On-line Aggregation and Filtering of Pattern-based Queries

Cédric du Mouza
LAMSADE
Univ. Paris-Dauphine
Paris, France
cedric.dumouza@dauphine.fr

Philippe Rigaux
LAMSADE
Univ. Paris-Dauphine
Paris, France
philippe.rigaux@dauphine.fr

Michel Scholl
Lab. CEDRIC
CNAM
Paris, France
scholl@cnam.fr

Abstract

We consider an environment where a subscription system continuously evaluates pattern-based requests over unbounded sequential data. We propose an extension of the traditional pattern-matching techniques for efficiently handling large sets of such continuous queries. This extension relies on the introduction of variables *in patterns in order to augment their expressivity.*

Based on this extended class of parameterized queries*, our main contributions are threefold. First, we define a refinement relation based on variable relaxation. Second, we use the semi-lattice structure of the set of parameterized patterns for patterns aggregation and filtering. We propose an on-line pattern aggregation algorithm so as to both reduce the cost of pattern-matching evaluation as well as to filter out sequences that cannot match any of the patterns in a subscription cluster. Finally we show, through analysis and experiments, that our technique reduces quite effectively the cost of the matching process.*

1 Introduction

The management of continuous queries over data streams has received much attention in the recent past [18, 11, 20]. It is motivated by the growing amount of data that needs not to be stored in a classical, disk-based DBMS, because it rather represents some constant evolution of data values for some domain of interest. Most cited areas include traffic management, data feeds from sensor networks, emails messages or Web documents, etc. Since such data streams can be seen as unbounded sequential strings, queries naturally tend to take the form of *patterns*, and the query evaluation process attempts to match these patterns with the incoming sequence. We adopt the following classical assumptions: (i) queries – sometimes called *subscriptions* – run continuously and deliver a notification as soon as a matching occurs and (ii) the evaluation is done in one pass, i.e., one cannot move backward on the stream. The challenge, in this context, is to process a large number of queries such that these constraints are satisfied. As a concrete example, this framework covers the continuous monitoring of a flows of emails coming from some Internet area, searching for a (possibly very large) set of fragments described by patterns.

The subscription language considered in the present paper is an extension of traditional patterns with variables, as proposed in [10]. Basically, such *parameterized patterns* are sequences mixing constant symbols and variables, the latter acting as "placeholders" which can be replaced by any constant. During a matching attempt at a given position of the input stream, a pattern variable is bound to the value of its corresponding symbol on the string and keeps this constant value until the attempt fails or succeeds.

The language is more expressive and concise than standard string matching. In [10] we showed that this expressiveness comes without an important computational overhead by describing an extension of the classical pattern-matching algorithm of Knuth, Moris and Pratt [14] which runs in linear time. This extended algorithm, denoted KMP_{var}, is used as a building block in what follows. The present paper considers specifically the context of scalable data streams monitoring, and makes the following new contributions:

1. we define a refinement relation over the set of parameterized patterns, and provide simple algorithms for testing whether a pattern refines another, and for computing the least upper bound (lub) of a group of patterns;
2. we propose a cost model and an incremental aggregation algorithm in order to maintain a structure of the set of subscriptions that minimizes the continuous evaluation cost;
3. finally we provide a set of experimental results which show the gain of the technique.

The main idea behind our work is to exploit the refinement between patterns so as to avoid useless computations. Basically, if a pattern P_2 refines a pattern P_1, and a matching attempt with P_1 fails, there is no need to carry out the evaluation at the same position for P_2. This can be generalized to a group of patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$. If we can find an aggregate pattern P such that any sequence that matches some P_i also matches P, then one should always make an initial attempt to match P against the stream. If this initial attempt fails, this filters out any further computation for the patterns in $\mathcal{P}$.

The rest of the paper presents first (Section 2) an overview of the approach, based on informal examples. We provide the model, including the formalization of the refinement relation and the decision algorithms, in Section 3. Section 4 gives the clustering algorithm and Section 5 describes our experimental setting and results. Related work is presented in Section 6 and Section 7 concludes the paper. Due to space restriction, the proofs and some algorithms are omitted and can be found in the long version available at *http://www.lamsade.dauphine.fr/∼dumouza/ssdbm06_long.pdf.*

2 Approach overview

This section provides a presentation of the paper's main ideas. It covers successively the role of patterns, the refinement relation and presents the intuition behind our algorithms.

2.1 Parameterized patterns

A pattern is a sequence made either of constant symbols, denoted `a`, `b`, `c`, or variables denoted `@x`, `@y`, `@z`. For clarity we separate the components of a pattern with the '.' concatenation operator. A sequence S *matches* a pattern P at some position l if there exists some mapping between P and a subsequence of S starting at l. Some examples follow:

1. `a.b.c` is a variable-free pattern which (as expected) is matched by a sequence containing the substring `a.b.c`;

2. `a.@x.c` is a pattern which is matched by any sequence containing a substring of size 3, beginning with `a` and ending with `c`;

3. the pattern `a.@x.b.@x` is matched by any subsequence of four symbols such that `a` and `b` are respectively in first and third position, while a third symbol, whatever its value, can be found both in positions 2 and 4;

Clearly parameterized patterns extend the expressivity of variable-free sequential patterns. They also allow to express concisely some search operations which would otherwise require long and tedious regular expressions. From a computational point of view, parameterized patterns constitute a convenient trade-off because they preserve a low space and time complexity (this would not be the case with regular expressions extended with variables, see [9]).

In [10], we proposed a matching algorithm which checks each symbol of the input sequence only once and requires a memory space proportional to the number of variables in the pattern. Our technique, denoted KMP$_{var}$, is essentially an extension of the classical pattern-matching algorithm of Knuth, Morris and Pratt [14], denoted KMP. When a KMP$_{var}$ automaton runs over a sequence S, the number of comparisons needed to find the matching subsequences is linear in the number of symbols read. This constitutes a quite effective tool which is used as a building block in our following multi-patterns evaluation framework.

2.2 Patterns refinement

A pattern P_2 is a *refinement* of a pattern P_1 if any sequence matching P_2 at position l matches also P_1 at l. Syntactically, refining a pattern can be seen as tighter matching constraints. There exists three possible ways of doing so: by simply adding some constant symbols to a pattern's suffix, by replacing a variable with a constant symbol, and finally by linking the variables. Some examples follow.

$P_1 = b.@x.a$ is refined by the pattern $P_2 = b.@x.a.c$, since, obviously, any sequence that matches P_2 at some position l also matches P_1. Generally, if P_1 is a prefix of P_2, then P_2 is a refinement of P_1. This is a trivial case which holds for patterns with or without variables.

The presence of variables gives rise to some more subtle cases. First, replacing a variable by a constant symbol strongly restricts the possible matchings because a variable captures the whole alphabet of symbols. For instance the pattern $P_0 = @x.@y.a$ is refined by $P_1 = b.@x.a$. The repetition of variables is meaningful because all the occurrences of a same variable must be instantiated to the same value during a matching attempt. Therefore P_1 is *also* refined by $P_3 = b.@x.a.@x$, because the repeated occurrence of $@x$ in P_3 constitutes a constraint which does not hold for P_1.

Variable names are not significant, hence patterns are equivalent up to a renaming of variables, e.g., $b.@x.a \equiv b.@y.a$. Note also that adding at the end of a pattern P a variable which does not already appear in P yields an equivalent pattern, i.e., $P_1 = b.@x.a \equiv b.@x.a.@y$. Indeed, since $@y$ can be bound to any symbol, a sequence that matches $b.@x.a$ matches $b.@x.a.@y$ as well, and conversely[1]. The refinement relation holds on equivalent classes of patterns, which are formally defined in the next section.

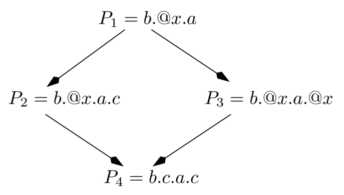

Figure 1. The refinement relation

Finally it is worth mentioning that a pattern can be the refinement of several other patterns. For instance $P_4 = b.c.a.c$ is generalized both by P_2 and P_3 and, by transitivity, by P_1 and P_0. Figure 1 shows the graph of the refinement relation for the set of patterns $\{P_1, P_2, P_3, P_4\}$ given as example above.

We can exploit this relation as follows: if a sequence S does not match the pattern P_1, it cannot match P_2, P_3 or P_4. Therefore, one initially runs a KMP$_{var}$ automaton only against P_1. If, and only if, the matching is successful at a given position l of the sequence, a KMP$_{var}$ evaluation becomes active for both P_2 and P_3. Finally, if this latter evaluation is successful for either P_2 or P_3, the matching with P_4 has to be considered. It is expected that this strategy saves a lot of computations, because matching attempts with P_1 will fail for most of the positions of the sequence S, thereby avoiding many useless comparisons with all the patterns that refine P_1.

2.3 Patterns aggregation and filtering

As suggested by the previous example, the graph of the refinement relation is a directed acyclic graph (actually it is easy to show that the graph is an upper semi-lattice). In principle it would be possible to maintain the transitive reduction of the graph but its construction turns out to be too costly[2]. Moreover this is not necessary since it is sufficient to maintain one and only one path from the root to any node. This guarantees that any pattern can be reached, if needed.

Our goal is thus to construct a spanning tree of the refinement graph such that the average evaluation cost is minimized. Intuitively, the minimization implies choosing the most filtering paths. Let us look at Figure 2, assuming an alphabet with only four symbols a, b, c and d. The pattern set is $\{@x, a.@x.b, @x.c, a.b.c.d\}$. The transitive reduction of the graph is shown at the top of the figure, and there exists two possible spanning trees, shown in the bottom part.

In order to decide which solution minimizes the evaluation cost, we must consider the *filtering rate* of each node. Clearly the pattern $@x$ is matched by any sequence, and its filtering rate is 1. One can estimate that $\frac{1}{4}$ of the matching attempts at a given position l against the pattern $@x.c$ will be successful, because, assuming a uniform distribution, this is the probability of finding the symbol c at position $l + 1$. Similarly, the filtering rate of the pattern $a.@x.b$ can be estimated to be $\frac{1}{4} \times 1 \times \frac{1}{4}$.

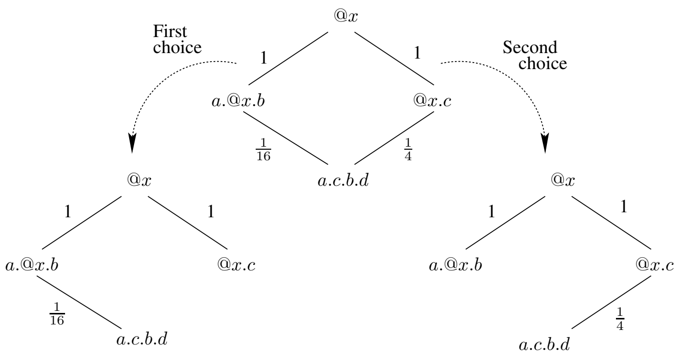

Figure 2. Issues in patterns tree construction

We assign to each out-edge of a node N a *weight* equal to the filtering rate of N. The problem is to find the minimum spanning tree, *i.e.*, the connected subgraph containing all the nodes such that the sum of the edge weights if minimal [2]. From our application point of view, this means that for each pattern P one keeps the path to P which is the most filtering one in the graph of the refinement relation. In the case of Figure 2, the bottom left choice is the best one. Intuitively, only $\frac{1}{16}$ of the matching attempts will have to evaluate the pattern `a.c.b.d`, instead of $\frac{1}{4}$ if the solution of the right part were adopted.

As usual with publish/subscribe applications, we assume that the subscriptions (patterns) may be added or removed dynamically. We must therefore construct and maintain incrementally the structure. However, maintaining incrementally the optimal solution results in a very significant cost (in the worst case, the whole tree must be reconstructed). We propose an algorithm which delivers an approximate solution and show through experiments that it still provides an effective reduction of the overall evaluation cost with respect to the trivial solution that evaluates separately each of the submitted patterns.

2.4 Multi-pattern evaluation

The evaluation mechanism associates one KMP$_{var}$ automaton with each pattern in the pattern tree. These au-

[1]We consider unbounded sequences, thus the length of a pattern is never an issue.

[2]The computation of the transitive reduction runs in $O(n)$ for a graph of size n, for each node insertion [3]. Therefore one might, in the worst case, need to update all the nodes of the graph when a new pattern is received.

tomata may be *active* or *inactive*, according to the following rules:

1. Initially the KMP$_{var}$ automaton $\mathcal{A}_{root}$ associated with the root is active; others automata are inactive.
2. When a matching is found for an automaton $\mathcal{A}_P$, all the inactive automata associated with the sons of P become active; $\mathcal{A}_P$ itself becomes inactive.
3. When a matching fails for some automaton $\mathcal{A}_P$, its parent becomes active and $\mathcal{A}_P$ becomes inactive.

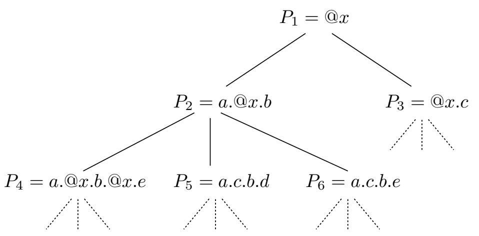

Figure 3. Evaluation over a pattern tree

Consider the tree of Fig. 3 and the sequence $a.c.b.c.a.d.b \cdots$. Each pattern P_i is associated with an automaton $\mathcal{A}_{P_i}$. Initially the automaton $\mathcal{A}_{P_1}$ is the only one active. Figure 4 illustrates the successive activation status.

The first symbol of the sequence matches P_1. Hence $\mathcal{A}_{P_2}$ and $\mathcal{A}_{P_3}$ become active in turn (Figure 4(b)). The next symbol is c: a matching is found with P_3. $\mathcal{A}_{P_3}$ becomes inactive, while the automata of its sons become active (Figure 4(c)). Let us now focus on the subtree rooted at P_2. When the third symbol, b, is read from the sequence, a matching is found. $\mathcal{A}_{P_2}$ becomes inactive, $\mathcal{A}_{P_4}$, $\mathcal{A}_{P_5}$ and $\mathcal{A}_{P_6}$ become active (Figure 4(d)).

When the fourth symbol, c, is received, the matching attempt fails for P_5 and P_6 (Figure 4(e)). The evaluation proceeds then as follows:

- P_4 is matched by the sequence; notifications are sent and the automaton $\mathcal{A}_{P_4}$ remains active;
- $\mathcal{A}_{P_5}$ and $\mathcal{A}_{P_6}$ become inactive but $\mathcal{A}_{P_2}$, their common parent, becomes active.

Finally after receiving $a.d.b$, $\mathcal{A}_{P_2}$ reaches a successful state and activates again $\mathcal{A}_{P_5}$ and $\mathcal{A}_{P_6}$ (Figure 4(f)). In that case a mismatch occurs at once because the second symbol is d whereas both $\mathcal{A}_{P_5}$ and $\mathcal{A}_{P_6}$ expect a c. Actually, when several sibling patterns fail, their common prefix is represented by their parent. Making active the parent is a way to "factorize" the query evaluation for this prefix, but this does not guarantee that any of its children will succeed.

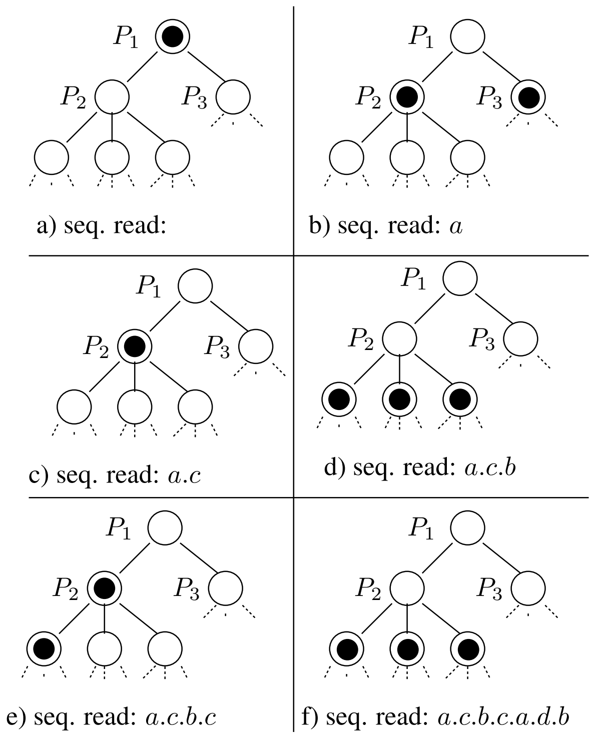

Figure 4. Active patterns during the evaluation of $a.c.b.c.a.d.b$

3 The Model

We first recall some basic definitions from [10], and then provide the semantic and syntactic characterization of the refinement relation.

3.1 Preliminaries

We assume a finite set of constant symbols Σ and a countable set of variables $\mathcal{V}$, with $\Sigma \cap \mathcal{V} = \emptyset$. A *sequence* is a word in Σ^+, a *pattern* is a word in $(\Sigma \cup \mathcal{V})^+$. The interpretation of a pattern P *without variable* is trivial: a sequence S *matches* a pattern P at position l if $P = S[l].S[l+1].\cdots.S[l+|P|-1]$.

Variables are useful to capture more general sequences where symbols are not explicitly assigned to specific symbols. The interpretation of patterns (with variables) is an extension of the subsequence matching semantics previously given: a sequence S matches a pattern P (at position l) if one can substitute each variable in P by a symbol from Σ, such that the resulting pattern is a subsequence of S starting at l. For instance the subsequence $S = c.b.c.e.f$ matches the pattern $P = c.@x.c.@y.f$ at position $l = 0$.

Definition 1 (Valuation). *A* valuation ν *is a finite set of* *the form* $\{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}$ *where* $x_i \in \mathcal{V}, i = 1, \ldots, n$*, and each* t_i *is a symbol in* Σ.

$\nu(P)$ denotes the pattern obtained from P by replacing, for each $x_i/t_i \in \nu$, each occurrence of x_i in P by t_i. If, for instance, $P = \texttt{b.@x.e.@y.f}$ and $\nu = \{\texttt{@x/c, @y/e}\}$, then $\nu(P) = \texttt{b.c.e.e.f}$. In the following $var(P)$ denotes the set of variables in P. If all the variables in P are bound by a valuation ν, then $\nu(P)$ is a word in Σ^+. Hence the definition:

Definition 2 (Interpretation of a pattern). *A sequence S matches a pattern P at position l iff there exists a valuation ν such that $\nu(P)$ is a subsequence of S starting at l.*

Two patterns may be semantically equivalent, yet syntactically different. For instance the patterns $P_1 = @z.a.@x.b.@y.@z$ and $P_2 = @y.a.@z.b.@x.@y$ are equivalent up to a permutation in $\mathcal{V}$. A pattern P is also equivalent to $P.@x$, with $@x \in \mathcal{V}\backslash var(P)$.

We define the *marking* of a pattern P as the pattern where each variable of $\mathcal{V}$ is replaced by $@x$ marked by a subscript over $\mathbb{N}$, representing the order of the variables in P. A pattern P is *normalized* iff the following conditions hold: (i) $P = marking(P)$, and (ii) if P is of the form $P'.@x$, then $@x \in var(P')$.

It is straightforward that for any pattern P, there exists a unique equivalent normalized pattern. For instance the normalized pattern of $@y.c.@x.@y.@z$ is $@x_1.c.@x_2.@x_1$. In the following we shall limit our attention to normalized patterns, seen as delegate of their equivalent class.

3.2 Pattern refinement

A pattern P_2 *refines* a pattern P_1 iff, for each position l such that S matches P_2, S matches P_1 as well. This means first that $|P_1| \leq |P_2|$, and second that there exists two valuations ν_1 and ν_2 such that both $\nu_1(P_1)$ and $\nu_2(P_2)$ are subsequences of S starting at l. Hence the definition.

Definition 3 (Refinement relation on patterns). *A pattern P_2* refines *a pattern P_1, denoted $P_2 \trianglelefteq P_1$, iff for each valuation ν_2, there exists a valuation ν_1 such that $\nu_1(P_1)$ is a prefix of $\nu_2(P_2)$.*

For instance the pattern $P_1 = a.@x_1.b$ is refined by the pattern $P_2 = a.@x_1.b.@x_2.c$, because for any valuation ν_2, $\nu_2(P_2)$ is of the form $\texttt{a}.\nu_2(@x_1).\texttt{b}.\nu_2(@x_2).\texttt{c}$. Any valuation ν_1 satisfying $\nu_1(@x_1) = \nu_2(@x_1)$ is such that $\nu_1(P_1)$ is a prefix of $\nu_2(P_2)$. P_1 is also refined by $P_3 = a.c.b$ because for any ν_2, $\nu_1(P_1) = \nu_2(P_2)$ as soon as $\nu_1(@x_1) = c$.

It follows from the definition that, if $P_2 \trianglelefteq P_1$, as long as one cannot find a valuation such that S matches P_1, there is no hope to find one such that S matches P_2.

There is a simple syntactic characterization of the refinement relation. Given two (normalized) patterns P_1 and P_2, it suffices to carry out a comparison of the patterns' symbols from left to right, and to check that each symbol from P_1 is a "relaxation" of the corresponding symbol of P_2, according to the following rules:

1. a constant symbol can be relaxed to itself or to a variable;
2. a variable can be relaxed to another variable whose subscript is greater or equal.

Consider again $P_1 = a.@x_1.b$ and $P_2 = a.c.b$ Then, $\texttt{a}$ can be relaxed to $\texttt{a}$, $\texttt{c}$ to $@x_1$ and $\texttt{b}$ to $\texttt{b}$: $P_2 \trianglelefteq P_1$. The second condition ensures that no problems comes from the multiple occurrences of a same variable. Take for instance $P_1 = a.@x_1.b.@x_1$ and $P_2 = a.@x_1.b.@x_2$. The variable $@x_2$ in P_2 cannot be relaxed to the second occurrence of $@x_1$ in P_1, because this would break the restriction on the subscripts monotonicity. However the second occurrence of $@x_1$ in P_1 can be relaxed to $@x_2$. Indeed, in that case, $P_1 \trianglelefteq P_2$. The algorithm CONTAIN, given in the long version, determines whether $P_2 \trianglelefteq P_1$ in linear time.

It is easily verified that $\trianglelefteq$ is a partial order on $(\Sigma \cup \mathcal{V})^*$. Furthermore, it can be shown that two patterns have a unique minimal common ancestor with respect to relation $\trianglelefteq$.

Proposition 1. *The set of patterns over $(\Sigma \cup \mathcal{V})$ ordered by $\trianglelefteq$ is an upper semi-lattice.*

The following algorithm computes the least upper bound (lub) of two patterns. Function NEWVAR returns a variable name that has not yet been used. SUBST(d_1, d_2, r) is a set of substitutions. A triplet (d_1, d_2, r) associates a pair of symbols (d_1, d_2) from, respectively, P_1 and P_2, with a symbol r in the lub.

```
LUB($P_1[0 \ldots n_1], P_2[0 \ldots n_2]$)
Input: two patterns $P_1$ and $P_2$
Output: the lub of $P_1$ and $P_2$
begin
    SUBST := $\emptyset$
    $n := min(n_1, n_2)$ // the length of the lub is $\leq$ to $|P_1|$ and $|P_2|$
    for ($k := 0$ to $n$) do
        if ($\exists x, P_1[k], P_2[k], x) \in$ SUBST) then
            $lub[k] := x$
        else
            if (($P_1[k], P_2[k]) \in \Sigma^2$ and $P_1[k] = P_2[k]$) then
                $x := P_1[k]$
            else
                $x :=$ NEWVAR()
            endif
            SUBST := SUBST $\cup \{(P_1[k], P_2[k], x)\}$
        endif
        $lub[k] := x$
    endfor
    return NORMALIZE($lub$)
end
```

The final normalization is necessary to remove the useless variables which may appear in the suffix.

4 Evaluation

We propose now an evaluation strategy that takes advantage of the refinement relation when a large number of patterns must be evaluated simultaneaously over an input sequence. Recall that we do not need to represent the whole graph of the refinement relation. Instead our goal is to construct a spanning tree that minimizes the overall evaluation cost. Second, patterns can be added or removed dynamically, from the set of subscriptions. We must therefore design an incremental algorithm to maintaining the tree.

A last issue relates to how and when we can cluster patterns which are close from one another, and whether we must materialize the lub of this clusters and put it in the tree. Note that by doing so, we introduce some "artificial" nodes which have not been subscribed by any user. This is justified because a lub, if it is close enough from the patterns that it covers, is a mean to factorize the computations that would otherwise be carried out independentlty. The simple cost model given below shows that adding lubs in the patterns tree it is almost always beneficial.

4.1 Cost model

In order to estimate the gain obtained when using the lub of a set of patterns as a filter we first define the *filtering rate* of a pattern. It estimates the percentage of comparisons that lead to a successful matching during pattern evaluation. We assume a uniform distribution of symbols from Σ in a sequence (our model does not depend on this assumption, but a more precise distribution law is application-dependent). Each constant symbol can be found with probability $\frac{1}{|\Sigma|}$. As far as variables are concerned, the first occurrence of a variable has no impact on the filtering rate because any symbol will match. All the other occurrences must be bound to the same value in Σ. In summary the filtering rate is estimated by the following inductive formula:

- if $P = \alpha$, then $\tau(P) = \begin{cases} 1\, if\ \alpha \in \mathcal{V} \\ \frac{1}{|\Sigma|}\, if\ \alpha \in \Sigma \end{cases}$
- if $P = P'.\alpha$, then $\tau(P) = \begin{cases} \tau(P') \times \frac{1}{|\Sigma|}\, if\ \alpha \in (\mathcal{V} \cap var(P)) \cup \Sigma \\ \tau(P')\, if\ \alpha \in \mathcal{V} \backslash var(P') \end{cases}$

If P is a pattern of length n with k *distinct* variables, its filtering rate is therefore estimated to be

$$\tau(P) = \frac{1}{|\Sigma|^{n-k}}$$

We can now measure the gain of adding the lub for a subset of patterns in the tree. Let $P_1, \ldots, P_m$ be a set of patterns and let $P = lub(P_1, \ldots, P_m)$, with $n = |P|$. The number of comparisons when testing n symbols of the sequence can be estimated as:

$$Cost_{cl} = (1 - \tau(P)) \times n + \tau(P) \times (n + m \times n) \quad (1)$$

This cost is an upper bound: when reading n symbols from the sequence, there is a probability $\tau(P)$ that the sequence matches the pattern, in which case we must also scan its children. In the worst case, all the n symbols have to be tested for each child, hence $m \times n$ comparisons. The cost of an independent evaluation of each pattern (without using P as a filter), is

$$Cost_{multi} = m \times n \quad (2)$$

The difference between the two solutions depends both on the filtering rate (the smaller the better), and on the number of children (the higher the better). Using the lub as a filter saves comparisons when $Cost_{cl} < Cost_{multi}$, $i.e.$, when

$$\tau(P) < \frac{m-1}{m} \Rightarrow k < n - \frac{\log(m) - \log(m-1)}{\log|\Sigma|} \quad (3)$$

where k denotes the number of free variables in P.

Let us apply the cost model to a DNA application. The size of the vocabulary is 4. Assume we have two patterns P_1 and P_2 whose lub is P_{lub}, with $|P_{lub}| = 8$. We then save comparisons if $\tau(P_{lub}) < 1/2$. Given that $\tau(P) = \frac{1}{|\Sigma|^{n-k}}$, there is a gain if the number of free variables in $|P_{lub}|$ is $k < 8 - \frac{\log 2 - \log 1}{\log 4} = 8 - \frac{1}{2}$. So the presence of only one constant symbol is sufficient to obtain a (statistical) gain.

The formula (3) is sufficient to conclude that in practice the evaluation cost is reduced by using the lub of two patterns for any alphabet whose size is greater than 2, as soon as the lub contains at least one constant symbol (or, equivalently, a variable with two occurrences). The construction of the patterns tree, presented below, systematically attempts to add new lubs in the tree during the insertion of a new pattern.

4.2 The pattern tree

The algorithm constructs incrementally a tree T of patterns. The insertion of a new pattern P is performed in two steps:

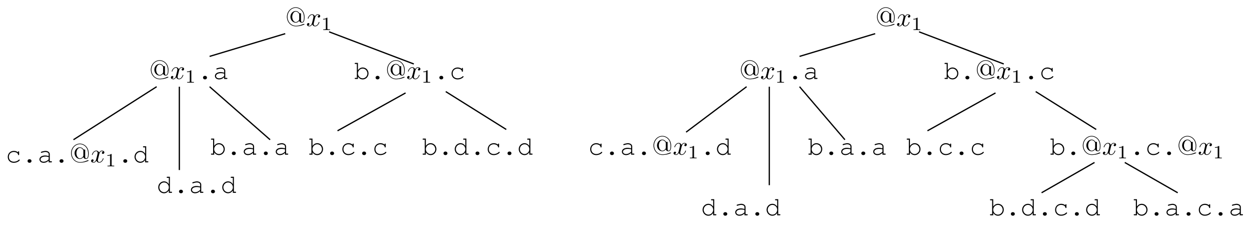

Figure 5. Insertion of the pattern `b.c.a`

- **Candidate parent selection**.
  A node N in T is a *candidate parent* for P if the following conditions hold: (i) $P \lhd N$, and (ii) for each child N' of N, $P \ntriangleleft N'$, i.e., P is strictly contained in N but does not contain any child of N.

  The algorithm performs a depth-first search to seek for a candidate parent. Starting from the root, one chooses at each level the most selective child which contains P. When such a child no longer exists the candidate parent is found. Note that this is an heuristic which avoids to follow an unbounded number of paths in the tree, but does not guarantee that the "best" candidate parent (i.e., the most selective one) is found.

- **Lub selection**.
  Once the candidate parent N is found, the second step inserts P as a child or grandchild of N as follows. First, for each child N' of N, one computes $lub(P, N')$ and keeps only those lubs which are strictly contained in N. Now:

  1. if at least one such lub $L = lub(P, N')$ has been found, the most selective one is chosen, and a new subtree $L(P, N')$ is inserted under N;
  2. else P is inserted as a child of N.

Let us take as an example the left tree of Figure 5. The pattern $P = b.a.c.a$ must be inserted. First one checks P against the root node. Since $P \lhd @x_1$, we consider the two children of the root. P is strictly contained in both $@x_1.a$ and $b.@x_1.c$ which constitute therefore two possible paths. Since $\tau(@x_1.a) = \frac{1}{|\Sigma|}$ and $\tau(b.@x_1.c) = \frac{1}{|\Sigma|^2}$, the second is chosen. None of its children contain P, therefore $b.@x_1.c$ is the candidate parent.

Next one determines the lubs of P with each child of $b.@x_1.c$, and keeps those which are strictly contained in $b.@x_1.c$. For instance $lub(b.a.c.a, b.c.c) = b.@x_1.c$, so it is not kept. However $lub(b.a.c.a, b.d.c.d) = b.@x_1.c.@x_1$ is a candidate lub. One finally obtains the tree of the right part of Figure 5.

It may happen that several equivalent choices are possible. Assume for instance that the pattern $P' = b.a.d$ is inserted. The candidate parent is $@x_1.a$, and one must choose between $L_1 = lub(P', d.a.d) = @x_1.a.d$ and $L_2 = lub(P', b.a.a) = b.a$. Both have the same filtering rate. The tie-breaking procedure compares the prefixes of length $l < min(|L_1|, |L_2|)$, starting with $l = 1$. As soon as one of the prefixes is found to be more selective than the other one, the corresponding lub is chosen. The rationale is that a mismatch will occur more quickly, and that less comparisons are necessary. In our example, since $@x_1$ is a less filtering prefix than b we create the lub $lub(P', b.a.a) = b.a.@x_1$.

The insertion algorithm is summarized below.

```
INSERT($P, T$)
Input: a pattern $P$ and a pattern tree $T$
Output: the new tree after the insertion
begin
    // First search the candidate parent using a depth-first search
    $parent$ := CANDIDATEPARENT($P, T.root$)
    // Second step: computes the candidate lubs
    $\mathcal{S} := \emptyset$
    for each $P_j \in children(N_i)$ do
            if LUB($P, P_j$) $\lhd N_i$ then $\mathcal{S} = \mathcal{S} \cup \{P_j\}$
    endfor
    if $\mathcal{S} = \emptyset$ then // No candidate lub. $P$ is a child of $parent$
        $parent.children := parent.children \cup \{P\}$
    else // Add the lub of the two patterns
        Choose a node $N$ in $\mathcal{S}$
        Add the subtree LUB($N, P$)($N, P$) to $parent$;
        Remove $N$ from $parent$'s children;
    endif
    return $T$
end
```

The following special cases are not represented in the algorithm: (i) a pattern P is already represented in T and (ii) the lub of P and a node N is P itself. The extension is straightforward.

The insertion algorithm follows a single path from the root to a node in the pattern tree. At each level, a comparison must be carried out with each children. Its complexity is determined by the following properties.

Proposition 2. *The depth of a pattern tree T is bound by $l + 1$ where l denotes the size of the longest pattern in T. Moreover each internal node N of the tree has at most $|\Sigma| \times (l - k)$ children, where k is the number of constant symbols or variable repetitions in N.*

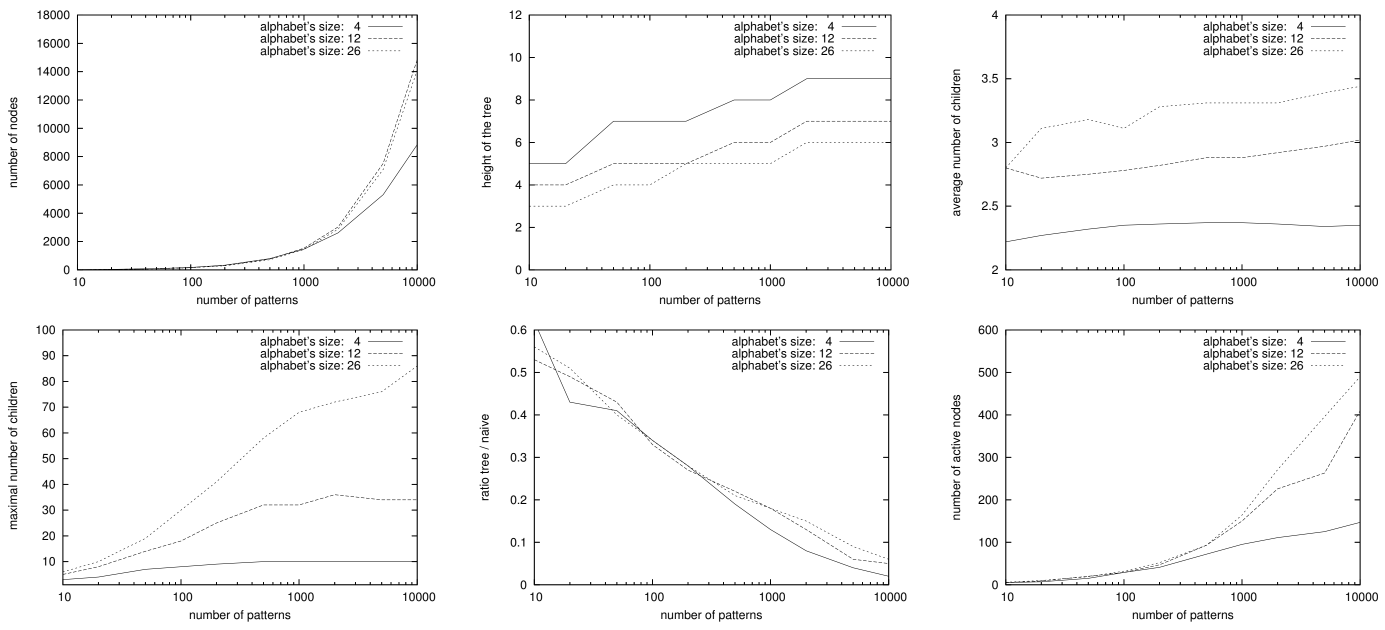

Figure 6. Tree properties and evaluation w.r.t. number of patterns

Pattern removal: a subscription may have to be removed either explicitely (users' choice) or implicitly (timeout). Removing a pattern P from a patterns tree is a trivial operation. Two cases occur:

1. If P has more than one sibling, one just removes P from its parent's children, and then replaces this parent by the lub of the siblings of P.
2. Else, if P has only one sibling P', P' replaces their common parent.

5 Experiments

The structure has been implemented in Java on a Pentium PIV processor (3000MHz) with 1024Mo of main memory. We compare our solution to the naïve approach which evaluates independently each pattern. We use synthetic data, both for patterns and sequence. The evaluation cost, measured as the number of symbols comparisons, is investigated with respect to the following parameters: (1) the number of submitted patterns, (2) the size of the alphabet, (3) the length of the patterns.

Figure 6 summarizes some properties of the pattern tree with respect to several alphabet sizes, for a number of patterns that vary from 1 to 10,000. The top-left graph shows that the size of the alphabet does not affect the total number of nodes when the number of patterns is less than 1,000 because of the low probability to draw duplicate patterns. The impact of the alphabet size on the number of nodes becomes significant for 1,000 patterns and more, and can be directly related to the probability of adding an already existing pattern, which is of course higher for small alphabets.

As expected, with a larger alphabet, the internal nodes have more children. The size of the alphabet has an opposite effect on the height of the tree. The reason is essentially that internal nodes contain more variables when the alphabet's size is large, and therefore capture more patterns. The probability of having two "close" patterns tends to be low, and prevents the generation of precise lubs. When a new pattern is introduced, if the size of the alphabet is large, there is a high probability to insert it as a child of an existing node, rather than creating a new internal node. These differences grow w.r.t. the number of patterns.

Figure 6 also illustrates the benefit of an evaluation based on our structure compared to the independent processing of each patterns. The higher the number of patterns, the more important the gain. The ratio between the number of comparisons for the two solutions is 0.55 for 10 patterns, reaches 0.35 for 100 patterns, 0.15 for 1.000 and 0.05 for 10,000. This ratio hardly depends on the size of the alphabet, even if a small alphabet gives slightly better results over a large number of patterns, for reasons presented above.

On the other hand, the number of simultaneously active automata is strongly related to the properties of the trees, and is therefore influenced by the alphabet size. Two of these properties, namely the filtering rate of an internal node and its number of children, have a divergent impact. A shown by the cost model, the filtering rate of internal nodes tends to be higher for large alphabets, while the number of children grows (on average) for each node. The latter factor explains the relative importance of active automata for large alphabets, since each time a matching is found for an internal node, all its children must be activated.

Next we study the influence of the patterns' length (Figure 7), the total number of patterns being fixed to 10,000 patterns. For a small alphabet, this parameter strongly impacts the number of nodes, the height of the tree and the average number of children. This is due to the low number of possible patterns. For instance with a pattern length of 4 and a size alphabet of 4, only $4^4 = 256$ distinct patterns are possible, whereas this grows up to $4^{12} = 16,777,216$ possibilities when the length is 12.

The same figure also shows that higher patterns' length generate more nodes in the pattern tree. The probability of discovering "close" patterns decreases with the length of the patterns. Nonetheless the size of the alphabet has a divergent impact since with a large alphabet, even short patterns may differ a lot. So to sum up:

- long patterns with a small alphabet increase the probability to present the same symbol at the same position, and therefore to obtain good lubs;
- short patterns with large alphabet quickly provide lubs that present only one constant symbol, and so that capture more patterns.

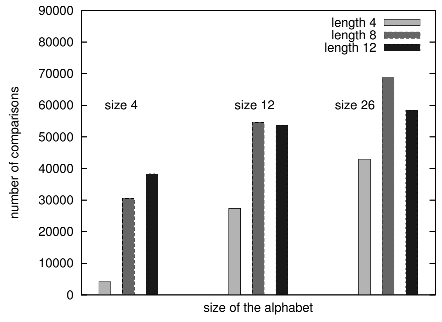

Figure 8. Comparisons w.r.t. the length

Finally Figure 8 shows that for an alphabet with 12 symbols, we obtain better results when the length of the patterns is lower than 8, whereas with a larger alphabet, larger patterns yield a more efficient evaluation.

6 Related work

The present work relates to data streams management [4], publish-subscribe systems [8], sequence databases [13, 17] and approximate string searching [1, 22, 16, 23].

Several papers present algorithms for querying and mining similar subsequences, as well as event detection from time series data (i.e., sequences of real numbers). In [19] the authors describe a fast mining algorithm for retrieving spatio-temporal periodic patterns for objects moving on a partitioned map. In [23], the authors present an algorithm that supports the retrieval of sequences matching a regular expression, with possibly errors. All these approaches are significantly different from ours. In particular none of these algorithms handle the concept of parameterized pattern, featuring variables bound to a single value during a matching attempt. In the approximate string matching area, the pattern-matching model which is closest to ours is presented in [1] which considers patterns with variables that stand for substrings and proves that the problem of finding the longest minimal pattern that subsumes a set of strings is NP-hard. Since we assume that a variable stands for a single symbol, we avoid this complexity. Our language, although less general, leads to a very efficient algorithm.

Several papers deal with the problem of multi-patterns matching [12, 5, 7, 15]. The authors of [5] propose an incremental algorithm which builds a multi-pattern tree. The relaxation does not rely on variables, but on a "wildcard" character. We believe that our refinement relation is more precise. Moreover their evaluation process is quite different for ours. The multi-pattern evaluation presented in [7] relies on two deterministic automata, one built on the prefix of the set of patterns and the other on the reverse patterns. They use the second automaton at a given position until a mismatch occurs, and use the first one to determine the "shift" on the sequence. The technique is designed for simple patterns without variables. Finally [15] describes an algorithm that detects system intrusion, based on multi-patterns matching. The authors propose two filtering algorithms, one based on automata and the other one based on the comparison of the number of occurrences of the symbols between patterns and sequence.

7 Conclusion

This paper proposes a framework for the continuous evaluation of a potentially large set of patterns over unbounded sequences. The introduction of variables in patterns gives rise to a refinement relation based on constraint relaxation. We provide a simple formal model that shows how this relation can be exploited to enable a multiple pattern evaluation mechanism. Our analytical and experimental results confirm the gain. Further, we believe that the ideas of the present work could be extended by applying our variable-relaxation mechanism to the nodes of to tree-structured documents (e.g., HTML or XML) [6, 21]. We plan to investigate this new class of pattern-matching application in the future.

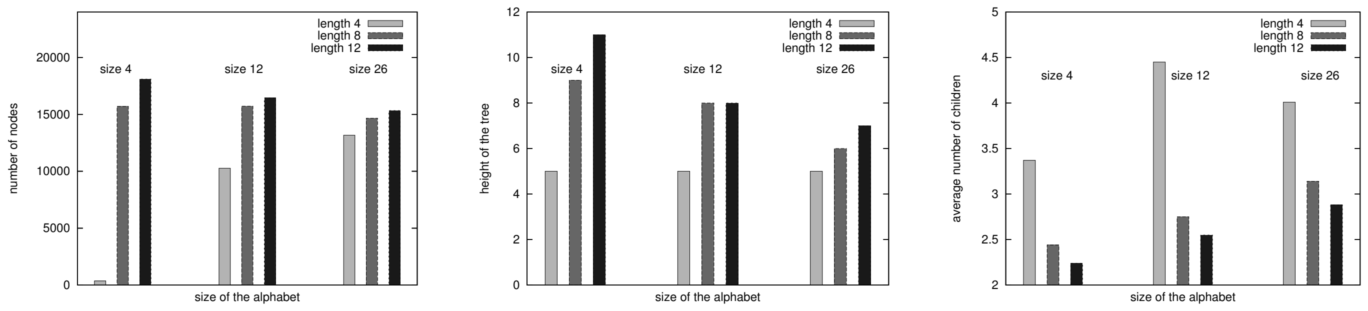

Figure 7. Tree properties w.r.t. the patterns' length

References

[1] D. Angluin. Finding Patterns Common to a Set of Strings. *Journal of Computer and System Sciences*, 21(1):46–62, 1980.

[2] M. J. Atallah and S. Fox, editors. *Algorithms and Theory of Computation Handbook*. CRC Press, Inc., 1998.

[3] A.V.Aho, M.R.Garey, and J.D.Ullman. The Transitive Reduction of a Directed Graph. *SIAM Society for Industrial and Applied Mathematics*, 1:131–137, 1972.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 1–16, 2002.

[5] J. Cai, R. Paige, and R. E. Tarjan. More Efficient Bottom-Up Multi-Pattern Matching in Trees. *Theoretical Computer Science*, 106(1):21–60, 1992.

[6] C. Y. Chan, W. Fan, P. Felber, M. N. Garofalakis, and R. Rastogi. Tree Pattern Aggregation for Scalable XML Data Dissemination. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 826–837, 2002.

[7] M. Crochemore, A. Czumaj, L. Gasieniec, T. Lecroq, W. Plandowski, and W. Rytter. Fast Practical Multi-Pattern Matching. *Information Processing Letters*, 71(3-4):107–113, 1999.

[8] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards Expressive Publish/Subscribe Systems. In *Proc. Intl. Conf. on Extending Data Base Technology (EDBT)*, 2006.

[9] C. du Mouza and P. Rigaux. Mobility Patterns. *GeoInformatica*, 9(4), 2005.

[10] C. du Mouza, P. Rigaux, and M. Scholl. Efficient evaluation of parameterized pattern queries. In *Proc. Intl. Conf. on Information and Knowledge Management (CIKM)*, pages 728–735, 2005.

[11] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, 2001.

[12] C. Forgy. Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. *Artif. Intell.*, 19(1):17–37, 1982.

[13] V. Guralnik and J. Srivastava. Event detection from time series data. In *Proc. Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 33–42, 1999.

[14] D. Knuth, J. Morris, and V. Pratt. Fast Pattern Matching in Strings. *SIAM J. Computing*, 6(2):323–350, 1977.

[15] J. Kuri and G. Navarro. Fast Multipattern Search Algorithms for Intrusion Detection. In *String Processing and Information Retrieval (SPIRE)*, pages 169–180, 2000.

[16] G. M. Landau and U. Vishkin. Fast String Matching with k Differences. *J. Comput. Syst. Sci.*, 37(1):63–78, 1988.

[17] Y.-N. Law, H. Wang, and C. Zaniolo. Query Languages and Data Models for Database Sequences and Data Streams. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 492–503, 2004.

[18] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously Adaptive Continuous Queries over Streams. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, 2002.

[19] N. Mamoulis, H. Cao, G. Kollios, M. Hadjieleftheriou, Y. Tao, and D. W. Cheung. Mining, indexing, and querying historical spatiotemporal data. In *Proc. Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 236–245, 2004.

[20] C. Olston, J. Jiang, and J. Widom. Adaptive Filters for Continuous Queries over Distributed Data Streams. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, pages 563–574, 2003.

[21] D. Shasha, J. T.-L. Wang, H. Shan, and K. Zhang. ATreeGrep: Approximate Searching in Unordered Trees. In *Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM)*, pages 89–98, 2002.

[22] E. Ukkonen. Finding Approximate Patterns in Strings. *J. Algorithms*, 6(1):132–137, 1985.

[23] S. Wu and U. Manber. Fast Text Searching Allowing Errors. *Commun. ACM*, 35(10):83–91, 1992.