_date, 08/08/2008), (ts\_time, 10:00), (tf\_date, 08/08/2008), (tf\_time, 13:00), (ID, E), (geom, pol6)]$ $[(ts\_date, 08/08/2008), (ts\_time, 17:10), (tf\_date, 08/08/2008), (tf\_time, 18:17), (ID, R1), (categoryName, restaurant), (geom, pol3), (typeOfFood, French), (price, cheap)]$
$O_2$	$[(ts\_date, 03/08/2008), (ts\_time, 11:00), (tf\_date, 03/08/2008), (tf\_time, 11:15), (ID, Z), (geom, pol7), (price, cheap)]$ $[(ts\_date, 08/08/2008), (ts\_time, 18:30), (tf\_date, 08/08/2008), (tf\_time, 21:00), (ID, Z), (geom, pol7), (price, cheap)]$ $[(ts\_date, 19/08/2008), (ts\_time, 09:00), (tf\_date, 19/08/2008), (tf\_time, 10:20), (ID, R1), (categoryName, restaurant), (geom, pol3), (typeOfFood, French), (price, cheap)]$ $[(ts\_date, 19/08/2008), (ts\_time, 17:00), (tf\_date, 19/08/2008), (tf\_time, 18:12), (ID, R2), (categoryName, restaurant), (geom, pol4), (typeOfFood, French), (price, expensive)]$

Figure 2: An instance of the ToI

denote these additional arguments  $\mathcal{A}$ . A valuation of  $I$  with  $\mathcal{F}$ , denoted  $\mathcal{V}(I, \mathcal{F})$  is the item resulting from applying  $\mathcal{F}$  to  $I$  as follows: pick one  $f_i$  in  $\mathcal{F}$  and apply it to the value  $v$  in a pair  $(attr_j, v)$  of  $I$ , probably using some constants in  $\mathcal{A}$ . Repeat the process with the remaining pairs, until all pairs have been valued. Note that the same function could be applied to more than one pair. If the function  $\text{Val}$  is applied to all the pairs in  $I$ , we obtain the identity valuation, i.e., the valuation of the item is the item itself.  $\square$

EXAMPLE 4. If we apply the rollup function to the first pair of the last item in Figure 2, and the  $\text{Val}()$  function to all other pairs in the item, we obtain the valuation  $[(ts\_date, Q3), (ts\_time, 17 : 00), (tf\_date, 19/08/2008), (tf\_time, 18 : 12), (categoryName, restaurant), (ID, R2), (geom, pol4), (typeOfFood, French), (price, expensive)]$ .  $\square$

DEFINITION 9 (TRANSFORMED SUBITEM). Given an item  $I$ , a set of functions  $\mathcal{F}$ , and a valuation of  $I$  with  $\mathcal{F}$ ,  $\mathcal{V}(I, \mathcal{F})$ , we denote any subset of  $\mathcal{V}$  a Transformed Subitem,  $TS(I)$ .  $\square$

EXAMPLE 5. In Example 4, the valuation of the item yields the transformed subitems  $[(tf\_date, 19/08/2008), (tf\_time, 18:12), (price, expensive)]$ ,  $[(ts\_date, Q3), (price, expensive)]$ , and other ones we omit for the sake of brevity.  $\square$

DEFINITION 10 (ITEMSET). An itemset  $(i_1, i_2, \dots, i_n)$  is a non-empty set of items, where  $n \geq 1$ , and for all  $i_k, k = 1..n$ , the  $ts\_date, ts\_time, tf\_date, tf\_time$  values are the same.  $\square$

REMARK 3. In the moving objects setting, since each moving object can be in only one place at each moment, all itemsets belonging to the same OID contain exactly one item.  $\square$

DEFINITION 11 (SUB-ITEMSET). Let  $IS = (i_1, i_2, \dots, i_n)$ , be an itemset. A sub-itemset of  $IS$  is a subset of  $(TS(i_1), TS(i_2), \dots, TS(i_n))$ , where  $TS(i_i)$  is any transformed subitem of  $i_i$ .  $\square$

DEFINITION 12 (SEQUENCE). A sequence is an ordered list of itemsets  $(i_1, i_2, \dots, i_m)$  such that, for every pair of integers  $j, g, j < g \Rightarrow \text{Val}(i_j, tf, v) < \text{Val}(i_g, ts, v)$  holds. (The  $i.a$  notation means that  $a$  is an attribute of item  $i$ .  $\square$

DEFINITION 13 (SUBSUMED SEQUENCE). We say that a sequence  $\langle a_1, a_2, \dots, a_n \rangle$  subsumes another sequence  $\langle b_1, b_2, \dots, b_n \rangle$  if  $\forall i \in 1..n, b_i$  is a sub-itemset of  $a_i$ .  $\square$

DEFINITION 14 (CONTIGUOUS LIST). Given a ToI instance with tuples of the form  $\langle O_j, i_k \rangle$ , let us denote  $\text{Items}(O_j)$  the set of items  $i_k$  associated with  $O_j$ . Also let  $CL(O_j) \subseteq \text{Items}(O_j)$ . We say  $CL(O_j)$  is a contiguous list for  $O_j$ , if  $\forall i \in \text{Items}(O_j)$  and  $i \notin CL(O_j)$ , the starting time of  $i$  (denoted  $v_{ts}(i)$ ) is less than the starting time of all the items in  $CL(O_j)$ , or all the starting times of the items in  $CL(O_j)$  are less than  $v_{ts}(i)$ . Note that a contiguous list is also a sequence.  $\square$

Now we are ready to give a precise definition of the notion of Support of a sequence.

DEFINITION 15 (SUPPORT). Given a ToI instance with tuples of the form  $\langle O_j, i_k \rangle$ . The support of a sequence  $S$  is the fraction of the different objects  $O_j$  in the ToI, associated with a contiguous list  $CL(O_j)$  which subsumes  $S$ .  $\square$

Definitions 10 through 15 can be more clearly understood through an example outside the MOD domain, i.e., in a scenario where itemsets are of size  $> 1$ . Example 6 below is adapted from classical data mining literature, and shows that our approach is general enough to capture other domains..

EXAMPLE 6. Consider the taxonomy shown in Figure 3, and a corresponding ToI instance in Figure 4. There are two different objects,  $C1$  and  $C2$ . We show below that the sequence  $S = (\langle (ID, F) \rangle, \langle (ID, Tolkien) \rangle)$  has a support of 100%, using the notions introduced above (note that the second part of  $S$  is obtained as a generalization of the ID attribute). The taxonomy can be seen as a rollup function where, for example,  $\text{rollup}(F) = \text{'Tolkien'}$ , and  $\text{rollup}(I) = \text{'Clarke'}$ .

The contiguous list  $CL(C1)$  is composed of the second, third and fourth items in Figure 4. This sequence contains two itemsets. The first one is  $(\langle (ts\_date, 10/10/99), (ts\_time, 00:02), (tf\_date, 10/10/99), (tf\_time, 00:02), (categoryName, Book), (ID, F) \rangle)$ . Applying the  $\text{Val}$  function and considering only the last pair, we obtain the sub-itemset  $(\langle (ID, F) \rangle)$ , which coincides with the first itemset of  $S$ . The second itemset is composed of two items:

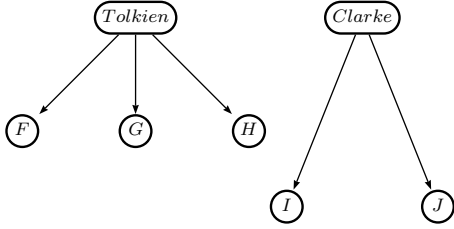


Figure 3: A taxonomy for Example 6

ID	Items
C1	[(ts_date,10/10/99), (ts_time,00:01), (tf_date,10/10/99), (tf_time,00:01), (categoryName,Book), (ID,I)]
C1	[(ts_date,10/10/99), (ts_time,00:02), (tf_date,10/10/99), (tf_time,00:02), (categoryName,Book), (ID,F)]
C1	[(ts_date,10/10/99), ((ts_time,00:15), (tf_date,10/10/99), (tf_time,00:15), (categoryName,Book), (ID,J)]
C1	[(ts_date,10/10/99), ((ts_time,00:15), (tf_date,10/10/99), (tf_time,00:15), (categoryName,Book), (ID,H)]
C2	[(ts_date,10/10/99), ((ts_time,00:01), (tf_date,10/10/99), (tf_time,00:01), (categoryName,Book), (ID,F)]
C2	[(ts_date,10/10/99), ((ts_time,00:01), (tf_date,10/10/99), (tf_time,00:01), (categoryName,Book), (ID,I)]
C2	[(ts_date,10/10/99), ((ts_time,00:20), (tf_date,10/10/99), (tf_time,00:20), (categoryName,Book), (ID,G)]
C2	[(ts_date,10/10/99), ((ts_time,00:50), (tf_date,10/10/99), (tf_time,00:50), (categoryName,Book), (ID,J)]

Figure 4: Instance of a ToI containing two objects (C1, C2)

$[(ts\_date, 10/10/99), (ts\_time, 00:15), (tf\_date, 10/10/99), (tf\_time, 00:15), (categoryName, Book), (ID, J)]$ , and  $[(ts\_date, 10/10/99), (ts\_time, 00:15), (tf\_date, 10/10/99), (tf\_time, 00:15), (categoryName, Book), (ID, H)]$ . Applying the Val function over all attributes (except ID, where we apply the rollup function) of the second item, we obtain the sub-itemset  $\{(ID, Tolkien)\}$ . This sub-itemset coincides with the second itemset of S. Thus, the object C1 contributes to the support of the sequence S. With a similar analysis, we can show that C2 also contributes to the support of S. Then, we conclude that the support of the sequence S is 100%.  $\square$

### 3. THE RE-SPaM LANGUAGE

We now introduce a language based on regular expressions where, instead of atomic items (as in previous proposals), the atoms are constraints expressed as formulas over attributes of the complex items defined in Section 2. The grammar for the constraints is given in Table 2. The regular expression language is built in the usual way, supporting the standard operators. The meaning of these operators is specified in Figure 5, and the precedence is the usual one ('()', '\*', '+', '?', ':', '|'). The language also supports variables (strings preceded by '@').

#### 3.1 Syntax and Semantics

**DEFINITION 16 (TERMS).** *There exist no terms other than the following ones: (1) Constants: a literal enclosed by simple quotes. For example, '3' for the integer three, '12/10/2007' for a date. (2) Attributes: See Definition 4. There exist two types of attributes: (a) attributes in categories, which are elements in the category schema (e.g., categoryName, ID, geom, price); (b) temporal attributes, i.e., attributes which identify temporal occurrences of an item. They are denoted ts\_date, tf\_date, ts\_time and tf\_time. (3) Variables: a literal that begins with the '@' symbol. For example, @x, @Y1, etc. (4) Functions of n arguments: An expression fn(attribute, 'ct1',*

Symbol	Meaning
	Disjunction. For example, C   D expresses "C" or "D".
.	Concatenation. For example, C.D expresses that constraint "D" immediately follows constraint "C".
*	Zero or more occurrences. For example, C* expresses that constraint "C" holds zero or more times.
+	One or more occurrences. For example, C+ expresses that constraint "C" holds one or more times.
?	Zero or one occurrence. For example, C? expresses that "C" is optional.

Figure 5: Regular Expression operators

R1	CONSTRAINT $\leftarrow$ [ CONDITION ]
R2	CONDITION $\leftarrow$ $\lambda$
R2	CONDITION $\leftarrow$ EQ
R2	CONDITION $\leftarrow$ EQ $\wedge$ CONDITION
R3	EQ $\leftarrow$ attr = 'constant'
R3	EQ $\leftarrow$ attr = @vble
R3	EQ $\leftarrow$ functionName(attr, ...) = 'constant'
R3	EQ $\leftarrow$ functionName(attr, ...) = @vble

Table 2: Grammar for constraints

'ct2', ..., 'ct<sub>n-1</sub>'),  $n \geq 1$ , is a function where the first parameter is an attribute and all the other ones are constants.  $\square$

**DEFINITION 17 (FORMULA).** *Let C, V, A and F be a set of constants, variables, attributes and functions, respectively. The expression term1 = term2 is a formula, where term1  $\in A \cup F$ , term2  $\in C \cup V$ , and '=' is the equality symbol. Moreover, if  $\mathcal{F}_1$  and  $\mathcal{F}_2$  are formulas,  $\mathcal{F}_1 \wedge \mathcal{F}_2$  is also a formula.  $\square$*

In RE-SPaM, a constraint is a formula enclosed in squared brackets. We denote this formula a *condition* (Table 2). There are two kinds of conditions: (a) empty condition: satisfied by every item; (b) non-empty condition: built as a *conjunction* of terms.

**EXAMPLE 7.** *The expression [].price = 'cheap'] includes two constraints. The first one is an empty condition, satisfied by all the items in an instance of a table of items (in what follows, ToI). The second one expresses the equality condition. In our running example it is satisfied by the items identified by Z, R1 and R3.*

Constraints can include functions over attributes. In our running example we use functions over OLAP hierarchies. These functions have the form  $rollup(attribute, 'level\_i', 'dimension\_k') = 'c'$ , and  $rollup(attribute, 'level\_i', 'dimension\_k') = @v$ . In the first case, the function evaluates to *true* whenever in the hierarchy of  $dimension\_k$ , the value of  $attribute$  in the bottom level rolls up to the member 'c' in the level  $level\_i$ . Analogously, in the second case, the function evaluates to *true* whenever in the hierarchy of  $dimension\_k$  the value of  $attribute$  in the bottom level rolls up to a member which matches an instantiation of the variable @v in level  $level\_i$ .

In RE-SPaM, a constraint is expressed as a regular expression  $\mathcal{R}$ ; to evaluate if a sequence satisfies the constraint, we build a DFA  $\mathcal{A}_{\mathcal{R}}$  which accepts the language generated by  $\mathcal{R}$ .

#### 3.2 RE-SPaM by Example

In this section through a set of queries, we give the reader the intuition of what RE-SPaM can express, and how it differs from, and substantially improves, other proposals. For example, existing efforts force the user to enumerate the IDs of the items to express disjunctions, like  $(A|B|C|D)^*$ . When the number of items becomes large, this solution would not be applicable. RE-SPaM allows writing concise expressions using the semantic information available.

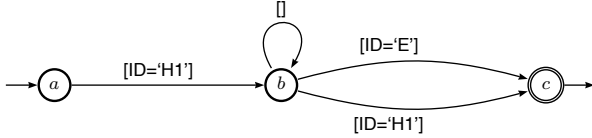


Figure 6: Automaton for Q2

Expressions can be built with attributes, functions, constants and variables. We now present examples of the different kinds of expressions supported by RE-SPaM.

*Constraints without variables.*

**Q1:** Trajectories of tourists who visit hotel H1, then optionally stop at restaurant R3 and the Zoo, and either end at H1 or visiting the Eiffel Tower.

$[ID='H1'].([ID='R3'].[ID='Z'])*.([ID='E']|[ID='H1'])$

Note that Q1 uses only ID attributes in all its subexpressions.

**Q2:** Trajectories that visit hotel H1, then, optionally visit different places, and finish at the Eiffel Tower or going back to H1.

$[ID='H1'].[]*.([ID='E']|[ID='H1'])$

The use of the empty condition allows avoiding the enumeration of all the items. If an expression includes an empty condition, during the mining process it is instantiated with all the IDs of the category instances. Figure 6 shows the DFA that accepts the language generated by the expression.

**Q3:** Trajectories of tourists who visit hotel H1 and then a cheap place or a place serving French food.

$[ID='H1'].([price='cheap']|[typeOfFood='French'])$

Q3 contains a subexpression with no ID. The disjunction is evaluated as follows. Places with cheap prices are R1, R3 and Z, and places that serve French food are R1 and R2. During sequential pattern mining, we compute the items which satisfy these conditions, without the need of explicit enumeration of all the possibilities (note that the expression is equivalent to  $[ID='H1']([ID='R1'] | [ID='R3'] | [ID='Z'] | [ID='R2'])$ ).

**Q4:** Trajectories that visited hotel H1 and then some cheap place, on 10/10/2007.

$[ID='H1'].([ts\_date='10/10/2007' \wedge price='cheap' ])$

Q4 contains a subexpression with a *temporal attribute*, which characterize the occurrences of items in the database of sequences.

**Q5:** Trajectories that visit a cheap place during the third quarter of any year.

$[rollup(ts\_date, 'quarter', 'Time')='Q3' \wedge price='cheap']$

Q5 contains a *rollup function* over time. Like in the previous query, during mining, by accessing the ToI we compute the items that satisfy the temporal constraint.

*Constraints with variables.*

**Q6:** Trajectories that start at a place characterized by price (i.e., a place such that *price* is an attribute of the item representing this kind of place), then stop either at the zoo or the Eiffel Tower, and end up going to a place that serves French food, and has the same price range as the initial stop. Figure 7 shows the DFA that accepts the language generated by the regular expression (we use this automaton later in the paper).

$[price=@x].([ID='Z'] | [ID='E']).[typeOfFood='French' \wedge price=@x]$

In our running example, 'cheap' and 'expensive' are the only possible values for prices; thus, the only valid combinations are: cheap-cheap and expensive-expensive. Sequences such as {H1 Z R1} and {Z E R2} do not satisfy the query. The first one because hotel H1 is not characterized by price, the second one because Z has

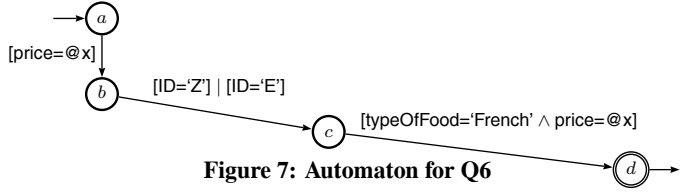


Figure 7: Automaton for Q6

cheap prices but R2 is expensive. On the other hand, the sequence {Z Z R3} does satisfy the query. In our implementation, variables are bound to items during the mining process as we explain later.

**Q7:** Trajectories that stopped at two places (the second one having cheap prices), at the same part of the day (e.g., both of them during the morning), on 10/10/2008.

$[rollup(ts\_time, 'range', 'Time')=@z \wedge ts\_date='10/10/2008']$ .

$[rollup(ts\_time, 'range', 'Time')=@z \wedge ts\_date='10/10/2008' \wedge price='cheap']$

Q7 uses variables that the system binds (during the mining process), to the result of applying a function over temporal attributes.

*Metadata constraints.*

Although, in general, variables are used to express matching conditions between different subexpressions, they can also be used to constraint items according to their structure. We call these kinds of expressions, metadata constraints.

**Q8:** Trajectories that visited a place characterized by price.

$[price=@Z]$

The semantics here is: a sequence is in the result if it contains an item with an attribute *price* in it. The variable here is not bound to any particular value. As a more involved example, the constraint  $[price=@x]^+$  is verified by sequences of one or more item (not necessary the same ones), all of them with the same price. In our running example, Z, R1, R2 and R3 are the items that satisfy this constraint. Let us analyze another example.

**Q9:** Trajectories that visited a place characterized by price, and finished somewhere, on October 10th, 2006.

$[price=@x].[ts\_date='10/10/2006']$

In our running example, only items Z, R1, R2 and R3 can satisfy the first constraint.

**Discussion.** In SPIRIT [6, 7], regular expressions are used to restrict the sequences produced by the mining algorithm. However, SPIRIT deals only with the IDs of items. Using attributes (supported by the formal model introduced in Section 2), lets us avoid the enumeration of items, as we explained before. But, besides *writing concise queries*, our proposal supports the binding of variables to any attribute (or function over an attribute) during the mining process. As far as we are concerned, this is the first proposal in this sense, in pattern discovery. We showed above that in RE-SPaM, variables are not only used for matching between unknown values of different attributes, but also for checking the existence of an attribute in an item. The next example illustrates this feature.

**EXAMPLE 8.**  $([star = @c][price = 'expensive']).[star = @c]$  is satisfied by sequences where the first item is expensive and the second item is characterized by a 'star' attribute, and also it can be satisfied by sequences where the first and last item are characterized by the same 'star' attribute. In the first case the variable is only used for restricting items (those with 'star' attribute) and in the second case variables are also used for matching values.

**REMARK 4.** Note that variables can also appear in optional paths (i.e., |, ?), or affected by a Kleene closure operator.

## 4. RE-SPAM EVALUATION

### 4.1 Preliminary considerations

We explained in Section 2 that the ToI is composed of itemsets such that items are associated with an OID. Since we assume that, at every time instant, each OID can only be in exactly only one place, *itemsets are of length one*. Nevertheless, the algorithm we present can be applied to itemsets of any length. We work with the category instances depicted in Figure 1. Temporal information associated with item occurrences is stored in the ToI (Figure 2), which in our implementation is decomposed (i.e., normalized) as follows: we have a table with schema (OID, ts\_date, ts\_time, tf\_date, tf\_time, ID), where ID is the identifier of the corresponding category instance. All other attributes of the ToI are stored in a different structure, and retrieved via the ID. Moreover, we implemented two alternative approaches: (a) information about category instances is stored in an XML file; (b) information is stored in a relational table, accessing the data through a join operation. Figure 8 shows the normalized ToI instance for our running example.

Computing the support of a sequence requires computing its Transformed Subitems (Definition 9).

**EXAMPLE 9.** Consider the regular expression  $[price = @x]. [price = @x \wedge typeOfFood = 'French']$ . To obtain the Transformed Subitems we use the function  $\mathcal{F} = \{Val\}$  over the attributes price (for the first subexpression), and attributes price and typeOfFood (for the second one). Now, let us denote  $S$  the sequence of the transaction with  $OID=O2$  composed of two sub-itemsets, the second and third lines in Figure 2, each one containing occurrences of items  $Z$  and  $R1$  respectively. Let us write again these items, for clarity reasons:

$Z = \{(ts, '08/08/2008 18:30'), (tf, '08/08/2008 21:00'), (ID, 'Z'), (price, 'cheap')\};$

$R1 = \{(ts, '19/08/2008 09:00'), (tf, '19/08/2008 10:20'), (ID, 'R1'), (price, 'cheap'), (typeOfFood, 'French')\};$

The question is: which are the sub-sequences supported by  $S$ ? Since the first itemset of  $S$  is composed only by item  $Z$ , all of its sub-itemsets are obtained building subsets of the Transformed Subitem  $TS(Z)$ , using  $\mathcal{F}$ , and the price attribute. These sub-itemsets are:  $\emptyset$  and  $\{(price, 'cheap')\}$ . The second itemset of  $S$  is composed only by item  $R1$ ; thus, its sub-itemsets are obtained building subsets of  $TS(R1)$ , using  $\mathcal{F}$  and the attributes price and typeOfFood. This sub-itemsets are:  $\emptyset$ ,  $\{(price, 'cheap'), (typeOfFood, 'French')\}$ . Then, the subsequences of  $S$  satisfying the regular expression are the ones whose items can be transformed to  $\{(price, 'cheap')\}$ , and  $\{(price, 'cheap'), (typeOfFood, 'French')\}$ . In our running example, these sequences are:  $\{Z\}$ ,  $\{R1\}$ ,  $\{R3\}$  for the first transformation;  $\{R1\}$  for the second transformation; and  $\{Z R1\}$ ,  $\{R1 R1\}$  and  $\{R3 R1\}$  for both of them. Note that, even  $S$  and the sequences  $\{R3 R1\}$  and  $\{R1 R1\}$  are not exactly the same, they can be considered semantically equivalent with respect to the attribute price, i.e., both of them are associated with a 'cheap' price.  $\square$

### 4.2 Incremental phases in the mining process

Typically, in GSP-based algorithms, frequent sequences with a user-specified minimum support are computed in incremental phases. At each intermediate step  $k$ , the following occurs: (1) A temporary set  $C_k$  is built using the previous set  $C_{k-1}$ . Its elements are candidate sequences of length  $k$ . (2) Each element in  $C_k$  which contains at least one sub-sequence with support less than the minimum is discarded due to anti-monotony property ( $C_{k-1}$  is analyzed). (3) The database is accessed in order to analyze support, and each element in  $C_k$  with at least minimum support is added to the set  $F$

of frequent sequences. (4) When an empty  $C_k$  set is obtained,  $F$  contains the frequent sequences with minimum support.

In RE-SPaM, a constraint is expressed as a regular expression  $\mathcal{R}$ ; to evaluate if a sequence satisfies it, we build a DFA, denoted  $\mathcal{A}_{\mathcal{R}}$  which accepts the language generated by  $\mathcal{R}$ . The idea of using  $\mathcal{A}_{\mathcal{R}}$  for pruning  $C_k$  before querying the database was first proposed in the SPIRIT algorithm. There, instead of using the original constraint  $C$ , a relaxed constraint  $C'$  (not necessarily anti-monotonic) is used during the mining process. When  $C'$  is not anti-monotonic, the second phase above is replaced with a strategy consisting in pruning the sequences in  $C_k$  which contain at least one sub-sequence which *satisfies*  $C'$  and does not have minimum support. In the last phase,  $F$  is analyzed to obtain the frequent sequences that *satisfy*  $C$ , i.e., a strict  $C$  verification is carried out. In what follows, we use a relaxed constraint  $C'$  that accepts the sequences (denoted *legal*) which correspond to a path in  $\mathcal{A}_{\mathcal{R}}$ . Informally, if the automaton accepts only two words "abbd" and "acabd", then the sequence "cab" is not pruned because it is a substring of the second one, but the sequence "cbd" is pruned because it is not a substring of any of these words. Other variations for  $C'$  are discussed in Section 5.

We identify three phases when building  $C_k$ : (i)  $C_k$  *population*:  $C_k$  is populated using the information previously obtained. (ii)  $C_k$  *pruning by  $\mathcal{A}_{\mathcal{R}}$* :  $C_k$  is pruned using the automaton and perhaps some extra information. If a candidate sequence does not satisfy the relaxed constraint  $C'$ , it is discarded at this moment. (iii)  $C_k$  *pruning by the ToI instance*:  $C_k$  is pruned using the ToI instance, as we explain later, and added to a set  $F$  of frequent candidate sequences. Finally,  $F$  is pruned using the original constraint  $C$ .

### 4.3 The RE-SPaM algorithm

*Using  $\mathcal{A}_{\mathcal{R}}$ : Check by verification.* During the candidate generation step, sequences that do not satisfy the constraints must be pruned, given that they do not contribute to the result. We discuss here how RE-SPaM uses the automaton to prune candidate sequences while generating intermediate  $C_k$ 's, *without accessing the ToI*. In step  $k$ , once  $C_k$  has been populated with candidate sequences of length  $k$ , we use  $\mathcal{A}_{\mathcal{R}}$  for pruning ( $\mathcal{A}_{\mathcal{R}}$  is stored in main memory). Using  $\mathcal{A}_{\mathcal{R}}$  we check which candidate sequences satisfy the relaxed constraint  $C'$ . In this step, the algorithm finds out whether or not the conditions in the edges of paths of length  $k$  in  $\mathcal{A}_{\mathcal{R}}$  are satisfied by candidate sequences in  $C_k$ . The edges of the automaton are labeled *with constraints*. This is a relevant difference with existing approaches (where edges are labeled with IDs).

Let  $p = \{e_1, e_2, \dots, e_k\}$  be a path of length  $k$  in  $\mathcal{A}_{\mathcal{R}}$ , and let  $cs = \{ID = 'ID_1', ID = 'ID_2', \dots, ID = 'ID_k'\}$  be a candidate sequence in  $C_k$ . We use  $\mathcal{A}_{\mathcal{R}}$  to determine whether or not each one of the items identified by  $ID_j$ ,  $j \in 1..k$ , satisfies the constraint that labels the edge  $e_j$ . Thus, we need to find the values of the attributes that characterize the item identified by  $ID_j$ . Notice that our intention is to prune  $C_k$  *without* accessing the ToI; thus, the only sources of information we can use are the automaton and the *category instances* (this table is also likely to be stored in main memory), and the only kinds of attributes that can be analyzed are the ones in categories. The analysis of *temporal attributes* is postponed to a later stage. In summary, during this step we only use the automaton and category instances to *verify* the sub-conditions that do not involve temporal attributes, and postpone the evaluation of the conditions over temporal attributes. Note that, actually, in this step we compute the *transformed subitem* of Definition 9.

**EXAMPLE 10.** In query  $Q6$ , the sub-conditions to evaluate are four:  $price = @x$ ,  $ID = 'Z'$ ,  $ID = 'E'$  and  $typeOfFood = 'French'$ . As

OID	Items
$O_1$	(([ts_date,04/08/2008],[ts_time,14:05],[tf_date,04/08/2008],[tf_time,14:33],[ID,R2])) (([ts_date,04/08/2008],[ts_time,15:10],[tf_date,04/08/2008],[tf_time,16:05],[ID,E])) (([ts_date,04/08/2008],[ts_time,17:30],[tf_date,04/08/2008],[tf_time,18:48],[ID,R3])) (([ts_date,08/08/2008],[ts_time,06:22],[tf_date,08/08/2008],[tf_time,07:05],[ID,R1])) (([ts_date,08/08/2008],[ts_time,10:00],[tf_date,08/08/2008],[tf_time,13:00],[ID,E])) (([ts_date,08/08/2008],[ts_time,17:10],[tf_date,08/08/2008],[tf_time,18:17],[ID,R1]))
$O_2$	(([ts_date,03/08/2008],[ts_time,11:00],[tf_date,03/08/2008],[tf_time,11:15],[ID,Z])) (([ts_date,08/08/2008],[ts_time,18:30],[tf_date,08/08/2008],[tf_time,21:00],[ID,Z])) (([ts_date,19/08/2008],[ts_time,09:00],[tf_date,19/08/2008],[tf_time,10:20],[ID,R1])) (([ts_date,19/08/2008],[ts_time,17:00],[tf_date,19/08/2008],[tf_time,18:12],[ID,R2]))

Figure 8: Normalized ToI instance

none of them involve temporal attributes, all of them can be analyzed using  $\mathcal{A}_{\mathcal{R}}$  and category instances.

However, in the case of query Q7, the sub-conditions are three:  $rollup(ts\_time, 'range', 'time dimension')=@z, ts\_date='10/10/2008'$  and  $price='cheap'$ . The only sub-condition that can be analyzed using  $\mathcal{A}_{\mathcal{R}}$  and category instances is  $price='cheap'$ .  $\square$

We have said that, in general, variables are used to match different constraints within the same expression. However, in the intermediate phases, when building set  $C_k$ , only sub-paths of length  $k$  are considered. If the same variable is used in both extremes of a path of length  $k$ , sets  $C_j$  with  $j < k$  are not useful for detecting if these variables coincide. Another question that arises is: which is the best strategy when a variable appears only once in an expression? Notice that such a situation can occur either when the user defines an expression which involves a variable only once, or when the system is building  $C_j$  with  $j < k$  and variables appear in the extremes of paths of length  $k$ . In both cases, the system can only check for the existence of the attribute in an item that it is being compared against the variable, and bind its value using the attribute of this item. We can conclude that, even though a priori it seems that early evaluation using the automaton is a good strategy, given that attributes in categories can be affected by functions, or compared against constants or variables, yielding rather complex expressions, it is not trivial to infer if this approach will always be the best, as we show in the next example.

EXAMPLE 11. Expression Q6 looks for a matching in the prices at the beginning and end of a path. The set  $C_1$  is composed of sequences of one item, that satisfy the constraints  $[price = @x]$ ,  $[ID = 'Z']$ ,  $[ID = 'E']$  and  $[typeOfFood = 'French' \wedge price = @x]$ . Also,  $C_2$  is composed of sequences of two items that satisfy the constraints  $[price = @x].[ID = 'Z']$ ,  $[price = @x].[ID = 'E']$ ,  $[ID = 'Z].[typeOfFood = 'French' \wedge price = @x]$  and  $[ID = 'E].[typeOfFood = 'French' \wedge price = @x]$ . Therefore, the binding of the variable  $@x$  is not used in these two phases (we are interested in matching both extremes). Only during  $C_3$  this is relevant, because both bindings can be compared, and perhaps some candidate sequences could be pruned. For example, the sequence  $\{Z Z R2\}$  does not satisfy this constraint because Z is cheap and R2 is expensive. Thus, deciding which strategy is better than the other, particularly when variables are involved, is not trivial. Finally, note that during the computation of  $C_1$ , the automaton  $\mathcal{A}_{\mathcal{R}}$  helps to eliminate H1 and H2 because they do not match any of the edges of the automaton.  $\square$

We propose two possibilities regarding the moment when the verification phase for non-temporal attributes can take place: *early evaluation* and *late or postponed evaluation*. In the former, the system determines if a sub-condition with no temporal attributes belonging to an edge of the automaton is verified by an item when building  $C_k$  and before querying the ToI instance. In the latter, the

verification occurs when the algorithm enters its final phase, i.e., when it must prune the set F using the original constraint C. Thus, verification is postponed until the final phase, and constraints are not checked while building intermediate  $C_k$ 's. Obviously, late or early evaluation only affects performance, not the final result. In the remainder of this paper, except when noted, we assume the following: *early evaluation* for conditions that involve non-temporal attributes and *constants*, and *late evaluation* for conditions that involve non-temporal attributes and *variables*. We compare performance between different combinations of these approaches in Section 5.

Using the ToI. We now discuss the use of the ToI for pruning candidate sequences with support less than the minimum. Assume we have computed  $C_k$ , which now contains the candidate sequences that satisfy the subexpression of length  $k$ . We continue with the analysis of Q6, and we want to discover the sequences that satisfy the constraint, with a support of 100%. At a first glance, in the table of Figure 8 there is one sequence with  $OID = O_1$ , which satisfies the constraint:  $\{R1, E, R1\}$ . With a similar analysis, there exists only one sequence with  $OID = O_2$  which satisfies Q6:  $\{Z, Z, R1\}$ . Since we are interested in categorical mining (i.e., we are using *semantic information*), not just in counting strict occurrences of items, we have to modify the way of counting support. Although none of the two transactions in Figure 8 contains the same sequence, we can say that both satisfy Q6. We do not intend to find the same sequence of IDs (like GSP or SPIRIT do), because we are introducing the idea of decomposing items into their descriptive attributes. Moreover, in this example, Z and R1 are semantically equivalent with respect to Q6, because both have cheap price associated<sup>1</sup>. This is the reason why the algorithm cannot discard a candidate sequence (although it has support less than the minimum), if it is supported by at least one transaction. Figure 9 depicts the three steps to compute  $C_1$ . Note that  $\mathcal{A}_{\mathcal{R}}$  prune H1 and H2 in step2, because they *do not match* any of the edges of the automaton.

REMARK 5. If we had followed the SPIRIT strategy,  $C_1$  would have only contained R1 and R2. Items Z, E, and R3, would have been pruned, because their support is less than the minimum (each of them are in only one transactions). Moreover, no sequential pattern with this support would have been found, given that the regular expression requires that the trajectory stops at Z or E.

In Step 3 (computing  $C_k$ ), we scan the ToI for pruning the candidate sequences not present in any transaction. After this,  $C_k$  is added to the temporary set F. Figures 10 and 11 show how  $C_2$  and  $C_3$  are computed. Given that  $C_4$  is empty, the algorithm enters its final phase, i.e., the *strict verification* of the sequences in F.

<sup>1</sup> In our implementation we also display the list of frequent sequences that have contributed to the computation of the support.

IDs
H1
H2
R1
R2
R3
E
Z

IDs
R1
R2
R3
E
Z

IDs
R1
R2
R3
E
Z

**Figure 9: Computing C1: Step 1 (left), Step 2 (Legal) (center), Step 3 (right)**

IDs
R1 R1
R1 R2
R1 R3
R1 E
R1 Z
R2 R1
...
R3 R1
R3 R2
...
E R1
...
E Z
Z R1
...
Z Z

IDs
R1 E
R1 Z
R2 E
R2 Z
R3 E
R3 Z
E R1
E R2
Z R1
Z R2
Z E
Z Z

IDs
R1 E
R2 E
E R1
Z R1
Z Z

**Figure 10: Computing C2: Step 1 (left), Step 2 (Legal) (center), Step 3 (right)**

The *final phase* uses all sequences in the temporary set F and proceeds as follows. First, it uses the automaton to prune all sequences which are *not accepted*. Notice that here we are using the automaton for *acceptance verification* and not for *legal verification*. Note that using the automaton to find sequential patterns with minimum support does not suffice, i.e., the ToI *must be scanned*. This scan has different goals: for verification of conditions that have been postponed (for example, expressions which involve variables or temporal constraints), and for calculation of minimum support. Until this phase we only know that the sequences in F are present in some transaction. Now, we have to check if the set F has enough support. Recall that we consider all sequences in F equivalent with respect to the regular expression under analysis. In our example, we detect that  $OID=O_1$  supports {R1 E R1} and that  $OID=O_2$  supports {Z Z R1}. Thus, all of these sequences verify the original expression, yielding a support of 100%. Figure 12 shows the set F before and after automaton verification, and set F after the ToI is scanned. As our expression does not involve temporal attributes, the ToI scan does not change sequences in the set F. However, this scan is necessary to compute the support.

**Algorithm details.** Algorithm 1 sketches the procedure for mining sequential patterns using the approach described above.

Once the user defines the regular expression RE and the minimum support, the DFA automaton is built. The relaxation constraint C' is also defined. We explained above that any relaxation constraint can be used (see Section 5 for further details). The algorithm proceeds in incremental phases until the final condition holds, which depends on the relaxation choice. We follow the approach of Garofalakis *et al* [6, 7], who proposed four variations of the algorithm based on this criteria. For example, in the *Legal* algorithm, the final state is reached when no legal sequences of length k with respect the start state of the automaton can be generated.

The main loop is the core of the algorithm. Line 9 corresponds to the generation of candidate sequences. For example, in the *Legal*

IDs
R1 E R1
R2 E R1
E R1 E
Z R1 E
Z Z R1
Z Z Z

IDs
R1 E R1
R2 E R1
Z Z R1

IDs
R1 E R1
R1 E R1
Z Z R1

**Figure 11: Computing C3: Step 1 (left), Step 2 (Legal) (center), Step 3 (right)**

IDs
R1
R2
R3
E
Z
R1 E
R2 E
E R1
Z R1
Z Z
R1 E R1
Z Z R1

IDs
R1 E R1
Z Z R1

IDs
R1 E R1
Z Z R1

**Figure 12: Table F: Initial (left); Accepted by automaton (center); After the ToI scan (right)**

and *WRT* algorithms this step corresponds to the generation of  $C_k$  using  $C_{k-1}$ . However, adopting the *Valid* variation would require using F and the automaton. Steps 12-14 analyze each of the candidate sequences  $c_i \in C_k$ . The idea consists in detecting *all the paths* in the automaton, which in fact represent sequences of conditions satisfied by  $c_i$ . To do this we need the information stored in the automaton and the category instances. If  $c_i$  does not verify any path of length k, it is pruned. Steps 16-18 scan the ToI instance to compute support. If we are using *early evaluation* for the temporal attributes, here is where the ToI is used to validate temporal conditions. For each OID we calculate the  $c_i \in C_k$  supported by consecutive sequences. Given that we are introducing semantic information into the mining process and we consider *equivalent sequences* to be exchangeable, we add all the sequences  $c_i$  supported by at least one OID to the set F, and prune the ones not supported by any OID. While the support of the set F is equal or greater than the minimum, the algorithm continues. In the final phase (Step 24-25), the algorithm repeats the sequence performed inside the loop, but using C instead of C'.

Either using early or late evaluation, we need to bind a variable to a value. In RE-SPaM, there is no limit on the number of variables that can be defined. Thus, we use a hashing structure to store and check if a variable has already been bound. This structure is built for each of the candidate sequences, i.e. the bindings cannot be shared between sequences. For each item in a candidate sequence we analyze the variables involved. For each variable that appears, it may happen that: (a) it is the first time that this variable is instantiated; thus, the variable and the value are hashed in a structure of variable bindings; (b) the variable has been already bound to a value; thus, the new binding is compared with the previous one. For an efficient support count, we also use hashing structures to store sequences and the OIDs which support them.

**EXAMPLE 12.** For query  $[price = @x]^+$ , suppose we obtain a candidate sequence {R1 R2}. Due to the item identified by R1, @x is stored in a hash table with its corresponding value ('cheap'). Later, analyzing the price associated with the item R2, we obtain the value 'expensive', which does not match the previous one. Thus, the candidate sequence does not verify the expression. □

---

**Algorithm 1** RE-SPaM Algorithm

---

```
01. minSupport := ReadSupport()
02. query := ReadQuery()
03.  $\mathcal{A}_{\mathcal{R}}$  := BuildAutomaton(query)
04.  $C'$  := define C relaxation
05. //Incremental phases
06.  $k := 1$ 
07. REPEAT
08.     // Add sequences of length  $k$  which verify  $C'$  to set  $F$ 
09.     // Populate  $C_k$ 
10.      $C_k := \{c_k \mid c_k \text{ is a candidate sequence of length } k\}$ 
11.     // Update  $C_k$  using  $\mathcal{A}_{\mathcal{R}}$  and category instances
12.      $tmp := \{c_k \mid c_k \text{ is not verified by any path}$ 
13.         of length  $k$  in  $\mathcal{A}_{\mathcal{R}}\}$ 
14.      $C_k := C_k - tmp$ 
15.     // Update  $C_k$  using ToI instances
16.      $tmp := \{c_k \mid c_k \text{ is not verified by any sequence}$ 
17.         of any OID  $\}$ 
18.      $C_k := C_k - tmp$ 
19.     // Update  $F$ 
20.      $F := F \cup C_k$ 
21. UNTIL FinalConditionHolds or  $\text{MinSupport}(F) < \text{minSupport}$ 
22. // Final phase
23. // Eliminate from  $F$  sequences that do not satisfy constraint  $C$ 
24.  $F := \text{VerifyOriginalConstraint}(F, \text{minSupport})$ 
25. ListSequences( $F$ )
```

---

*Final considerations: concise expressions.* A strong point of RE-SPaM is that, when a disjunction is present in an expression, we do not need to enumerate all possible alternatives, like existing approaches require. For a large number of items, this would be hard to handle. For instance, if instead of three restaurants we had a more realistic number, a query asking for a disjunction of all of them would result in an extremely long expression. RE-SPaM allows to write equivalent queries in a more elegant and concise way. Let us consider, for instance, the expression:  $([ID='Z'] \mid [ID='E'])$ . We have shown that the RE-SPaM algorithm populates the  $C_k$ 's with the IDs of the items that verify the expression. In this case,  $C_1$  would be populated with E and Z. RE-SPaM allows other ways of writing concise sub-expressions, that result in the same set  $C_1$ . Suppose that instead of considering the Eiffel tower and the Zoo as two different categories, we would have considered them two occurrences of, say, the "attractions" category. A query equivalent to the above one would have been:  $[categoryName='attractions']$ . During query evaluation,  $C_1$  would be populated with E and Z.

As another option, we could chose to define two different category names (e.g., Eiffel tower and Zoo), but a common attribute, say, *Family*. Further, the user could have organized the places of interest into a hierarchy (e.g., an OLAP dimension) named *PoI*, such that the level *categoryName* rolls up to the level *family*. For example, the expression  $[rollup(categoryName, 'family', 'PoI') = 'tourist attraction']$  returns *true* when *categoryName* is instantiated with the Zoo or the Eiffel tower. In this case, the query would read:  $[rollup(categoryName, 'family', 'POI') = 'tourist attraction']$ . Again, during query evaluation,  $C_1$  is populated with E and Z.

## 4.4 Complexity

We now briefly analyze the complexity of Algorithm 1. Each step of the algorithm is composed of three phases. Each phase generates candidate sequences of the same length (i.e., at each step  $k$ , candidate sequences of length  $k$  are computed in three phases). The elements in  $C_k$  (line 10, first phase) are computed using the sequences in the set  $C_{k-1}$ . We compute  $C_k$  by means of a self-join between the sequences in  $C_{k-1}$ , and discard the ones such that

their suffixes and prefixes of length  $k - 2$  do not match. For example, when joining the sequences 'ABC' and 'BCD', we generate 'ABCD' (the suffix and prefix 'BC' is the same for both sequences). In the worst case, this operation has complexity  $\mathcal{O}(|C_{k-1}|^2)$ , where  $|C_{k-1}|$  is the number of candidate sequences in  $C_{k-1}$ .

The second phase (see lines 12, 13 and 14) consists in using  $C_k$  and pruning it with the automaton generated from the constraints used in the query (i.e., the automaton is used to verify if a candidate sequence of IDs in  $C_k$  satisfies some path of length  $k$ ). If  $|\mathcal{A}_{\mathcal{R}}|$  is the number of states of the automaton, then, in the worst case the algorithm performs  $|\mathcal{A}_{\mathcal{R}}|^{(k+1)}$  comparisons to detect if a  $k$ -sequence of IDs in  $C_k$  satisfies an expression in a path of length  $k$  in the automaton. Since  $|\mathcal{A}_{\mathcal{R}}|$  is the number of nodes of the graph representing the automaton, the number of paths of length  $k$  could be at most  $|\mathcal{A}_{\mathcal{R}}|^{(k+1)}$ , because the automaton accepts loops. Checking which sequences in set  $C_k$  (generated in the previous phase) satisfy the constraints in the automaton, takes, in the worst case  $\mathcal{O}(|C_k| * |\mathcal{A}_{\mathcal{R}}|^{(k+1)})$ . All these sequences of candidate  $C_k$  are organized in a hash table, for the following phase.

The last phase of step  $k$  consists in pruning using the ToI, thus, accessing the database (lines 16, 17 and 18). A database scan must be performed in order to build contiguous lists (see Definition 14) of length  $k$ . Let us denote  $|Items|$  the number of items in the ToI. In the worst case, all these items belong to the same object and the number of consecutive lists (each one of these lists is composed of IDs.) of length  $k$  that are generated is given by  $|Items|-k+1$ . This is an upper bound. If these items are distributed among different objects (transactions) the number of lists becomes considerably smaller. The candidate sequences in  $C_k$  are organized in a hash table; thus, checking if a list of IDs belonging to the ToI matches a list of IDs in  $C_k$  could be done in  $\mathcal{O}(1)$ . Depending on the query, this phase requires more than a simple matching between IDs in the list and IDs in some element in  $C_k$ . Recall that we do not evaluate only *coincidence* here. For example, if a sequence in  $C_k$  is 'ABA' and there is a list 'ABA' in the database, we may initially think that this sequence satisfies the query. However, let us consider the  $[ts = @x \wedge price = 'cheap']$ .  $[tf = @x \wedge food = 'Italian']$ . Although 'AB' is a candidate sequence in  $C_2$  (because A verifies the price='cheap' and B verifies food='Italian'), we must postpone the evaluation of  $ts$  and  $tf$  to the database scan stage. We cannot do this with the automaton because *temporal* attributes are not part of category occurrences. Thus, the database scan is not only for evaluating support. Then, in the worst case, this phase can be done in  $\mathcal{O}(|Items| - k + 1)$ . These three phases are performed repeatedly until no more candidate sequences are generated or the candidate sequences in  $C_k$  are all pruned accessing the ToI, because there does not exist any useful list in the database of length  $k$ .

The last part of the algorithm (line 24) is analogous to the former ones, except for the first phase (generation of  $C_k$ ). Phases two and three use all the candidate sequences not pruned in previous steps, to detect if they are recognized by the automaton and have the required support.

## 5. EXPERIMENTAL RESULTS

We ran our tests on a dedicated IBM 3400x server equipped with a dual-core Intel-Xeon processor, at a clock speed of 1.66 GHz. The total free RAM was 4.0 GB, and there was a 250GB disk drive. The ToI was stored in PostgreSQL 8.2.3. Information about categories has been stored in the database and also in an XML file (both strategies delivered similar results). The rollup functions were implemented on Mondrian<sup>2</sup>, using the MDX query language. Algo-

<sup>2</sup><http://mondrian.sourceforge.net>.

$\mathcal{R}e1$	[price=@x].[price=@x]
$\mathcal{R}e2$	[price=@x].([ID='E'   [ID='Z'])*.[price=@x]
$\mathcal{R}e3$	[price=@x].([ID='E'   [ID='Z']].[price=@x]
$\mathcal{R}e4$	[price=@x].([ID='Z'   [ID='E']].[typeOfFood='French' ^ price=@x]
$\mathcal{R}e5$	[ts_date='10/10/2007' ^ price=@x].([ID='Z'   [ID='E']].[ts_date='10/10/2007' ^ typeOfFood='French' ^ price=@x]
$\mathcal{R}e6$	[ts_date=@d ^ price=@x].([ID='Z'   [ID='E']].[ts_date=@d ^ typeOfFood='French' ^ price=@x]
$\mathcal{R}e7$	[ID='E']+.[typeOfFood='French' ^ rollup(ts_date, 'quarter', 'date dimension')='Q2']

Figure 13: Queries

rithms have been implemented in Java 1.6.

We used real moving object data of 1.9 million of records of the form  $(O_{id}, x, y, t)$ , collected by GPS sensors at intervals of one second. We worked with trajectories of 6276 different moving objects (OIDs). The PoIs correspond to the category instance discussed in Section 1, with additional attributes. For example, we included *parking* and *speciality* for restaurants, and *parking* and *pets* for hotels. These PoIs were created for the experimental evaluation, and do not correspond to a real world situation. There are 17 occurrences of category instances, 5 hotels, 10 restaurants, a zoo and the Eiffel tower. We have rewritten each trajectory as a *sequence of stops*, and this information was stored in the ToI<sup>3</sup>. Each OID has associated items that represent the visited PoIs. The minimum, average and maximum of number of items per transaction (per OID) was 2, 17 and 363 respectively. In short, in our database there exist moving objects that have only passed through two places and other ones that have passed through 363 places. The queries used in our experiments are shown in Figure 13. They have been executed three times, and we report the average execution time.

## 5.1 Experiments

We explained in Section 4 that different algorithms could be used to prune the sequences which do not match a path in  $\mathcal{A}_{\mathcal{R}}$ . To present the algorithm we assumed the variant denoted *Legal* [6, 7]. We also implemented the *valid WRT* alternative. In a nutshell, WRT obtains a family of automaton by moving the initial state to any other state in the automaton, but leaving the final state untouched. The algorithm prunes the sequences not recognized by any of the automata in this family. On the other hand, the reader may have noticed that the *Legal* approach is equivalent to consider all states in the automaton as starting and final. We also analyzed different strategies for processing expressions containing variables: early and postponed evaluation. Thus, four variants were tested: legal-early, legal-postponed, WRT-early, and WRT-postponed.

## 5.2 Discussion of results

Figure 14 compares the performance of the four algorithm variations. Each query was run with different minimum support: 20%, 40%, 60%, 80% and 100%. The average time of each one is reported. For  $\mathcal{R}e1$  and  $\mathcal{R}e7$ , the performance of all four algorithms is similar. This is because the expressions contain no variables (the first case) or two variables (the second case, where variable comparison is used in the early computation of C2). Differences appear when we use variables in paths of at least length three. In all of them, the “early evaluation” outperforms the “postponed evaluation”. Thus, the combination of *Legal-Early evaluation* has better performance than *Legal-Postponed evaluation*. Analogously, *WRT-Early evaluation* has better performance than *WRT-Postponed evaluation*. In general, *WRT-Early* delivered the best performance, especially when expressions involve many variables or long paths in

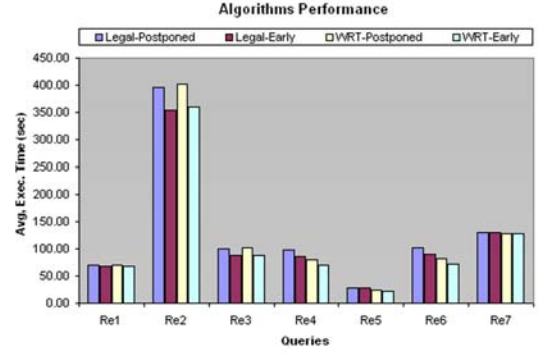


Figure 14: Performance

Min. Sup.	$\mathcal{R}e1$	$\mathcal{R}e2$	$\mathcal{R}e3$	$\mathcal{R}e4$	$\mathcal{R}e5$	$\mathcal{R}e6$	$\mathcal{R}e7$
20%	38	75	16	10	0	7	9
40%	38	75	16	10	0	7	0
60%	38	75	16	0	0	0	0
80%	38	75	0	0	0	0	0
100%	38	75	0	0	0	0	0

Figure 15: Number of sequences obtained

the automaton. Finally, the queries involving loops (+ or \* meta-symbols) are the most expensive ones because the intermediate sets  $C_k$ 's must be expanded through many iterations.

Figure 15 shows the number of sequences obtained for each query. These numbers are coherent with the restrictions that queries  $\mathcal{R}e1$  to  $\mathcal{R}e7$  incrementally add. For instance, the set of sequences obtained by  $\mathcal{R}e2$  includes the set of sequences obtained by  $\mathcal{R}e1$ , because  $\mathcal{R}e2$  adds an intermediate optional sub-path. In  $\mathcal{R}e3$ , the intermediate sub-path is not optional. In this case, the number of sequences changes as the minimum support changes. Note the use of variables in these three queries.  $\mathcal{R}e1$  produces only sequences of length 2,  $\mathcal{R}e3$  only sequences of length 3 and  $\mathcal{R}e2$ , due to the optional sub-path, produces sequences of length of at least two. Query  $\mathcal{R}e4$  is similar to  $\mathcal{R}e3$ , but its last subexpression is more restrictive, as the last item must be not only a place with *price* attribute, but also one serving French food. This is why  $\mathcal{R}e4$  produces less sequences than  $\mathcal{R}e3$ , and no sequence for a support of 60%. Query  $\mathcal{R}e5$  adds conditions on the date of the events. It is reasonable, then, to obtain less sequences than in  $\mathcal{R}e4$  (indeed, no sequences are obtained for  $\mathcal{R}e5$ ).  $\mathcal{R}e6$  replaces the specific date in  $\mathcal{R}e5$  with a variable. Further, we are interested in finding sequences where all their item occurrences correspond to the same day. Finally,  $\mathcal{R}e7$  is a query with no variables, but that includes a rollup function over *ts\_date*.

Figure 15 shows that queries  $\mathcal{R}e1$  and  $\mathcal{R}e2$  produce 38 and 75 sequences, respectively. However, the performance of the latter is approximately eight times slower. Figures 16 and 17 show the phases executed for solving  $\mathcal{R}e2$  and  $\mathcal{R}e1$ , respectively. The first column shows the number of candidate sequences before automaton pruning, the second column the number of candidate sequences *after* automaton pruning, and the third one, the number of sequences after pruning using the ToI instance. We can see that the number of iterations when solving  $\mathcal{R}e2$ , is larger than in  $\mathcal{R}e1$ .

Figures 18 and 19 show comparisons between the average execution time of the queries against the number of sequences obtained, for supports 20% and 60%. We omit the graphics for other supports for the sake of space. As expected, the execution time increases as the number of sequences obtained also increases. In all cases, the curves for the four combinations behave similarly.

<sup>3</sup> The datasets and a demo can be found at <http://piet.exp.dc.uba.ar/mo-patterns>.

Phase	Population	Pruning by Automaton	Pruning By Tol
C1	17	11	11
C2	187	73	38
C3	646	0	0

Figure 16: Solving  $\mathcal{Re}1$ : sizes at each phase

Phase	Population	Pruning by Automaton	Pruning By Tol
C1	17	11	11
C2	187	87	46
C3	782	109	23
C4	391	73	13
C5	221	41	7
C6	119	18	4
C7	68	9	2
C8	34	9	2
C9	34	9	1
C10	17	0	0

Figure 17: Solving  $\mathcal{Re}2$ : sizes at each phase

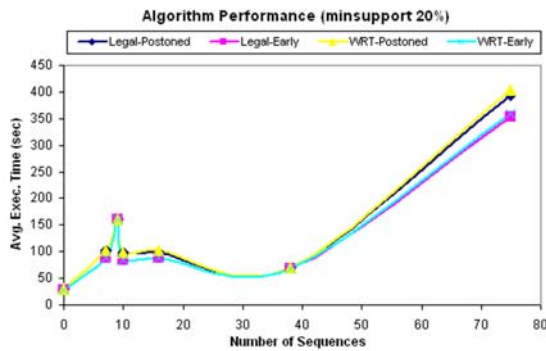


Figure 18: Performance Minimum Support 20%

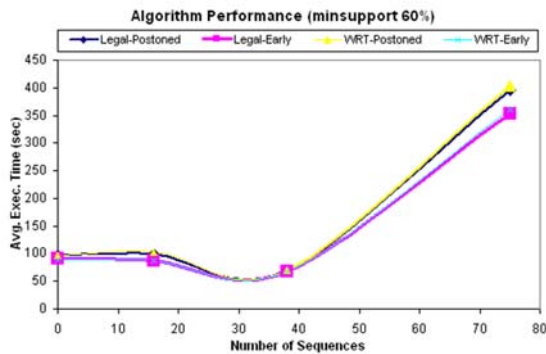


Figure 19: Performance Minimum Support 60%

## 6. FUTURE WORK

We believe that there is still room for improving the language we proposed in this paper. Thus, we are planning to extend RE-SPaM, in order to support more complex kinds of constraints. For instance, we would like to express conditions like “the second stop in a trajectory is a hotel with more stars than the first stop”, and to allow conditions stating that the difference between the initial and ending instants of the trajectory, less than 5 days have passed.

**Acknowledgments.** This research has been partially funded by the Research Foundation Flanders (FWO- Vlaanderen), Research Project G.0344.05, the European Union under the FP6-IST-FET

programme, Project n. FP6-14915, GeoPKDD: Geographic Privacy-Aware Knowledge Discovery and Delivery, and the Argentina Scientific Agency, project PICT 2004 11-21.350.

## 7. REFERENCES

- [1] R. Agrawal, T. Imielinski, and A. Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD Int'l Conference on Management of Data*, 1993.
- [2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th Int'l Conference on Very Large Databases*, 1994.
- [3] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of the Int'l Conference on Data Engineering (ICDE)*, 1995.
- [4] L. Cabibbo and R. Torlone. Querying multidimensional databases. In *Proceedings DBPL'97*, pages 253–269, East Park, Colorado, USA, 1997.
- [5] C. du Mouza and P. Rigaux. Mobility patterns. In *Proceedings of the STDBM'04*, pages 1 – 8, Toronto, Canada, 2004.
- [6] M. N. Garofalakis, R. Rastogi, and K. Shim. Spirit: Sequential pattern mining with regular expression constraints. In *Proceedings of the 25th VLDB Conference*, 1999.
- [7] M. N. Garofalakis, R. Rastogi, and K. Shim. Mining sequential patterns with regular expression constraints. In *IEEE Transactions on Knowledge and Data Engineering*, 2002.
- [8] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *ACM Trans. Database Syst.*, 25(1):1–42, 2000.
- [9] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M. Hsu. Freespan: frequent pattern-projected sequential pattern mining. In *KDD '00: Proceedings of the sixth ACM SIGKDD Int'l conference on Knowledge discovery and data mining*, pages 355–359, 2000.
- [10] H. Mannila, H. Toivonen, and I. Verkamo. Discovering frequent episodes in sequences. In *Proceedings of the First Int'l Conference on Knowledge Discovery and Data Mining (KDD'95)*, pages 210–215, 1995.
- [11] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *Proceedings of the 17th Int'l Conference on Data Engineering*, pages 215–224, Washington, DC, USA, 2001.
- [12] J. Pei, J. Han, and W. Wang. Constraint-based sequential pattern mining: the pattern-growth methods. *J. Intell. Inf. Syst.*, 28(2):133–160, 2007.
- [13] S. Spaccapietra, C. Parent, M. L. Damiani, J. A. Fernandes de Macedo, F. Porto, and C. Vangenot. A conceptual view of trajectories. In *Technical Report*, 2007.
- [14] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT)*, 1996.
- [15] X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large databases. In *SDM*, 2003.