An extensible light-weight XML-based monitoring system for sequence databases

Peer-reviewed author version

VAN DE CRAEN, Dieter; NEVEN, Frank & KOCH, Kerstin (2006) An extensible light-weight XML-based monitoring system for sequence databases. In: Data Integration in the Life Sciences, Proceedings. p. 280-296.

DOI: 10.1007/11799511_25
Handle: http://hdl.handle.net/1942/1419

An extensible light-weight XML-based monitoring system for sequence databases

Dieter Van de Craen*, Frank Neven, and Kerstin Koch

Hasselt University and Transnational University of Limburg
School for Information Technology
{firstname.lastname}@uhasselt.be

Abstract. Life science researchers want biological information in their interest to become available to them as soon as possible. A monitoring system is a solution that relieves biologists from periodic exploration of databases. In particular, it allows them to express their interest in certain data by means of queries/constraints; they are then notified when new data arrives satisfying these queries/constraints. We describe a sequence monitoring system XSeqM where users can combine metadata queries on sequence records with constraints on an alignment against a given source sequence. The system is an XML-based solution where constraints are specified through search fields in a user-friendly web interface and which are then translated to corresponding XPath-expressions. The system is easily extensible as addition of new databases to the system then only amounts to the specification of new mappings from search fields to XPath-expressions. To protect private source sequences obtained in labs, it is imperative that researchers do not have to upload their sequences to a general untrusted system, but that they can run XSeqM locally. To keep the system light-weight, we therefore introduce an optimization technique based on query containment to reduce the number of XPath-evaluations which constitutes the bottleneck of the system. We experimentally validate this technique and show that it can drastically improve the running time.

1 Introduction

Motivation. Due to the increase in the speed of sequencing of genes and proteins, sequence databases, such as Genbank, double in size every two years [26]. This rapid expansion of data motivates researchers to repeat search queries over time. Indeed, a BLAST-search [13] that does not produce any useful result today might do so tomorrow. In this paper, we therefore propose a user-friendly sequence monitoring system XSeqM (*eXtensible Sequence Monitor*) that relieves researchers from repeating such searches over time.

We provide two motivating examples:

* Contact author

1. Researchers in a lab have obtained one or a few sequences of genes or proteins for which a BLAST-search only gives similarities for small regions of the sequence. No highly similar, annotated sequences are available in any database which might give hints for the function of the gene or protein. Therefore, the researchers regularly repeat BLAST-searches against several databases to find genes or proteins with a higher similarity.
2. A researcher has obtained a gene g expressed in the central nervous system (CNS) of the rainbow trout and is interested to learn about genes similar to g which are expressed in the peripheral nervous system (PNS) in any fish organism or mammal. She therefore repeats a BLAST-search with the gene g on a weekly basis.

The two tasks described above are tedious and time consuming when executed manually: not only the BLAST-searches themselves, but also the post-processing of the BLAST-reports (if any) to sort out relevant matches from irrelevant ones. Indeed, in situation (1), a match could be irrelevant as the matched part of the sequence is too small or the likelihood of the match expressed by the E-value is too large. In situation (2), all BLAST-hits from non-fish and non-mammal species should be discarded together with those that are not mRNA and that do not refer to the PNS.

A solution: the XSeqM-system. In the XSeqM-system users can register BLAST-requests combined with constraints on the metadata of a sequence record. All requests are checked locally by the system after retrieval of the daily updates from the respective databases and users are informed, for instance through email, when relevant results are found. Figure 2 shows part of the monitor request related to situation (2). In brief, every such request specifies the following information:

- a database of interest (e.g., Genbank, SwissProt, ...),
- a sequence of interest (e.g., the gene g),
- constraints on the metadata (e.g., classification should contain the string 'fish' and molecular type should equal 'mRNA')
- an alignment program and its parameters (e.g., BLAST with word size 11 and matrix PAM30)
- relevance constraints (e.g., size of match should be greater than 20 and E-value should be smaller than e^{-10}).

The XSeqM-system has the following characteristics:

1. XSeqM is light-weight. It can be installed locally in a lab on a computer with average system requirements. This is important, as, referring to situation (1) above, research labs can be hesitant to upload their newly found sequences in a public system as some of them might be candidates for a patent application.
2. XSeqM is user-friendly as it hides all use of XML: users interact with the system through a Web-interface where search fields can be combined using the logical operators, much like other query and monitoring systems such as SRS and PubCrawler [22].

3. XSeqM is a flexible XML-based solution to which any sequence database can be added that makes updates available and whose format can be transformed into XML. Almost all sources nowadays allow to export information in XML-format or there are third party tools available to convert existing formats to XML. The administrator determines for every sequence database a number of search fields. For every search field f, an XPath-expression P_f is created that selects the corresponding value in every XML-file in the update. Table 1, for instance, lists the interesting search fields for a GenBank record and the corresponding XPath-expressions. Every user request is then translated under the hood to a Boolean combination of XPath-expressions. Similarly, relevance constraints on BLAST-reports are translated into XPath-expressions over the XML-representation. Therefore, in principle, any XPath-expressible constraints can be used.

Efficient evaluation. The main technical part of the paper deals with efficient execution of all monitoring requests. In brief, the system executes the following steps. Let $m_1, \ldots, m_k$ be all monitoring requests with corresponding constraints $p_1, \ldots, p_k$ on the metadata, i.e. Boolean combinations of XPath-expressions. For every sequence record s in the update, we need to check which expressions p_i match s. When p_i is successfully matched, we BLAST the sequence in s against the sequence in m_i. When all relevance constraints of m_i on the BLAST-report are satisfied, the owner of request m_i is alerted. As an alignment of sequences through BLAST is expensive, it is imperative to first check the metadata constraints and only start BLAST for those sequences which are selected.

As every local lab is considered to have its own system, we consider systems of moderate size (say, a few thousands of monitoring requests). Daily updates to Genbank vary in size from 50 to 200 Megabytes (zipped): these contain between 30000 and 150000 sequences. The bottleneck of the system is in the evaluation of the constraints $p_1, \ldots, p_k$ for every sequence record s in the update. A direct approach using a standard XPath-evaluator like Xalan[1] takes more than 24 hours and is therefore not an option. Powerful fast streaming XPath-engines have been proposed over the past years [21,12] which can handle millions of XPath-expressions. Unfortunately, we cannot use these engines directly: to ensure high throughput streaming engines do not consider full XPath. In particular, they do not consider arbitrary Boolean combinations of XPath-expression or allow to test whether a certain given string occurs as a substring of a text element. We therefore make use of the state-of-the-art evaluator YFilter [18,19] as a first preprocessing step to extract string-values from sequence records. More precisely, by evaluating for every search field the corresponding expression P_f on the update, we get for every sequence record a complex value representation on which the metadata constraints can be checked. E.g., Table 2 contains such a representation for the GenBank record of Figure 1 through the XPath expressions in Table 1. In a second step, we then evaluate every pattern p_i on this representation. An additional advantage of this method is that more advanced pattern matching on string values can be used than is available in XPath. For instance, one could require that the string value matches a given regular expression.

f	P_f
organism	/p/e[@class="source"]/Qualifier[@value-type="organism"]/@value
accession	/p/@ic-acckey
gi	/p/Attribute[@name="primary_id"]/@content
author name	/p/q[@title="Sequence References"]/Reference/RefAuthors/text()
title	/p/q[@title="Sequence References"]/Reference/RefTitle/text()
keyword	/p/Attribute[@name="keyword"]/@content
comment	/p/Attribute[@name="comment"]/@content
classification	/p/Attribute[@name="classification"]/@content
Feature key	/p/e/@class
Gene name	/p/e[@class="gene"]/Qualifier[@value-type="gene"]/@value
Protein name	/p/e[@class="cds"]/Qualifier[@value-type="product"]/@value
chromosome	/p/e[@class="source"]/Qualifier[@value-type="chromosome"]/@value
molecular type	/p/e[@class="source"]/Qualifier[@value-type="mol_type"]/@value
tissue type	/p/e[@class="source"]/Qualifier[@value-type="tissue_type"]/@value
tissue library	/p/e[@class="source"]/Qualifier[@value-type="tissue_lib"]/@value
cell line	/p/e[@class="source"]/Qualifier[@value-type="cell_line"]/@value
development stage	/p/e[@class="source"]/Qualifier[@value-type="dev_stage"]/@value
EC Number	/p/e[@class="cds"]/Qualifier[@value-type="EC_number"]/@value
p	/Bsml/Definitions/Sequences/Sequence
e	Feature-tables/Feature-table[@title="Features"]/Feature
q	Feature-tables/Feature-table

Table 1. Search fields for a GenBank record and corresponding XPath-expressions.

We consider an optimization based on containment of constraints. As the system runs at a local lab, chances are high that many constraints on the metadata are related. For instance, a constraint could require that the organism should contain the string 'Oncorhynchus' while another query could require that the organism should equal 'Oncorhynchus mykiss' and the tissue type equals 'brain'. Clearly, the second constraint implies the first. So, we know that the first constraint is true when the second is, and the second is false when the first is. Our optimization technique exploits these ideas to reduce the number of evaluations. More precisely, we define a graph structure that captures the relationships between the constraints and consider two forms of propagation: false propagation and true propagation. We experimentally show that false propagation outperforms true propagation and the pure streaming approach.

Finally, we discuss how to incrementally maintain the containment graph. It never has to be computed from scratch. The insertion operation is time consuming as in the worst case it involves a linear number of containment checks (a coNP-hard problem [20]). Luckily only a limited number of insertions are expected on a daily basis, say at most hundred, which for a system already containing 5000 requests can be done in less than 100 minutes. In case a larger number of insertions is required, we discuss a technique that accelerates the containment check at the expense of introducing more requests: constraints are transformed into disjunctive normal form, testing containment of conjuncts can then be done in linear time. For instance, adding 100 request to a containment graph with 5000 nodes then only takes 12 seconds.

f	values
organism	{“Oncorhynchus mykiss”}
accession	{ “AM181351” }
gi	{ “84993308” }
author name	{ “Zarkadis,I.K. and Marioli,D.”, “Zarkadis,I.K.” }
title	{ “Cloning of the vitronectin gene in rainbow trout”, “Direct Submission” }
keyword	{ “vitronectin protein 1”, “vtn1 gene” }
comment	{ }
classification	{ “mykiss Oncorhynchus Salmonidae Salmoniformes Protacanthopterygii Euteleostei Teleostei Neopterygii Actinopterygii Euteleostomi Vertebrata Craniata Chordata Metazoa Eukaryota” }
Feature key	{ “source”, “gene”, “cds” }
Gene name	{ “vtn1” }
Protein name	{ “vitronectin protein 1” }
chromosome	{ }
molecular type	{“mRNA”}
tissue type	{ “liver” }
tissue library	{ }
cell line	{ }
development stage	{ }
EC Number	{ }

Table 2. Complex value representation of the GenBank record in Figure 1

Outline. This paper is organized as follows. In Section 2, we survey other monitoring approaches. Section 3 introduces XML and XPath. Section 4 gives an overview of XSeqM. In Section 5, we outline several evaluation strategies. Section 6 reports on our experiments. In Section 7, we discuss the incremental maintenance of the containment graph. We conclude in Section 8.

2 Related Work

Existing alerting systems like BioMail, JADE or Science Direct are used for literature alerts [3,4,9]. They search the PubMed database in given intervals and alert users via email if new publications matching special keywords are available [25]. The only system integrating query possibilities for Genbank in addition to literature alerts is PubCrawler [22,23]. PubCrawler provides a user with the possiblity to define two types of queries. The first type is a keyword search and the second is a neighborhood query. With a neighborhood query a user can express his interest in articles or sequences that are similar to given articles or sequences already present in the database. A limitation of this approach is that the user can not enter an unpublished sequence which has no identifier assigned yet. Also, advanced options in the comparison with other sequences are not provided, e.g., the minimal length of a match or the E-value. XSeqM does provide these possibities and allows for the combination of a keyword search and an alignment with any given sequence.

XML filtering systems evaluate a set of queries against a stream of documents. The XMLTK system [21] combines all path expressions into a single deterministic finite automaton. YFilter [18,19], the successor of XFilter [12], combines all

expressions in one nondeterministic finite automaton. These systems thus employ a finite state automaton for all the XPath-expressions. The XML stream is parsed by a SAX parser and the SAX events are streamed through the finite state automaton. A query matches a document if during parsing an accepting state for that query is reached. The main limitation of these systems compared to XSeqM is that they do not support full XPath. As the translation of user queries in XSeqM can result in complex XPath-expressions, these systems can not be applied directly in our situation.

In [14,15] and [24] optimization of navigational queries on life science sources is investigated. In this setting alternate paths are possible to evaluate a query. The focus in [14,15] is on finding a set of paths that maximizes the number of results while satisfying a constraint on the evaluation cost. Minimizing the total number of accesses to sources when evaluating multiple queries in batch mode is discussed in [24]. The goal of XSeqM differs from these as we want to monitor multiple sources seperately rather then answering queries over multiple sources.

3 XML and XPath

The eXtensible Markup Language (XML) is a standard for data exchange on the Web [10]. Most bioinformatics data formats can be converted into an XML representation. Numerous XML formats for a wide range of biological data are available. Some examples are BSML, SPTr-XML, GO-XML,... [16].

XPath is an XML pattern language for locating information in XML documents [17]. In particular, XPath can retrieve the value of elements or attributes and can test whether that value satisfies a certain condition. We give an example of both. The expression `//Attribute[@name="classification"]/@content`, for instance retrieves the classification of an entry as the actual classification is the value of the `content` attribute of an `Attribute` element that has a `name` attribute with value 'classification'. The expression

```
boolean(//Attribute[@name="classification" and contains(@content,"Mammalia"])
```

checks whether the classification contains the string 'Mammalia'. XPath can also be used to query the XML-representation of a BLAST-report. For instance, the expression `//Hit[Hit_num/text()="1"]/Hit_hsps/Hsp/Hsp_evalue/text()` selects the E-value of the first hit.

4 Monitoring System

We detail the three different components of XSeqM which are graphically illustrated in Figure 3.

1. The Input Module consists of the WWW Interface and the Query Translation Module. As illustrated in Figure 2(top), a query is created in the WWW Interface by uploading a sequence and specifying search terms in search fields. These search fields are then linked together by selecting the appropriate logical connectors: AND, OR and NOT, and parentheses. This method of operation is similar to the one used in other query and monitoring systems

```
LOCUS       AM181351                674 bp    mRNA    linear   VRT 16-JAN-2006
DEFINITION  Oncorhynchus mykiss partial mRNA for vitronectin protein 1 (vtn1
            gene), isolated from liver.
ACCESSION   AM181351
VERSION     AM181351.1  GI:84993308
KEYWORDS    vitronectin protein 1; vtn1 gene.
SOURCE      Oncorhynchus mykiss (rainbow trout)
  ORGANISM  Oncorhynchus mykiss
            Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
            Actinopterygii; Neopterygii; Teleostei; Euteleostei;
            Protacanthopterygii; Salmoniformes; Salmonidae; Oncorhynchus.
REFERENCE   1
  AUTHORS   Zarkadis,I.K. and Marioli,D.
  TITLE     Cloning of the vitronectin gene in rainbow trout
  JOURNAL   Unpublished
REFERENCE   2  (bases 1 to 674)
  AUTHORS   Zarkadis,I.K.
  TITLE     Direct Submission
  JOURNAL   Submitted (11-JAN-2006) Zarkadis I.K., Dept. of Biology, School of
            Medicine, University of Patras, Rion, Panepistimioupolis, ...
FEATURES             Location/Qualifiers
     source          1..674
                     /organism="Oncorhynchus mykiss"
                     /mol_type="mRNA"
                     /db_xref="taxon:8022"
                     /tissue_type="liver"
     gene            <1..>674
                     /gene="vtn1"
     CDS             <1..>674
                     /gene="vtn1"
                     /codon_start=1
                     /product="vitronectin protein 1"
                     /protein_id="CAJ57657.1"
                     /db_xref="GI:84993309"
                     /translation="SCCMDF..."
ORIGIN
        1 agctgctgca tggacttcga cagtgcctgc cctaggaaga tttcccgcgg tgacacattt
          ...
      661 tgtgtgcgct tgac
```

Fig. 1. Example GenBank entry

such as SRS and PubCrawler. Queries entered by users are then translated by the Query Translation Module to a Boolean formula and a mapping from which the corresponding XPath expression can be constructed. An example of part of this translation is given in Figure 2(bottom). The user therefore does not have to be aware of the underlying technology used. The monitor request is stored in the local repository.

2. The Evaluation Module is responsible for the actual evaluation of the monitor requests. It consist of the Request Evaluator, the Alignment Module and the Report Generator. When the Evaluation Module receives an update of a database, say GenBank, the monitor requests concerning GenBank are fetched from the local repository. The evaluation then proceeds as follows. First, the Request Evaluator evaluates the metadata constraints of the monitoring requests on all sequence records in the update. The Alignment Module then aligns every selected sequence with the corresponding source sequences. Finally, the Report Generator constructs a report from every BLAST-report that satisfies the relevance constraints and notifies the owner of the monitor request.

3. The Update Module consist of the BioDBInterface and the XML Converter Module. The BioDBInterface Module checks at regular timepoints whether updates to some of the monitored databases are available. If such an update is available, then the BioDBInterface Module fetches this update and passes it on to the Evaluation Module. Despite the fact that more and more biological data is available as XML, not all data is. In such a case, the XML Converter Module will convert the update into an XML-format.

5 Evaluation strategies

We provide an abstract view of the different parts of the evaluation algorithm. Let $m_1, \ldots, m_k$ be an enumeration of all monitoring requests. Every request $m_i = (p_i, s_i, r_i)$ consists of a metadata constraint p_i, a sequence s_i, and a relevance constraint r_i. Let $u_1, \ldots, u_\ell$ be an enumeration of sequence records constituting an update. The system provides the following steps:

1. Compute the set of pairs $N = \{(j, i) \mid u_j \text{ matches } p_i\}$.
2. For every $(j, i) \in N$, align the sequences u_j and s_i through BLAST resulting in a BLAST-report $R_{j,i}$.
3. When $R_{j,i}$ matches r_i, warn the owner of request m_i.

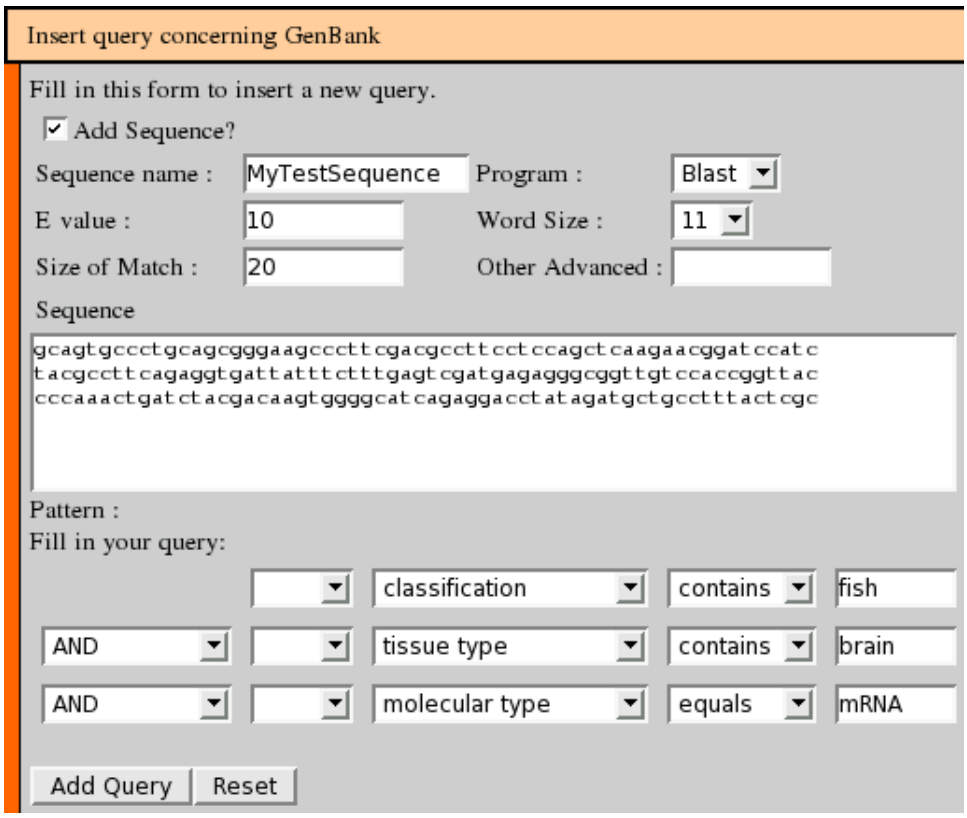
The bottleneck of the system is located in step (1) above: testing the metadata constraints. When M and N are the number of monitoring requests and the number of sequence records in an update, respectively, then $M \times U$ constraints need to be checked. We consider systems where M can be 5000 and U can be 10^5. Step (2) can be evaluated quite fast (on average 10^4 pairs of sequences can be aligned with BLAST on a local system in less then half a minute). For step (3), the same techniques as for step (1) can be used, although in general this step can be done quite fast as the number of BLAST-reports will be much smaller than $M \times U$. In the rest of this section, we outline several evaluation strategies for step (1) which are experimentally evaluated in the next section.

5.1 Naive brute force evaluation

The first evaluation strategy is a simple brute force method which tests every constraint p_i for every entry in the update. To evaluate the XPath-expressions, we use Xalan [1].

5.2 XML streaming approach

An XML stream query processing system takes as input a stream of XML documents on which it evaluates queries simultaneously. Filtering systems such as XFilter, YFilter, XMLTK, ... are freely available and provide efficient evaluation of large numbers of XPath-expressions. The problem with these systems is that the XPath fragment they consider is not powerful enough to express our user constraints. However, if we look at the number of search fields that can be queried in our setting, we observe that this number is small (typically 10 to 20) and fixed in advance. So, instead of evaluating the XPath-expressions generated from the user constraints directly on the updates, we proceed in two steps:



Blast

ID	sequence	Evalue	wordsize	MatchSize
...				
51	gcagtgcc...	10	11	20
...				

Query

ID	userID	database	formula
...			
51	8	genbank	v_51_1 & v_51_2 & v_51_3
...			

Mapping

ID	variable	querytype	keyword	value
...				
51	v_51_1	contains	classification	fish
51	v_51_2	contains	tissue_type	brain
51	v_51_3	equals	molecular_type	mRNA
...				

Fig. 2. Example of a monitoring request and its translation.

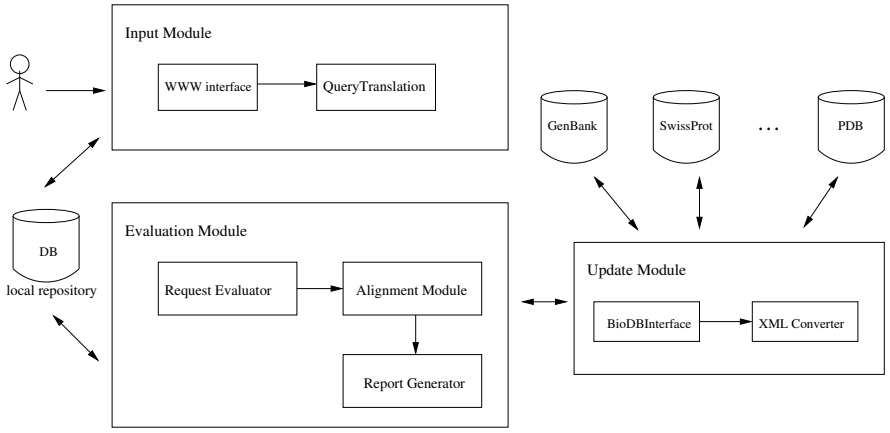


Fig. 3. Overview of the modules

1. Using YFilter, we retrieve all the values for the search fields for a sequence record of an update and create for each record a complex value representation. E.g., Table 2 contains a complex value representation of the GenBank record of Figure 1 obtained through the XPath expressions in Table 1.
2. In a second step, we evaluate the metadata constraints on this complex value representation. For instance, the constraint

$$\textit{classification.contains('Teleostei') AND tissue\_type.contains('brain') AND molecular\_type.contains('mRNA')}$$

is not satisfied on the record in Table 2 as the tissue_type does not contain brain. The semantics of the contains('s') operator is that at least one of the strings in the set should contain the string 's' as a substring.

5.3 Query containment

The evaluation of expressions in step two above is still naive: all expressions are matched against all entries in the update. As the XSeqM-system runs at a local lab where researchers are working on related topics, chances are high that some constraints on the metadata are related. A useful notion in this context is the following: a constraint p is contained in a constraint p', denoted $p \subseteq p'$, if whenever a sequence record satisfies p it also satisfies p'. For instance, let p be the constraint

$$\textit{organism.equals('Oncorhynchus mykiss') AND tissue\_type.contains('brain') AND molecular\_type.contains('mRNA')}$$

and let p' be the constraint *organism.contains('Oncorhynchus')*. Then it should be clear that every record which satisfies p also satisfies p'.

So, containment checking of constraints basically reduces to containment checking of propositional logical formulas. However, some care is needed when

dealing with the 'contains' and 'equals' predicate referring to the same search field. We use the following algorithm that we illustrate on the above example:

- Rewrite the constraints p and p' to logical formulas q and q' over different propositional symbols.
  That is, q equals $a \wedge b \wedge c$ and q' equals d. Here, a, b, c and d stand for organism.equals('Oncorhynchus mykiss'), tissue_type.contains('brain'), molecular_type.contains('mRNA'), and organism.contains('Oncorhynchus'), respectively.
- Let γ be a propositional formula initially set to true. For every pair of propositional variables x and y referring to the same search field, test whether the constraint corresponding to x is contained in the constraint corresponding to y. If so, add $\neg x \vee y$ to γ. The intuition is that γ restricts the set of possible truth assignments to those that correspond to the semantics of the 'contains' and 'equals' predicates. In particular, the formula $\neg x \vee y$ only accepts truth assignments that assign true to y when x is also true, which encodes that x implies y.
  The only variables referring to the same search field are a and d. Clearly, a is contained in d as every record satisfying organism.equals('Oncorhynchus mykiss') also satisfies organism.contains('Oncorhynchus'). So, γ is the formula $\neg a \vee d$.
- Now, $p \subseteq p'$ iff $q \wedge \neg q' \wedge \gamma$ is unsatisfiable.
  So for our example, we need to test that $a \wedge b \wedge c \wedge \neg d \wedge (\neg a \vee d)$ is not satisfiable, which is the case. Indeed, the only way to satisfy the first four conjuncts is to set a, b, and c true and d false, but this is prohibited by the last conjunct.

In general testing unsatisfiability is coNP-complete [20]. Fortunately, the expressions we consider are very small. We make use of the state-of-the-art SAT-solver Limmat [6]. As our formulas are in general not in CNF, we use Limboole [5], a front end to Limmat that allows to check unsatisfiability of arbitrary formulas and not just formulas in CNF.

The **containment DAG** of a set of constraints is a directed acyclic graph (DAG) without any transitive edges where every node represents a set of equivalent constraints and there is an edge from node n to node n' if every expression in n is contained in every expression in n'. Note that it is sufficient to test if one expression from n is contained in one expression from n'. A **source** is a node without incoming edges; a **sink** is a node without outgoing edges. Note that a DAG can have multiple sources and sinks.

We make the following observations:

- to check whether a sequence record matches the expressions in a node n, it suffices to test this for one expression in n;
- when an expression in n is true for a sequence record, then all expressions in descendant nodes of n are true for that record; and
- when an expression in n is false for a sequence record, then all expressions in ancestor nodes of n are false for that record.

In the following, the evaluation of a node against a sequence record corresponds to selecting one of the equivalent expressions the node represents and matching this expression against the record. The above observations lead to two related optimization techniques allowing to discard nodes in the containment DAG:

1. false propagation : start evaluation in the sinks, when a node evaluates to false all ancestors can be discarded as they evaluate to false, when the node evaluates to true all parents have to be addressed;
2. true propagation : start evaluation in the sources, when a node evaluates to true all descendants can be discarded as they evaluate to true as well, when the node evaluates to false all its children have to be addressed.

Note that a node can be reached by multiple paths. So, to avoid multiple evaluations of nodes every node carries a bit indicating whether the node is already evaluated or not. It is clear that if expressions seldom match entries in the update then false propagation will result in a strong decrease in the number of actual evaluations. In the case that expressions frequently match entries, the use of true propagation can be advantageous.

6 Experiments

In this section, we experimentally validate our optimization techniques. We created monitoring requests resulting in three types of containment DAGs T1, T2, and R (cf. Figure 4). We repeated our experiments for di erent numbers of monitoring request (from 1000 till 5000). We only report on the case with 5000 requests as all experiments produced similar results. The experiments were performed on a Pentium IV (3.0 GHz) architecture with 1 GB of internal memory running under Linux 2.6. All programs are written in Java.

The metadata constraints were created by extracting possible values out of available updates. The rst type of containment DAG (T1) is specially tailored for false propagation. Part of a DAG of type T1 is given in Figure 4(top). It is a reversed tree consisting of a small number of sinks. It is constructed by only making use of AND-operators. The idea is that every sink represents the most general constraint which is subsequently re ned by additional constraints when progressing upwards. For instance, a sink may state that the organism in the update matches `Oncorhynchus mykiss', its parent may re ne this by adding another constraint, namely that the molecular type must be `mRNA'. Trees can be disjoint, for instance, when each of them corresponds to an organism.

The shape of the second type of containment DAG (T2) is the reverse of the rst one and is ideal for true propagation. Part of a DAG of this type is given in Figure 4(middle). The idea is that each source is the most restrictive constraint which gets relaxed by every descendant.

The last type of containment DAG (R) was created by generating random constraints (using AND, OR, and NOT-operators) and creating the containment DAG. Figure 4(bottom) shows the typical shape of such a containment DAG. To keep the comparison of the di erent evaluation strategies fair, we have eliminated all equivalent constraints but one from every node.