Where is my Information?

Raul Miguel da Silva Martins

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

João Pereira

Presidente: Doutor Luís Rodrigues
Orientador: Doutor João Dias Pereira
Vogais: Doutor Andreas Miroslaus Wichert

September 2010
Agradecimentos

I wish to express my utmost gratitude to my Professor, Dr. João Pereira for providing me such knowledgable guidance while allowing me the room to work and solve problems on my own. I reckon that this thesis would not have been possible if it was not for his great expertise, problem-solving ability and constant availability to help me, either by answering my questions or by guiding me to better solutions. I sincerely thank him for being the kind of supervisor I needed to encourage and support me through this thesis and most of all, I thank him for his patience.

I am also grateful to my parents for supporting all my studies throughout my life and for their understanding and endless encouragement provided during my Master grade, as well as for providing such an amazing environment of peace in which I could easily focus on my thesis.

Lisboa, November 23, 2011
Raul Martins
Resumo

Atualmente há uma quantidade muito elevada de informação dinâmica na web, trazendo a necessidade de encontrar formas mais eficientes de ajudar os utilizadores a encontrar as informações que lhes interessam assim que estas são publicadas. Uma solução comum para lidar com este tipo de problemas é empregar sistemas publish/subscribe. No entanto, de forma a ser eficiente, estes sistemas devem conseguir resolver o problema de filtragem. Isto é, computar as subscrições que são verificadas por um determinado bloco de informação. Geralmente, usando XML como uma forma de representar a informação publicada implica o uso do XPath como linguagem de subscrição. Devido à elevada expressividade do XPath, resolver o problema de filtragem do XML de uma forma eficiente torna-se mais complexo, como tal, normalmente as soluções acabam por reduzir a expressividade da linguagem de subscrição à de um pequeno subconjunto da linguagem XPath, de forma a alcançar uma maior eficiência. Neste trabalho, propomos um algoritmo que, baseando-se num algoritmo orientado a sistemas publish/subscribe value-based, pode, de forma eficiente resolver problemas de filtragem num contexto publish/subscribe em XML suportando simultaneamente um subconjunto da linguagem XPath com alta expressividade.
Abstract

Currently, there is a very high amount of dynamic information on the web, bringing the need to find efficient ways to help users select the information they are interested in as soon as it is published. A common solution to cope with this problem is to employ publish/subscribe systems. However, in order to be efficient, these systems must solve the filtering problem: compute subscriptions matched by a given piece of information. Usually, using XML as a way to represent published information entails the usage of XPath as a subscription language. Due to XPath’s high expressiveness, solving the XML filtering problem efficiently becomes more complex, and as such, common solutions usually reduce the expressiveness of the subscription language to a small subset of the XPath Language in order to achieve higher efficiency. In this work we propose an algorithm that, based on a value-based matching algorithm, can efficiently solve the filtering problem in a XML publish/subscribe environment, while supporting a significantly expressive subset of the XPath language.
Palavras Chave

broken_at_sentence_boundaries

Palavras Chave

Filtragem XML
XPath
Publish/Subscribe
Disseminação de Informação
Algoritmo de Matching

Keywords

XML Filtering
XPath
Publish/Subscribe
Information Dissemination
Matching Algorithm
Índice

I  Context  1

   1  Context  3
      1.1  Introduction  3
      1.2  XPath  4

   2  Related Work  7
      2.1  Overview  7
      2.2  XFilter  8
      2.3  YFilter  10
      2.4  FiST  12
      2.5  pFiST  14
      2.6  AFilter  15
      2.7  XFiS  18
      2.8  Le Subscribe  19
      2.9  Predicate Filter  21
      2.10  Summary  24
         2.10.0.1  Automata-based  24
         2.10.0.2  Sequence-based  25
         2.10.0.3  Stack-based  26
         2.10.0.4  Pushdown Approaches  26

II  WFilter  29

   3  WFilter Matching Solution  31
      3.1  Event Representation  32
      3.2  Subscriptions  32
List of Figures

2.1 A query Q1 and its respective three path nodes .......................... 9
2.2 Query index on the beginning of a matching process .................. 10
2.3 NFA Transitions for basic location steps ................................. 11
2.4 Resulting NFA combining eight queries ................................. 12
2.5 A query and the respective LPS ............................................. 13
2.6 A query's profile sequence .................................................. 13
2.7 Sequence Index ................................................................. 14
2.8 Axisview for Q1,Q2,Q3 and Q4 .............................................. 16
2.9 PRlabel-tree for Q1,Q2,Q3 and Q4 ........................................ 17
2.10 SFlabel-tree for Q1,Q2,Q3 and Q4 ........................................ 17
2.11 Stackbranch after parsing <a><d><a><b> ................................. 17
2.12 A Document T converted into a TSS with the support of a stack .... 18
2.13 An example of twigpositions during an execution of the matching algorithm ............................................. 19
2.14 The predicate index ......................................................... 23
3.1 A XML Document broke down into three paths: p1, p2, p3 ............... 32
3.2 Variables in an attribute expression ....................................... 34
3.3 Process of converting an attribute expression to a predicate that can be indexed in WFilter .................................................. 35
3.4 A sample predicate vector with 6 predicates indexed .................... 36
3.5 An example of a cluster that stores subscriptions of size 3 with access predicate p3 .......................... 36
3.6 A sample of a possible execution using the previous examples .......... 38
3.7 An example of the result of compressing events .......................... 41
4.1 Loading time for increasing amount of subscriptions with no attribute expressions .......................... 46
4.2 Loading time for increasing amount of subscriptions with attribute expressions ................................... 47
4.3 Matching time for increasing amount of subscriptions with no attribute expressions .......................... 48
4.4 Matching time for increasing amount of subscriptions with attribute expressions 49
4.5 Impact of wildcards on both WFiler and YFilter .......................... 50
4.6 Impact of ancestor-descendant on both WFiler and YFilter using subscriptions without attribute expressions ................................................. 51
4.7 Impact of ancestor-descendant on both WFiler and YFilter with subscriptions containing attribute expressions ................................................. 52
4.8 Impact of varying the depth on the subscriptions .......................... 52
4.9 Impact of increasing the amount attribute expressions on the subscriptions .... 53
List of Tables

2.1 Summary of the key components of the matching algorithms . . . . . . . . . . 27
Context
1.1 Introduction

The Internet acts a world-scale global distributed system of interconnected computers storing and providing information to users all over the world. Due to the magnitude of the Internet, we can consider it as an astronomical ever-growing repository of new information. At the same time, the number of users interested in some of that information is also growing and because of this, the accessibility of the information and the easiness of publishing new information has increased. This has brought attention upon ways of selecting which information concerns which users. One of these ways, is having a system working as an intermediary between users and information, allowing such users to specify their interests and, as the information flows in, automatically deliver the desired information to the interested users, providing a way to help users filter which information is relevant to them.

The systems that provide the aforementioned filtering are called Information filtering systems (Hanani, Shapira, & Shoval, 2001). These systems apply automated methods to filter information streams determining which information is relevant to which users. Their main concern is to prevent information overload and to spare interested users of having to find information by themselves. In order for this to happen, users must explicitly specify their interests through a specific language beforehand or, have their interests determined implicitly based on their behavior. The main focus in this paper is the first situation where the users interests are stored in the system as a set of subscriptions; this way, information streams are processed and the filtering system can send the relevant information to the users.

These filtering systems work based on a publish/subscribe paradigm: the users specify their interests by adding subscriptions to the system. The system then stores the users subscriptions and, as information changes or new one becomes available, an event containing that information is fired and the system handles it accordingly. The system then, takes care of determining which users are interested in that new information and delivers it to them automatically in every event. In these systems, information and subscriptions can be represented on many ways. For instance, information can be represented as a set of attribute-value pairs of XML. In this case, information is represented in the form of documents and subscriptions are expressed in a specific subscription language that can range from a simple conjunction of equality/non-equality conditions over the event attributes to a more complex language allowing greater selectivity and more operations over the incoming information.

A key issue in the filtering systems is the expressiveness of the subscription language, that is, the language might have a limited ability to accurately represent user’s interests. Due to the wide use of the XML as a standard form for representing data in several contexts, including the web, along with the high expressiveness of XML query languages, these query languages are very interesting subscription languages and are highly expressive. Besides this, they also
address XML content which is, as stated above, a widely used common form of representing structured data nowadays. Because of this, many XML based filtering systems have been developed where users express their interests through an XML query language and the information is represented in XML. The XML query language that is mainly used is XPath (Berglund et al., 2003), which also makes it the most interesting subscription language when dealing with XML data.

An example of an XPath statement would be to consider an XML document that contains the following structure: The root node is called News and all its child nodes are categories of news like politics, economy and sports. Assuming there was an interest in selecting the politics news, it could be represented in the following query: /News/Politics

The high expressiveness of the XPath/XML comes with a cost, the higher the expressiveness of the language, the higher the cost to determine the matching information. Taking into account that these systems are dealing with millions of users, millions of subscriptions and hundreds of events, it’s critical that they offer an efficient performance when determining the subscriptions matched by an event. With such a high-rate of events and so many subscriptions, it’s the only way for the system to be able to keep up. Basically, the goal is in finding an efficient enough way to determine the information relevant to each user based on his/hers interests. This challenge of determining which information concerns which users efficiently is called the matching problem.

The matching problem is the opposite problem of a traditional database efficient querying. In traditional databases there is a high amount of information stored and optimized ways of querying it are exploited (i.e. indexes). In this situation there is a high amount of queries (user profiles) stored and the main objective is to process which of the many queries are satisfied by a particular piece of information. Thus, an analogous approach is indexing the queries so that only the queries likely to be satisfied by the incoming piece of information are actually tested.

In this thesis we propose to design an algorithm that solves the matching problem on an XML context in an efficient way, by, based on the XML document structure, storing the queries in an optimized way and exploiting that storage to efficiently determine which queries are satisfied by the incoming event. This document is organized as follows: Section 1.2 will make a quick overview of the XPath language explaining the base syntax, providing examples and showing how the subscription language will fit in this context. Chapter 2 presents the related work, referring common techniques to handle the proposed problem, several relevant algorithms and how they use those common techniques, providing a critical analysis of their key features and basic paradigm. Chapter 3 details our proposed solution, explaining our choices and explaining our intentions in adapting the Le Subscribe algorithm to the XML filtering problem. Chapter 4 will show the results of the experiments we performed in order to evaluate the quality of our solution, measuring its performance in different scenarios. Chapter 5 will conclude this thesis, making a critical analysis of our work, the results achieved and possible future work to further enhance WFilter.

### 1.2 XPath

XPath is an XML path language that addresses parts of an XML document, it allows for the specification of patterns to be matched in the document. Since a XML document can be considered as a tree of nodes that might have attributes related to them, XPath allows for the
specification of paths in the tree, and, as a result, will output the nodes that match the specified path. These paths can either be absolute paths that start on the root node of the document tree or relative paths which start from a known location. A path query is composed of one or several location steps. The simplest form of location step is a node name along with the respective hierarchical operator.

In path queries the hierarchical relationship between nodes is represented by the parent-child ("/"/" operator, representing adjacent relationship, and by the ancestor-descendant ("//") operator, representing a descendant relationship separated by many-levels. There is also a special location step: the wildcard ("*"), which is used between hierarchical operators instead of normal node names and basically represents "any node", meaning it matches any node. So, as long as the document has one node in that depth, this location step is deemed as matched.

XPath expressions also allow filters. These filters are represented between "[" and "]" and are specified after a node-name. It indicates a condition that must be validated in order for that node to be matched. Such condition can either be a conditional predicate or another path. In case the filter is a conditional predicate, it is just a conditional expression that must be evaluated to true, it can be, either over the node names or over the attributes name. When addressing one of the nodes attribute we must use the special character Attribute-Annotation ("@") behind the attribute name and that is called an attribute expression. Otherwise, if the filter is another path we have what we call a nested path expression or twig expression, as the expression no longer represents only one path. When the filter’s expression is not started with a special operator such as Attribute-Annotation ("@"), it is considered to be a nested-child with the parent-child ("/"/) relationship by default.

One can aggregate the filters into each other by using aggregation operators such as AND and OR. Specifying an AND operator between filter expressions will force all of them to evaluate to true for the node to be accepted. For the OR operator, only one of filter expressions needs to evaluate to TRUE.

Example 1 /news/sports/football/ would address all nodes football, that have a parent node sports which in turn have a parent node news. In this context we would be selecting all news about football.

Example 2 /news/sports//transfer this would also address all nodes sports with a parent node news, but the result would be all transfer nodes that were descendant of a node sports. That will happen independently of the depth distance between sports and transfer, meaning that somewhere in the ancestors of transfer there had to be a node sports child of a node news.

Recall that each of these location steps can also contain one or more filters. These filters act as a way to refine the selected set of nodes. Filters are applied to the node corresponding to the location step where they are defined. Their evaluation must return TRUE in order for the path to continue to evaluate the next location step. Following the previous Example to exemplify some filter expressions:

Example 3 /news/sports//transfer [fee>1000]/player will return all players elements if the transfer they’ve been part of had a fee above 1000. The filter here is the fee >1000 (choosing only the transfer nodes with a child node that have a value above 1000). With this declaration, fee is also considered to have a parent/child relationship with transfer but it’s not part of the path being considered, it’s considered in a parallel way. This allows for the declaration of twig path queries.
Example 4 /news [@day=17] Here we would be selecting all news items and applying a filter to select between all of the news nodes found. That filter would be, having an attribute date that equals “17”. This is an Example of a conditional predicate filter. XPath supports equality operator (“=”), and inequality operators (“>”, “>=”, “<”, “<=”, “!=”).

Example 5 /bank/client [loan]/name would get all client nodes that had a child node called loan and would return their names. This is a twig expression since the query checks for two parallel paths, client/loan and client/name.

Example 6 /bank/client [loan<3000 AND balance<3000] Here we have two conditional filter expressions aggregated by an AND meaning both of them must be true.

Example 7 /bank/client[@balance<50 OR loan] this is an example of combining different filters, a conditional one and a path based filter. This will select the nodes client that have either a balance attribute lower than 50 or have a child node called loan.

While we are using XPath as the language to represent user subscriptions, the context here is slightly different. Instead of selecting paths, we are selecting documents, so, it’ll work as follows: A document matches a subscription if it contains at least one path that matches the subscription. This means that there is no need to output all the matching nodes, and only one match needs to be found. Also, we do not support nested path expressions in our subscriptions.
2.1 Overview

The problem of filtering XPath expressions with XML streams solutions can be divided into two main sub-problems: structural matching problem and attribute expression matching problem (filters). The structure matching problem consists in matching the structure of subscriptions (the location steps) with the structure (DOM structure) of the event to match. The attribute expression matching problem consists in verifying if the predicates composing the attribute expressions of the location steps of subscriptions are satisfied by the document respective match.

The main goal to be able to perform efficient matching is to find effective ways of indexing the subscription queries. Every solution uses a specific approach to represent the queries internally. A different query storage structure will enable different indexing mechanisms which is the core of efficient filtering. Common techniques to improve the matching performance are based on exploiting the following:

- **Query prefix commonality**: When dealing with several subscriptions it’s highly probable that many queries share the same starting nodes, especially the first nodes that (e.g. root node) are usually repeated across many queries. This is something some strategies take advantage of. By grouping all queries by prefix in some way, all queries with a certain prefix can be matched simultaneously, meaning the match is only computed once, improving time and space performance. The redundant work of computing the prefix matching is spared, this usually implies a significant reduction in the universe of possible solutions, so, we only need to match the remainder of the queries that had their prefix matched.

- **Query suffix commonality**: The information that truly concerns a user querying the filtering system is always the last location step in his query. Considering this, it’ll not be unusual for queries to share a suffix, since it’s also probable that many users will share interest in the same nodes. Taking this into account, just like prefix-based grouping, suffix-based grouping becomes an exploitable advantage in some cases, especially when dealing with bottom-up query matching (starting on the leaf nodes and going up).

- **Attribute expressions commonality**: Sometimes the attribute expressions in the subscription filters are equal between subscriptions. Taking that attribute expression commonality into account gives the advantage of only computing each unique attribute expression once, as all the other equal attribute expressions will have the same result.

- **Caching sub-results**: Sometimes, certain node patterns that do not match any subscription are common among several different events. In these cases, it is possible to store in a “cache” that a certain pattern will not match any query. So, instead of constantly
matching a part of that pattern to ultimately discard it, when that pattern appears, it’s immediately discarded.

- **Holistically match non-linear twigs**: When dealing with subscriptions with twig expressions, they require the matching of two or more paths. A common, simple technique, to deal with this, is to consider them individually as separated linear paths and apply an extra post processing step to check if both the linear paths belonging to a twig are satisfied. However, in some approaches it is possible to do the matching simultaneously. Thus skipping the post-processing step and matching them holistically.

In the context of the XML filtering problem, an XML document is an incoming piece of information and generates several internal events. All of the presented solutions are based on having a SAX parser that fires an event every time it finds the start of an element and the end of an element. Algorithms, depending on their scope, will handle the events useful to that particular scope, for example some of them might want to have access to an element’s attributes.

### 2.2 XFilter

XFilter (Altinel & Franklin, 2000) is the first algorithm that attempted to tackle the XML filtering problem. It implements techniques such as: exploiting commonality of a single location step for each subscription and pre-filter the subscriptions regarding elements not present in the document to more efficiently compute the matches.

In XFilter, each XPath expression is converted into a Finite-State Machine (FSM) by mapping the location steps of a subscription into machine states. Each location step of an XPath expression corresponds to a state in the associated FSM, with the last location step corresponding to the accepting state. Each FSM state is internally represented as a path node. A path node consists of a structure to hold location steps from a query with some metadata about those location steps, namely the subscription to which the path node belongs and some information about the position in the subscription and the required position in the document. An example of a query’s path nodes can be seen on Figure 2.1. There is also a pointer to the next path node referring to the same query.

When the queries contain predicate based filters they are stored in the path nodes in an appropriate field. In the case they are twig expressions, the storage is done by adding the nested path expression as a different subscription and setting up a pointer in the main subscription. However, sometimes it is not possible to determine at the time if the nested expression is satisfied. In this case, the algorithm continues assuming the expression as satisfied and marks it for post-processing.

The key feature of XFilter is the way it stores the FSM’s (each representing a XPath expression). It uses an index structure to store the path nodes for each subscription called the *query index*. The *query index* is a hash table with one entry for each unique element present in the set of all subscriptions. Each position of this hash table stores two lists: the candidate list and the waiting list, each of them containing path nodes (machine states). It is through this structure that the matches are computed. In the beginning all the subscriptions are stored in the *query index* in the following way: the first path node of each subscription is on the candidate list of that elements entry in the hash table. The remaining elements are stored in their respective
2.2. XFILTER

Figure 2.1: A query Q1 and its respective three path nodes

In Figure 2.2 we can see a query index with Q1, Q2 = { // b / * / c / d } and Q3 = { / */ a / c // d }.

So, as an event arrives, it is used to drive each FSM through their various transitions. A subscription is considered to be matched every time a FSM reaches an accepting state for that subscription (last path node is matched). This way, when a start tag event is fired by the parser, the query index is accessed and the path nodes on the candidate list of the element of the found tag are tested to check if they match the newfound element in terms of depth and position in the document. In the cases where the matching path node matches the newfound element and it was the last path node, the document is deemed to match that subscription, otherwise that subscription’s FSM has to move to the next state. This is done by promoting the next path node in the subscription from its waiting list to the respective candidate list, thus, making that state active. When the parser fires an end tag event, the respective path node is removed from its candidate list. This, allows for the algorithm to deal with different tags with the same name.

XFilter also proposes the following techniques as a way to improve performance:

**List Balancing** There is a high probability that the first nodes in the document appear on most queries, leading to an inefficient distribution of the path nodes on the hash positions. So, XFilter proposes the List Balancing solution, which checks the candidate lists for each path node and chooses the one with the less amount of elements. Then, that path node is considered the query “pivot” and will be the starting state for the query FSM. For this to work, a pre-condition attribute is added to the path node structure containing the elements before that one, that way, when a path node pivot is matched, before we move to the next state it is necessary to check the pre-condition containing the nodes before that one. To be able to check the pre-condition more efficiently, this approach keeps stack with the previous visited elements in the document.

**Prefiltering** This technique consists in not evaluating the queries that contain location steps relating to nodes that are not present in the document, as it is certain they will not be matched. It is performed before the matching is computed. So, it requires parsing the document one extra time before the regular solution is applied. However, when dealing with high amounts of subscriptions, this extra parse might remove a significant amount of subscriptions from consideration. Thus, not only, will XFilter recover the time of the extra parse, but actually achieve a significant performance improvement.
Prefiltering works by choosing an element from the subscription making that element the subscription key for that subscription using the same criteria as the list balancing. Then, it adds it to a table in a position based on the chosen element. When the document arrives, an occurrence table is constructed with all the elements in the document. They are stored in a hash table. For each element in the table, all the subscriptions that have that element as key element will be added to that position. After the aforementioned table is constructed, all subscriptions (i.e. whose key element was present on the document) will be scanned to check if all the elements of each subscription occur in the document (i.e. are present in the hash table).

2.3 YFilter

YFilter (Diao, Fischer, Franklin, & To, 2002) uses a single Non-Finite Automaton (NFA) to represent the combination of all path expressions present in the subscriptions. In this solution, the main technique is the NFA and as such, the computation of the matched subscriptions is done in the NFA. When the algorithm starts, the NFA is already defined but is only composed by the starting state. When a new subscription arrives in the system, it is decomposed into each of its location steps, very much like XFilter. Then, that subscription is added to the NFA. This will be done by following the NFA based on the location steps of the new subscription as follows:
Beginning on the starting state, if there is a transition to a state representing the same element as the current location step of the new subscription, then the transition is followed. If there is no transition, one is created, alongside with the respective state of the location step. When a state corresponding to the last location step of the new subscription is found, the subscription is added to that state list of matched subscriptions. Note that, if it’s the first subscription to be added to that state, the state is converted to an accepting state. Basically, considering a subscription as a regular expression, for each subscription a FSM that accepts such regular expression is constructed. The FSMs of all subscriptions are then merged into one, with states accepting the same symbol being merged into a single state.

It is important to refer that YFilter has a special way of dealing with wildcards and ancestor-descendant ("//") operators. Because unlike in XFilter, nodes are not marked with the depth and relative level they are expected to have. Instead, some special transitions are added to the NFA in order to represent this kind of operators as shown on Figure 2.3.

![Figure 2.3: NFA Transitions for basic location steps](image)

By constructing the NFA this way, the resulting automaton will be the result of merging all the subscriptions into a single structure and group all of the common location steps between them. Note that, states representing a "//Child" are added to the transition list of its father with an $\epsilon$-transition to an extra special state. An example of an NFA for several queries is shown in Figure 2.4.

The NFA is internally represented by pointers between states. Each state stores a hash table with all the possible transitions from that state for a certain input element. States also contain the list of subscription identifiers that are deemed to be matched if that state is reached. In the case the state contains a non-empty list it means it is an accepting state.

The NFA works supported on an auxiliary run-time stack that stores the states that are active and keeps track of the states previously processed. When an event occurs, the NFA’s initial state is pushed immediately into the runtime stack. As a start tag event is fired by the parser, the NFA follows all possible transitions from all the states that match that element. Because YFilter is using an NFA, several states can be active at the same time. To determine which path to follow, the element name is searched in each of the active states respective local hash table. If it’s found, then that transition destination state is pushed into the runtime stack. After all the regular transitions have been checked, the hash table is checked for $\epsilon$-transitions which will always be followed, and as such, are always pushed to the stack. On the next start tag event, the states on the runtime stack will be the active states of that event. The runtime stack is also useful for backtracking. When an end tag event happens it is necessary to backtrack.
YFilter offers support for predicates as an independent mechanism that interacts with the NFA to deal with value-based predicate filters in the subscriptions such as: `//client/payment[@value <= 400]`. YFilter presents two techniques: Inline and SP (Selection postponed) (Diao, Altinel, Franklin, Zhang, & Fischer, 2003).

**Inline**: With Inline, the predicate matching is done while the NFA is being executed. It requires adding to each state of the NFA the predicates required to migrate to the next state and the predicates are verified when deciding if a transition is to follow.

**Selection-Postponed**: As suggested by the name Selection-Postponed first performs the path matching as referred above, and then performs an extra post-processing step where it checks the predicates. When the NFA finds an accepting state, there is a pointer to the respective queries that are satisfied at that state and the respective group of predicates that must be matched for that query to actually be accepted. In order to do this, the NFA is required to also output the path matches when a subscription is deemed as matched, so then the path is traversed and the respective predicates are applied at the adequate nodes. As shown in (Diao et al., 2003), Inline is outperformed in almost all of the cases. Therefore, YFilter best results are the combined computing of the NFA for the structural matching and of SP for predicates.

### 2.4 FiST

FiST (Kwon, Rao, Moon, & Lee, 2005) is an Algorithm oriented to perform twig matching. It supports only structural matching, and it is based on a special internal representation of the subscriptions in order to improve performance when dealing with twig expressions. Since a XPath subscription selects a certain branch of data from an XML tree, it can be treated as a tree. Based on this observation, FiST encodes each subscription using a modified Prüfer sequence (Prüfer, 1918), which is basically a unique way of representing a tree as a string, called LPS. This is exemplified in Figure 2.5.
Initially, the LPS is constructed as if the subscription is a regular tree, not taking into account the difference between ancestor-descendant ("//") and parent-child ("/") edges. Then it is stored in a structure that is the internal representation of subscriptions called profile sequence. The profile sequence consists in a list where each position contains a label from the LPS and information about its position in the sequence and the relationship between nodes. Figure 2.6 shows the profile sequence for Q1. The Sym field is where the relationship between nodes is indicated. It can have the following values: parent-child relationship ("/"), ancestor-descendant relationship ("//"), branch node ("$"), root node ("#”). For parent-child or ancestor-descendant related nodes the relationship information is only stored on the descendant profile.

To efficiently store all the queries in the system, FiST uses an index called sequence index. This index is a hash table with each element present in the subscriptions as a key. In each position, it stores the element from the profile sequence of each subscription that expects that label. Considering the queries Q1 from Figure 2.6 and Q2: //B[E]/C with LPS(Q2) = E B C B, an example of the sequence index can be seen on Figure 2.7.

In this algorithm, the incoming document must also be converted to the same representation as the subscriptions. So an LPS of the document is generated to be compared with the LPS of the subscriptions. FiST uses a stack to support the matching process, the stack is filled as the document is parsed, generating an LPS sequence. It is also event based, meaning, when the document is parsed, every time a start tag event is fired the algorithm reacts by pushing that element into the stack, the matching process is only triggered when and end tag event is found. When this end tag event is fired, the algorithm pops the top element from the stack and performs a test using the sequence index. In case it succeeds, the next element from the profile
sequence whose element just matched the top of the stack, is added to the sequence index.

To test if a certain node matches an element of a profile sequence, FiST has to perform special tests to check if the relationship between the profile sequence label and the document label is verified based on the respective symbol. The stack, by containing the parsed nodes from root to the current node works not only as an auxiliary to perform such tests, but also offers some efficiency benefits. Another advantage of the stack is that at any given time, it stores all the segments of a sequence till the branch nodes, so, it can be used as a forward-checking buffer, to predict if that sequence part is worthy of being loaded into the sequence index.

It’s important to refer how FiST handles wildcards, as they work as a special case. When a wildcard appears in a non-branch node its sequence node is not generated and is instead added as an extra information field to the next element in the profile sequence (basically next node).

When the wildcard is present in branch nodes, a sequence node is generated for the wildcard and is processed alongside with regular node. When this node is computed, all of the eligible nodes to satisfy the parent-child (“/”) relationship or the ancestor-descendant (“//”) relationship are stored in the auxiliary branch structure as possible matches.

As shown in (Rao & Moon, 2004), subsequence-based filtering can produce false matches as a result. To handle this, FiST applies a post-processing refinement step. For this to work, a special processing is required for branch nodes beforehand. In this case, FiST stores in the branch node structure the number of the document node that matched successfully that branch node. After the subsequence matching phase returns a superset of query candidates. All of them have their branch sets examined as to finally compute the true matches.

2.5 pFiST

One of the most recently presented solutions for XML filtering is pFiST (Kwon, Rao, Moon, & Lee, 2008). It’s an extension to the FiST algorithm, presented in Section 2.4, by the same authors. Since FiST is only a structure matching algorithm with focus on twig matching efficiency, the purpose of pFiST is to extend FiST to support a fairly complete subset of the operations present in the XPath language related to predicate-based filters.
2.6. AFilter

The way predicates are handled by pFiST depends on the operator present in the predicate. The predicates are tested during the structure matching phase, instead of requiring a post-processing step to verify if all the nodes that passed the structure matching phase verify the respective value-based predicates. In the context of pFiST this comes as an advantage, since FiST is able to compute twig patterns, the evaluation of predicates at the same time as structure matching allows to discard other predicate tests and respective structure matches in the twig.

pFiST supports the following predicate operators: equality operator (=), non-equality operator (“<”, “>”, “<=”, “>=”), conjunction (AND) and disjunction (OR) operators can be specified in the twig patterns. pFiST applies different strategies to match different predicate operators. While the equality operator is equivalent in the documents to an extra child node, non-equality operators require a different filtering approach. Because of this, when handling an equality operator the basic approach of this algorithm is to treat the predicate as if it is an extra sequence node. The matching document twig will have that value, so, we can consider the equality predicate as a node and add it to the LPS representation of the subscription containing the equality predicate. By doing this, the predicate matching for equality predicates is done during structure matching. The former approach cannot be used for non-equality predicates due to the operator not being representable in the LPS. The solution proposed by pFiST in this case is to add additional fields to the sequence node that is parent of the predicate. The extra fields are a 3-tuple containing the operator, the value and the type of data being handled.

For FiST to be extended into pFiST an extension is required to the SAX event handlers, since FiST didn’t consider predicate matching, the attributes of the elements of matching document didn’t generate events. In pFiST, the attributes must generate events since they need to be handled for predicate matching. As stated above, pFiST also supports operators between each of the value-based filter predicates. To deal with these OR and AND statements, pFiST works by decomposing them in the following way:

- **AND** Having predicates separated by AND statement means all of them must evaluate to true, so, it’s equivalent to have all those predicates in a consecutive declaration. Hence pFiST solution is to convert the predicates separated by the AND operator to successive predicates that can be handled as many individual predicates.

- **OR** In the Or case, the subscription is considered satisfied if at least one of the predicates is satisfied. Because of this, pFiST splits a single query into two different queries each of them containing all of the steps of the original but only one of the predicates. For instance, assume the following query q on the system \( Q = \{a/b[c = 10 \text{ OR } d = 7]\} \), since only one of the predicates needs to be matched we can rewrite q as two queries \( Q_1 = \{a/b[c = 10]\} \) and \( Q_2 = \{a/b[d = 7]\} \).

2.6 AFilter

Even though AFilter (Candan, Hsiung, Chen, Tatemura, & Agrawal, 2006a) is included in the category of stack-based approaches, meaning the actual computation of the matches is done in a stack-like structure, its complexity ranges far beyond that single data structure. This algorithm consists in many very different structures working together to perform a transformation in the data stored at each step that in the end, makes the matching faster.
In this solution, the subscriptions are first stored in a data structure that works solely as a fingerprint of the system, containing only a convenient representation of subscriptions, meaning no processing directly related to matching will be done. This structure is called the Pattern-view and is composed by three lesser structures: Axisview, the PRlabel-tree and the SFlabel-tree.

The Axisview is the core of the Pattern-view. It is here that all the subscriptions in the system are stored. This is done under the form of a directed graph where each node represents a location step from at least one subscription (for each unique element present in all subscriptions there will be a node in the graph) and each edge represents an axis relation specified in a query. This way, the graph is an actual fingerprint of the set of all subscriptions. Here, the relationship between nodes in the graph is represented backwards. For example: a subscription $Q = a/b$ would produce an edge from $b$ to $a$. Figure 2.8 shows the Axisview data structure when considering the following subscriptions: $Q_1 = //d//a//b$, $Q_2 = //a//b//a//b$, $Q_3 = //a//b//c$ and $Q_4 = /a/*//c$. A system with this queries would origin the graph shown in Figure 2.8.

![Figure 2.8: Axisview for Q1,Q2,Q3 and Q4](image)

Axisview further stores information in the edges of the graph under the form of assertions. The stored information is based on the kind of relationship between both location steps. If the relationship between two location steps is the parent-child ("/")), the edge will be associated with a "\(|\)" symbol. If it is an ancestor-descendant ("//") relationship, it will be associated with the "\(||\)". In the cases where the source node is the last location step in a subscription the symbols "\(|\)" and "\(||\)" will be replaced by "\(↑\)" and "\(↑↑\)" respectively.

The PRlabel-tree and SFlabel-tree are optional structures, which means that AFilter can still compute matches without them, sacrificing time efficiency for space efficiency if needed. Both of these structures are tries and act as a way to group the subscriptions based on their prefixes and suffixes respectively. The goal of the tree is to avoid having to perform redundant computations by enabling sharing of sub results across queries. In these trie structures, each node is a label from a subscription and each branch represents a prefix for PRlabel-tree and a suffix for SFlabel-tree. An example of those tries can be seen on Figures 2.9 and 2.10 respectively.

Summing up the above, the Pattern-view is a fingerprint of the subscriptions in the system. It stores all the subscriptions in Axisview and organizes common prefixes and suffixes in PRlabel-tree and SFlabel-tree respectively.

As stated before, in Pattern-view there is no matching processing. Basically, AFilter uses the
2.6. **AFILTER**

Cleverly stored information in Patternview to compute the matches in another, more convenient structure: the Stackbranch. This Stackbranch structure consists in a collection of stacks, one stack for each node in the Axisview. When a start tag event is fired by the parser, a new stack object is pushed to the stack of that tag element. A stack object stores the index of the element, depth and one pointer for each relation that node has on Axisview. If that stack has at least one node, the pointers point to the top of the destination (on Axisview) node stack, otherwise they will be null pointers. If there are queries with wildcards, then, a special extra stack representing the "*" will be used. In this case, every time a stack object is pushed into a stack, that object is also pushed into the wildcard ("*"") stack. When an end tag event is fired, the topmost element of the respective element stack is deleted including its pointers. Figure 2.8 shows an example of a Stackbranch structure after parsing `<a><d><a><b>` using the Axisview shown on Figure 2.8. When a new stack object is added and the edges in Axisview are read to generate pointers, it is possible to determine the subscriptions matched since the last location step in the subscription the edge will contain an assertion with either the "↑" or the "↑↑" symbol.

AFilter also uses a caching mechanism, the PRcache. It caches the results obtained or lack of when traversing a pointer with a certain assertion from one stack object to another. This way, to avoid traversing pointers that lead to no matches, if the result of traversing a pointer with a certain assertion is already cached, only a cache lookup will be needed.

When PRlabel-tree is in use, the PRcache stores the results obtained by prefix group, as all the subscriptions in the same prefix group can obviously share the same cache results.

When using SFlabel-tree, the Axisview edges instead of including the subscriptions, include the suffix clusters that include that transition. Basically, the subscriptions are clustered by suffix
and then the algorithm works with the clusters (even if some clusters only have one subscription).

### 2.7 XFiS

XFiS (Antonellis & Makris, 2008) is a sequence based algorithm focused on efficiently computing twigs in subscriptions. Currently, the most efficient techniques to perform twig matching is to holistically determine them. Although, when performing holistic matching, false positives are produced, which will still require a post-processing step to discard them. XFiS proposal is a new way to perform holistic matching of twig patterns with the ability to discard false positives on-the-fly as they arise. The technique consists in representing the location steps of the XPath subscriptions as strings, very much like FiST, but with a slightly different representation that contains more information.

For the XFiS string representation to work, it requires every node to be represented only by a single character. Because of this, it requires a pre-processing step where a subsystem assigns a one-character unique char label to each distinct tag and stores the relation between them. A tag is always represented by its char label during the execution of the algorithm. Note that, since it generates a label only for distinct tags it does not support tag recursion in the data.

![Figure 2.12: A Document T converted into a TSS with the support of a stack](image)

As subscriptions are added to the system their TSS is determined and they are added to an index called `twigpositions`. This index consists of a hash table where the keys are all the labels in the system and the values are all the TSS that currently expect that element. As a document’s TSS is being generated, when a document element label is determined, that character is sent to the matching algorithm. Then, there is a lookup on `twigpositions` to get the TSS whose current position is equal do the document label. To keep track of which label a subscription is expecting a structure called `twigpointers` that has a one-to-one relation between each subscription and an integer indicating which is the position of the label currently expected.
2.8. LE SUBSCRIBE

LeSubscribe is a matching solution for value-based Publish/Subscribe systems presented in (Pereira, 2002). Even though it is not a XML filtering algorithm we will provide a detailed explanation as it represents the foundation of our solution. In the context of value-based Publish/Subscribe systems, events are represented as sets of attribute-value pairs. Subscriptions are conjunctions of predicates over attributes. Available predicate operators are: equals ("="), greater-than (">"), less-than ("<"), greater-than-or-equal (">=") and less-than-or-equal ("<=") and the is-kind-of for hierarchical data types. Although there are several versions of this matching algorithm, we will only present the Single Access Predicate Clustering Algorithm, since we will base our solution in this approach. LeSubscribe is a fairly complex and complete algorithm so, we will start by detailing the data structures involved in the processing before explaining its behavior. The data structures used in this solution are as follows:

**Predicate Index Set:** The main purpose of this structure is to enhance the processing speed of the predicates satisfied by a given event. This index set is composed of several Predicate families. Each family consists in grouping all the predicates containing conditions over the same attribute and with the same operator. So, we have five different possible predicate families for each attribute: equals ("="), greater-than (">"), less-than (<), greater-than-or-equal (>=) and less-than-or-equal (<=). Then, to provide fast access to those families an index is employed, where the access keys are the values used in the predicates of that family. It’s important to note that the type of the indexes depends on the type of operator associated with the indexed family. For example, a hash index can be used for equality predicates, but it’s of no use for non-equality predicates. Non-equality predicates can be stored in binary trees. It’s important to refer, that in LeSubscribe the predicates of a subscription are sorted by decreasing order of
selectivity, meaning the chosen predicate is the most selective, causing a greater reduction in the candidate solution universe.

**Predicate Vector:** This is a vector that contains one position for each existing predicate in the entire set of subscriptions. Each predicate is represented in the system by a unique identifier, allowing instant access to the corresponding position in the vector. This vector keeps the result of predicate evaluation for the matching event. If a predicate is matched by the event, the corresponding position on the predicate vector is set to 1, otherwise it is set to 0. The idea is that each predicate is just evaluated once per event to match. The position corresponding to a vector works as an internal representation of the predicate, meaning every time a predicate is stored in another structure. What is actually stored is a reference to the respective position in the Predicate Vector, meaning that the predicates are stored once in the corresponding predicate family and evaluated once per incoming event.

**Subscription Predicate Table:** As suggested by the name, this is a table that contains the set of predicates belonging to each subscription. In LeSubscribe, each subscription is associated with a unique identifier for instant access to the corresponding position.

**Subscription Clusters:** In LeSubscribe, subscriptions are grouped together in clusters based on some of their common predicates and their size (number of predicates). The set of common predicates that were used to cluster those subscriptions together are designated as "access predicate" of that cluster. In the single access predicate clustering the access predicate is composed of a single predicate. Each cluster is basically a table with a row with all the subscription identifiers presented in the cluster and all the other rows contain the predicates for those subscriptions. Therefore, the table contains in each column: the top position with the subscription and all the others positions in the column have one predicate of that subscription. For instance, assuming S0 was the first subscription stored in a cluster C, the identifier os S0 and the full set of predicates of S0 minus its access predicate compose the first column of the table.

**Cluster Vector:** A data structure that holds a relation between the access predicates and the respective Subscription Cluster Lists. In this version of the matching algorithm, the access predicate has the same identifier of its associated predicate. This unique identifier represents a position in the vector.

**Subscription Cluster Lists Index:** A simple way to prevent the processing of unnecessary subscriptions is to only compute the subscriptions in clusters whose access predicate is satisfied. Because of this, a quick way to determine which access predicates are matched by an event is needed, so, an index is used to store the references to the access predicates and return all the access predicates satisfied by a set of satisfied predicates. It returns Cluster Vector positions as a result.

**How it works:** This algorithm is divided in three phases: Initialization and predicate matching, Computation of the candidate clusters and Process matched Subscriptions.

The first step, initialization and predicate matching, initializes the predicate vector with
2.9. PREDICATE FILTER

all entries set to 0 and determines which of the predicates are matched by the incoming event, through the predicate index set. Then, each position in the predicate vector corresponding to a verified predicate is set to 1.

The Computation of the Candidate Clusters discovers, based on the set of verified predicates, which cluster Lists have their access predicate verified by the satisfied predicates computed in the first step.

Determining matched subscriptions is the third and last step. In this step, each subscription in the candidate clusters have each of their predicates (except the access predicate) checked in the predicate vector. If the entry is 0, the matching process ends and the subscription is considered not matched. If it is one, the predicate was matched, so the next predicate of that cluster has to be checked. In the cases where the matched predicate is the last one, the subscription is considered matched.

As already stated, there are several approaches proposed for Le Subscribe, when regarding the method to generate the clusters we will base our approach in the “Single access predicate clustering algorithm”. This algorithm stores the subscriptions in clusters based on an access predicate composed of a single predicate, the first predicate of the subscription.

2.9 Predicate Filter

The main concept of Predicate Filter (Hou & Jacobsen, 2006) approach is to store the position of each tag of an XML Document and encoding the relative position expected of the XPath expressions in a storage structure called predicate. Then convert a XPath expression into an ordered set of predicates that can be evaluated using the previously stored information. These predicates are represented by a 3-tuple \((\text{attribute, operator, value})\) and are sequenced in a set by being separated by the \(\rightarrow\) operator, indicating the sequence between the predicates in that set.

In order to simplify the predicate sets, this solution, considers each path in the XML document from the root node to a leaf node a separate input, and as such, breaks down each input document into several, simpler inputs. Due to this simplification, the simplified information contained in an input can be represented as a set of \((\text{attribute, value})\) pairs where the attribute consists of the XML node name and the value consists of the position in that path. So, each path is composed by the length of that path and a set of \((\text{attributes, value})\) pairs, where \text{attribute} field can contain other \((\text{attributes, value})\) pairs in the case where there are XML attributes, for example: \((a(x, 6), 3)\) represents a node \(a\) at depth 3 with an attribute \(x\) that is equal to 6. Each of these pairs come accompanied by an annotation indicating the occurrence number of that node in the current path.

Predicate Filter attempts to explore overlaps in different XPath expressions by storing and evaluating unique predicates only once and using references to that predicate in the encoded expressions. The are four kinds of predicates:

**Absolute** Represents the depth at which an XML node should appear. This can be represented through the use of the \(=\) operator or the \(\geq\) operator. An example of this is \((a, \geq, 3)\) which would mean that \(a\) is expected to be found at least at depth 3 of a path.

\(^1\)In this case, the attribute is not necessarily just a name. Sometimes the attribute field is actually a set of other tuples or a value that has to be computed during execution.
**Relative** Indicates the distance between two nodes in a path. It is represented using the $\geq$ or $=$ operators and another predicate representing the distance between node $a$ and $b$. For example, $(d(a, b), =, 2)$ means that node $a$ should appear two location steps before the node $b$.

**End-of-path** Used for XPath expressions that end in wildcards. This predicate indicates the number of nodes that should appear after the last non-wildcard node. This number is equal to the number of wildcards in the ending of the expression.

**Length-of-expression** In the case a XPath Expression is composed only of wildcards, it is represented by this predicate showing that an expression should have size equal to an input path.

In this solution, each unique predicate is assigned a unique predicate id and then that unique id is indexed using the predicate as key. This group of indexes is called the predicate index, it is detailed below and is shown on Figure 2.14.

In the case of absolute predicates the indexing is done based on both the hash value of their node name. This index takes the hash value of the node name as input and returns two arrays, one for each operator containing in each position, all the possible values this predicate can take paired with the predicate id respective to that value.

When dealing with relative predicates, the same technique is used but with an additional level of indirection. There is basically an additional hash table for the first node that leads to a hash for the second node that works exactly in the same way as the one for absolute predicates. Basically it’s a 2-dimension index that receives a pair of nodes.

The two remaining predicates kinds are indexed in a simpler way: The end-of-path predicate is stored in a hash table that uses the node name hash value as key and contains an array with all the predicates of this kind. For the predicate length-of-expression only an array is used as both length and the operator $\geq$ are constants.

This algorithm uses a predicate vector to index the results of the evaluation of the predicates. There is a direct mapping between the predicate id and the position in the vector. So, through the use of the aforementioned predicate index, a node can be mapped into the predicate ids of the predicates it satisfied. Thus, marking a predicate as satisfied will only consist in accessing the predicate vector for those ids and mark the respective entry as 1.

This matching algorithm has two phases: The predicate matching stage and the XPath expression matching stage. The first one determines all the predicates that are individually verified and the second computes which XPath expressions composed of several predicates are matched based on the information determined in the first stage.

The first phase consists in parsing an already isolated path from root to leaf and computing the predicates satisfied by the current matching event (path). These matches are represented by a pair $(depth, occurrence\_number)$ where depth represents the position in the path where the node appeared and occurrence\_number indicates which occurrence of that node appeared at that depth$^2$.

The second phase computes which of the XPath expressions, now converted into predicates that store the depth and occurrence number, are verified. To do this, it must determine if the order in which the predicates appeared corresponds to the order specified in the predicate.

$^2$Recall that a single path can have repeated tags and they must be handled differently.
2.9. PREDICATE FILTER

Figure 2.14: The predicate index

set (by the operator $\rightarrow$) representing a XPathExpression. This verification is a constraint satisfaction problem and Predicate Filter uses another algorithm, presented in (Dechter & Frost, 1999), to solve this problem.

To further optimize their solution, they group XPath expressions in clusters based on a single common predicate which they call access predicate, in this case they choose to group subscriptions by their first predicate. Due to this, the clusters are evaluated only when that access predicate is verified. When verifying the subscriptions inside the clusters, the first predicate of each subscription can be assumed as matched and can be skipped since, for a cluster to have its subscriptions evaluated, its respective access predicate (first predicate) must have already been verified.

Experiments shown that this solution has a very similar performance to YFilter. In some cases it performs slightly faster, while in others, slightly slower. However, this algorithm could have achieved a better performance through the choice of a more selective access predicate. Clustering the subscriptions is highly dependent on having a selective access predicate. By using the first predicate, they’ve achieved subpar performance when compared to other predicates regarding deeper nodes as usually the first predicate is not very selective.
2.10 Summary

As stated before, the key element in the matching solution is storing and handling the subscriptions in an efficient way. Most strategies base themselves in common paradigms, so, we will classify them in the following categories: automata-based, stack-based and sequence-based. Next we describe the paradigm behind each category.

2.10.0.1 Automata-based

This approach is based on Automatons, either deterministic or non-deterministic. The paradigm in this approach is to store the queries as state-machines with each location step representing a state in the state-machine and use the document nodes as input for the machine. The concept here is using an automaton that uses the nodes in the subscriptions as the language set and has a set of transitions that represent their location steps. Then, as the document is parsed, it generates an event for each tag it finds. Such event will be used as input for the automaton. This way, after all the input is consumed the document is considered to match the subscription if the automaton reaches an acceptance state.

It was the first proposed approach to this problem, being the subscription storage technique of the matching algorithm XFilter, the first matching solution in the context of XML filtering systems. In this solution, each subscription is stored as an FSA (Finite-State-Machine), and, as the input streams in, the state machines move their states accordingly. In the end, the FSA's that are in an accepting state are considered matching subscriptions. Since XFilter, indexes the subscriptions and supports filters with predicates and with additional paths, it’s considered a very complete solution. However, most of the techniques employed by XFilter fail to explore a significant amount of the commonalities in queries. In XFilter index, queries are grouped based on the next node required for their FSA to proceed, so, basically the queries that expect the same node, are grouped together. Even though, the commonality is exploited for every single node in the queries, it’s still only one node at a time. XFilter also lacks clever processing for predicate-based filters. Instead, it just processes them as they show up, which is basically a Naive approach slightly enhanced by their structural matching. When handling twig expressions, the approach is slightly better, but still requires post-processing, and sometimes computes the entire structure matching for structures that do not satisfy the filter.

The next step on automata-based solutions is YFilter, which instead of using a FSM for each query uses a single FSM representing all the queries, but in this case, the FSM is a NFA (Non-deterministic Finite Automaton) containing several accepting states (for each individual subscriptions). This approach is more successful in exploring the commonality in the queries, in fact, it exploits all the structural commonality, since it uses a structure that represents all the queries merged. YFilter, also supports predicates in filter expressions, but, do not exploring predicate commonality. It is also an approach close to a naive solution when dealing with attribute expressions matching. Another possible problem with this algorithm is a combination of deep documents with recursive data, as this will cause the same states to be visited multiple times causing an exponential growth in the amount of active states and states stored in the runtime stack. Nonetheless, YFilter still presents itself as one of the most complete and well performing solutions.
2.10. SUMMARY

2.10.0.2 Sequence-based

This type of approach is based on considering the matching problem as string matching problems. It requires the conversion of the subscriptions into a string format (or just treat location steps as strings).

There are usually two kinds of sequence-based techniques: The first one applies structures already proven efficient for regular string matching as support structures to enhance matching performance. This is the base technique of Xtrie (Chan, Felber, Garofalakis, & Rastogi, 2002). This algorithm consists in using a trie structure (Knuth, 1998) to store the subscriptions represented as substrings. Even though it has a special way to deal with twig expressions, it still matches them as linear paths and requires a post-processing step. However, since the trie contains the subscriptions scattered as substrings it exploits commonality between them, making Xtrie perform better in terms of space and time when compared to XFilter, the first solution of the automata-based paradigm.

The second kind of sequence-based technique is to convert the subscriptions into a different string format that provides another way of computing the matches. FiST takes such approach, by converting both the document and the subscriptions, into a Labeled Sequence based on the Prüfer sequences. This, as shown in PRIX (Rao & Moon, 2004), allows for a very efficient processing of the twigs matched.

FiST is very focused on enhancing twig matching. So, it only includes structural matching and uses an indexing scheme very close to XFilter. Yet, by adopting this implementation, FiST can perform holistic twig matching, meaning twig expressions are matched simultaneously. However, it should be noted that FiST’s concrete approach produces false positives, so even with lower cost it still requires a post-processing step to eliminate them. pFiST, the extended form of FiST supports predicate evaluation in a form also similar to XFilter. However, because the processing is done on a bottom-up fashion way, the pre-processing of predicates allows pFiST to prevent many redundant computations.

XFiS is another system that converts the input into another format. Like FiST it’s also based on converting the subscriptions and the documents into a Prüfer sequence-based string. However, by adding more data regarding the structure to the strings, it is able to match twigs holistically and cleverly discard false positives. Thus, skipping the post-processing step. It also uses an indexing structure similar to FiST but uses a labeling system for nodes, an extra structure to achieve a more complete representation being able to avoid the aforementioned post-processing step. Because of the labeling system having to assign one character labels to each distinct tag for the algorithm to work, it does account for tag recursion in data, which is a serious limitation, as, for example in (Choi, 2002) 35 of the 60 DTDs include recursive data. XFiS also does not support predicate-based filters.

Also following this approach is NETS (Salloum & Tsotras, 2009). This algorithm uses NETS (Node Encoded Tree Sequences) as a representation for both the document and the subscriptions. This form of representing the data also allows NETS to perform a matching that permits doing the holistic match and discard false positives immediately. Despite performing slightly better than XFiS, it is follows nearly the same approach as XFiS. So, opted to only describe one of them.
2.10.0.3 Stack-based

Even though Stacks are not suited for efficient look-ups of data or matching, they are attractive because of their ability to store data based on the way that it is sequenced. This allows for a very simple mechanism of backtrack. Also, since stacks are simple structures, they can always be used as auxiliary structures (YFilter, an automata-based algorithm uses this). Because of this, matching solutions that use stacks as their main structures are rare and often complement them with other more complex structures.

TwigM (Chen, Davidson, & Zheng, 2006) is such an algorithm, although it’s suited to a slightly different problem: efficiently match a single XPath query with streaming XML data. While the solution of this problem can be looked on the perspective of efficiently matching a single subscription with many incoming documents, in this context, it lacks the ability to efficiently handle a high number of subscriptions, since subscriptions are considered in an isolated way and commonality between them is not explored.

AFilter is another stack-based algorithm. Even though its main structure is a group of stacks, it successfully takes advantage of combining simultaneous prefix commonality and suffix commonality based storage. However, several other structures are used, and a pointer based technique that connects certain elements between stacks is also present. The key feature here, is that it uses a structure that acts as a fingerprint of all the queries in a single representation. Then, instead of traversing that structure, AFilter uses this fingerprint to be able to construct a more efficient structure that allows better caching and easier matching. Another unique characteristic of AFilter is the fact that is prepared to work without some of its structures, sacrificing speed performance to use less space if needed. Despite not supporting predicate-based filters, it’s probably the most complete structure matching solution, presenting specific techniques to more efficiently combine prefix and suffix exploration and uses caching techniques to store intermediate computations.

2.10.0.4 Pushdown Approaches

This kind of approach is usually used in algorithms supporting a very large or the entire set of XPath Expressions. Usually their main structure is a pushdown automata (Hopcroft, Motwani, & Ullman, 2000) or a slightly modified version of it.

Because the pushdown automata includes a stack that can influence the state transition, they can combine some the characteristics of the automata-based approach (ability to easily represent document trees) with stack-based approaches (easy backtracking and sequential storage). A pushdown automata requires very complex algorithms that usually result in an implementation that is rarely able to accommodate optimizations. However, with these algorithms we are usually able to fully encompass the entire set of XPath expressions. An example of this approach is XSQ (Candan, Hsiung, Chen, Tatemura, & Agrawal, 2006b) which is based on pushdown transducer and is suited for streaming. It also allows for a very complete set of XPath expressions. However, it lacks the ability to handle queries with wildcards. Also, since it’s focused on efficiently applying a single query to streaming XML data it does not efficiently handle multiple subscriptions. However, the authors intend to expand it to handle more subscriptions by concealing it with an automaton-based approach.

The most relevant algorithm in this category is XPush (Gupta, 2003). XPush focus is on exploring commonality between attribute expressions. However, it does not support adding
queries after the pushdown automata has been constructed and so it is not desirable for a
dynamic context, such as publish/subscribe system. Also, even though it supports twig ex-
pressions, they cause the amount of states in the automaton to grow very fast. Overall when
compared to other approaches, its time performance is not significant, even when using aver-
age/small sized (4/5 location steps) subscriptions.

The following table summarizes the fundamental XML matching algorithms:

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Type</th>
<th>Name</th>
<th>Prefix Commonality</th>
<th>Suffix Commonality</th>
<th>Conditional Filters</th>
<th>Enhanced Twig Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automata</td>
<td>XFilter</td>
<td>Weak</td>
<td>Weak</td>
<td>Yes</td>
<td>Very weak</td>
<td></td>
</tr>
<tr>
<td></td>
<td>YFilter</td>
<td>Very strong</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Sequence</td>
<td>FiST</td>
<td>Weak</td>
<td>Weak</td>
<td>No</td>
<td>Strong</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Xtrie</td>
<td>Average</td>
<td>Average</td>
<td>No</td>
<td>Weak</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XFis</td>
<td>Weak</td>
<td>Weak</td>
<td>No</td>
<td>Very Strong</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pFiST</td>
<td>Weak</td>
<td>Weak</td>
<td>Yes</td>
<td>Strong</td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>TwigM</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AFilter</td>
<td>Strong</td>
<td>Strong</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Pushdown</td>
<td>Xpush</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XSQ</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>Predicate Filter</td>
<td>Strong</td>
<td>Strong</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Summary of the key components of the matching algorithms
WFilter
WFILTER MATCHING SOLUTION

WFILTER is an algorithm suited to perform efficient XML filtering in a publish/subscribe XML context. In this context, events are represented in the form of XML documents and the subscriptions are expressed in the form of an XML path query language. In this case, we will be using the commonly used query language XPath. The XPath language is a very expressive language and allows users to express their interests with a high accuracy. However, highly expressive languages make the process of matching events against subscriptions more complex.

In order to efficiently cope with the aforementioned complexity we based our solution on the LeSubscribe unidimensional solution presented in Section 2.7. LeSubscribe employs indexing techniques based on clustering that fully explore subscription commonality between predicates in a value-based publish-subscribe system. Also, since the key technique to efficiently index XPath expressions in a XML filtering environment is to explore the commonality between those expressions, we had the goal of taking advantage of the efficient processing capacity of the LeSubscribe indexing techniques by adapting its solution to the XML Filtering environment.

Since the main purpose of a publish/subscribe XML based system is to provide a very high expressiveness concerning both the subscriptions and publication languages, we aimed to encompass the most complete subset of the XPath language we could achieve. Therefore, we support the following subset $P(\text{//}/\text{*}[0])$. This includes all the structural operators and both equality and inequality operators predicates regarding attributes. However, we do not support nested path expressions, which are expressions that address more than one branch in a single query.

Due to the increased complexity of the XML compared to attribute-value pairs, adapting LeSubscribe could be done by one of two approaches. Either, enhance the matching solution which would inevitably increase its complexity and hinder its efficiency or adapt the events to a format closer to the original format used by LeSubscribe, where events are represented as a set of attribute-value pairs. We opted to decrease the complexity of each event in order to simplify the matching problem.

Even though the simplification of events reduces the complexity of incoming information, this is not enough. Because LeSubscribe indexing techniques work with subscriptions consisting of conjunctions of predicates, we also needed to find a way to convert the subscriptions to a suitable format. Therefore, we also found a way to convert the subscriptions into a set of predicates.

By fully transforming the input, we changed the format of both events and subscriptions. This, allows us to successfully adapt the indexing techniques used in LeSubscribe to an XML filtering context and be able to retain the strong predicate commonality exploitation used by LeSubscribe, where each distinct predicate is treated as a single entity.
3.1 Event Representation

We will begin by detailing how the events are transformed into simpler events and what is an event in our algorithm. On publish/subscribe value-based systems each event consists in a set of pairs containing a name and a value, in the form of \{\langle attribute_1, value_1 \rangle; \ldots; \langle attribute_n, attribute_n \rangle\}. An example of this could be the event \{(a, 5); (b, 6)\}, that represents an event with two attributes, a and b, where attribute a has the value 5 and attribute b has the value 6. As previously stated, LeSubscribe is suited for value-base processing and consequently we intended to break down the XML documents which we will call external events into several smaller internal events that could be represented as a set of attribute-value pairs. For instance, the external event shown in Figure 3.1 can be represented by three internal events e1, e2 and e3.

In order to do this transformation, we implemented a component that parsed each internal event and converts it into the corresponding set of internal events and then submit each internal event to the component that implements the matching algorithm. Each internal event is then represented as a set of attribute-value pairs with the format \{\langle nodename_1, depth_1 \rangle; \ldots; \langle nodename_n, depth_n \rangle\}, where nodename_i appeared on the internal event at the depth depth_i.

Figure 3.1: A XML Document broke down into three paths: p1, p2, p3

Example 10. For example, the internal events corresponding to the paths shown in figure 3.1 are the following: e1 = \{(a, 1); (b, 2); (c, 3)\}, e2 = \{(a, 1); (b, 2); (d, 3)\}, e3 = \{(a, 1); (x, 2)\}.

3.2 Subscriptions

Subscriptions in our XML filtering context are represented in XPath. This representation contains information regarding the XML node names that must be present in a path and also their relative position in that path. To use the Le Subscribe techniques we need to convert XPath expressions into a conjunction of predicates. In this case, since our subscriptions consist of XPath expressions addressing only a single path in the XML document we only need to take into account the XML node names and their respective position in the path (depth) to achieve a full representation of the subscriptions.
The obvious approach was to convert each XPath expression directly into a conjunction of predicates over the position of the nodes in their XML path. This would lead to the following predicates just to deal with the structural part of the subscriptions:

- **root** \( x (//x) \rightarrow \text{pos}_x = 0 \)
- **exists** \( y (//y) \rightarrow \text{pos}_y \)
- **child of** \( x (//x/y) \rightarrow \text{pos}_y = \text{pos}_x + 1 \)
- **descendant of** \( x (//x/y) \rightarrow \text{pos}_y > \text{pos}_x \)
- **wildcard offset** \( (//x/ * /y) \rightarrow \text{pos}_y = \text{pos}_x + 2 \)

For example, the subscription \( s_1 = \{ //a/b//c \} \) would be represented as \( s_1 = [ \text{pos}_a = 0 \&\& \text{pos}_b = \text{pos}_a + 1 \&\& \text{pos}_c > \text{pos}_b ] \).

However, with this approach, each conditional expression (besides the root) is dependent on two predicates, causing each subscription to be represented by very high amount of predicates, making the subscriptions complex and the predicates harder to index. Not only this, but we would also need to keep a predicate for path length to deal with the case where a subscription ends with a wildcard. Moreover, this approach does not consider the possibility of recursion in an XML document\(^1\). When dealing with various nodes with the same name at different depths, there is a risk of having false positives when dealing with the ancestor-descendant("//") operator, to properly handle this, one must add more predicates to distinguish between which of the nodes with the same name matched a certain predicate, either in an explicit or in an implicit way.

As shown above, this representation would lead to a very high amount of predicates and some of the predicates would not be easily indexable since they involve a condition over two attributes. This would imply a high cost to determine the predicates satisfied by an event. Therefore, we decided to follow a different approach.

Instead of the previously shown approach we’ve represented each subscription as a set of predicates and operators. Basically, this predicates only store the occurrences of a node in the current path and the operators take care of computing the relation between those predicates:

- **root** \( x (//x) \) This predicate stores if node \( x \) was or not the root of the current event.
- **pos** \( x (//y/x) \) This predicate stores all the positions in the path at which the node \( x \) appeared in the current event.
- **pos at** \( x (//x[@attr = 7]) \) This predicate stores the positions in the path at which the condition applied over the attribute \( attr \) of node \( x \) was satisfied.

The operators are used to represent the operator already present in the subscription. They compute the relation between two nodes based on the node positions stored in the previously shown predicates. Those operators are the following:

- **parent operator** \( (../b/c) \rightarrow \text{pos}_b \) **parent** \( \text{pos}_c \) returns the set of positions where \( c \) appears immediately after \( b \).
- **ancestor operator** \( (../b//c) \rightarrow \text{pos}_b \) **ancestor** \( \text{pos}_c \) returns the set of positions where \( c \) appears after \( b \).

\(^1\)Recursion in an XML Document means to have nodes with the same name in the same path.
attribute verified operator \((a[@id = 7])\rightarrow pos_a\) verifyAttribute \(pos, d_a\) returns the set of positions where node \(a\) had an attribute \(id\) and it was verified.

To efficiently implement this approach, we represented the value of each predicate as a 64-bit number. This way, each predicate’s value will have the \(nth\) bit set to 1 in the case where the node corresponding to that predicate appears at that \(nth\) position in the current path. For example, for the XML path \(p = \{a \rightarrow b \rightarrow c\}\), the respective predicates would have the following values: \(root_a = 0...01\), \(pos_b = 0...10\), \(pos_c = 0...100\).

Since predicates now, represent only positions, we handle the wildcard location step ("**") by using a special position predicate. For the case where the wildcard location step addresses the root in a subscription\(^2\), we have the special predicate \(p_0\) that has its first bit set to 1 for any event. In the case where the wildcard appears in any other location step, we have \(p_1\), that has the \(n\) lower bits set to 1, where \(n\) is the depth of the path to process. This way, when applying the operators, we can handle the wildcard("**") location step as if it is a regular location step.

Note that, the generated predicates are independent of the operators. The predicates are stored together with the operators in order to combine each predicate with the respective operator. This is done when they are stored as part of a subscription. We shall detail this further in this work when we explain subscription indexing.

### 3.2.1 Indexing Predicates

Just like Le Subscribe, WFilter also generates a unique identifier, called the predicate id, for each distinct predicate. Recall that each predicate address either an XML node or an attribute (associated with a node). So, the predicates are indexed in order to provide fast mapping between each predicate and its respective predicate id. The index for the predicates concerning node names consists in a hash table, and it’s called predicate index. This predicate index takes the node name as input and returns the correspondent predicate id.

Besides the operator, structural location steps\(^3\) consist simply of a node name and hence can be indexed in a hash table as previously mentioned. However, when dealing with attribute expressions, a more complex indexing scheme is required as attribute expressions are also more complex.

\[
S = \{/a [ @id = 3 ]\}
\]

![Variables in an attribute expression.](image)

To cope with this additional complexity in attribute expressions we opted to use some intermediary operations where the variables were gradually combined into a single key through

\(^2\)It’s the first location step of the subscription in the form \(s = \{/ \ast /\ldots\}\).

\(^3\)Subscription location steps that address the structural part of a document exclusively, namely "/node","//node","\ast\ast"."
the use of auxiliary indexes. Recall that a condition regarding an attribute is represented by a
combination of: node name, attribute name, operator and value as shown on Figure 3.2. We ba-
sically used one hash index to convert the pair (nodename, attributename) into an id called
attribute id. Then, because the operator field has only 5 possible values, we index the set
(attributeid, operator, value) in a group of five 2-dimensional indexes using the operator value.

Each of those indexes will contain a mapping that receives the attribute id as a key and
returns a set pairs consisting of (value, predicateid). This is basically, the set of values that
every subscription has expressed with that operator to that attribute id. The predicate id is
the predicate in the system that addresses that combination of (attributeid, operator, value). A
more illustrative example of how attribute predicates are indexed can be seen on Figure 3.3.

Figure 3.3: Process of converting an attribute expression to a predicate that can be indexed in
WFilter.

Looking at Figure 3.3 we can see the structure of any of the five 2-dimensional indexes. This
index is very similar ways to the predicate families shown in LeSubscribe shown in Section 2.7.

3.2.2 Indexing Predicate Evaluation Results

Predicate evaluation results are indexed based on the id of that predicate in a vector called
predicate vector. The $n^{th}$ position of the predicate vector will store the evaluation the predicate
of id $p_n$, taking into account the processed event. Each entry in the predicate vector can be seen
a bit-chain. Recall that, the purpose of 64-bit number is to represent the depth in which each
of the predicates was found on the current event. Therefore, the way these entries are filled
depending on the kind of the predicate corresponding to that entry. A sample of the predicate
vector can be seen in Figure 3.4.
3.2.3 Indexing Subscriptions

Each subscription is also assigned a unique id and is then stored into a cluster. Each entry in the cluster will contain the subscription id and all the predicate ids of the predicates composing that subscription. These clusters group subscriptions based on their size and on one\(^4\) predicate common to all the subscriptions in that cluster. This means that, a cluster is characterized by the size of the subscriptions it stores and the chosen common predicate between those subscriptions, called the access predicate. The access predicate combined with the subscription size, indicates in which cluster the subscription should be stored. A sample cluster in a system with 3 subscriptions is shown in Figure 3.5.

\(^4\)This is because WFilter is based on LeSubscribe unidimensional solution, LeSubscribe multidimensional solutions use several predicates for indexing.
3.3. MATCHING ALGORITHM

Note that, as previously mentioned the operators between the predicates are stored in the clusters together with a subscription’s predicates. This is done through the use of flags. The default case is when the operator preceding a location step is a parent("/") operator and as such this is the assumed operator if there is no flag set. In the case the operator is an ancestor("///") operator, the predicate will be stored in the cluster with a negative id. When dealing with the attribute-verified operator, the predicate’s value in its respective predicate vector entry will have the most significant bit set to 1 (1....00 as default value) to represent this operator.

3.2.4 Indexing Clusters

As explained before, after a subscription is converted into a set of predicates, it is stored in a cluster. To be able to quickly determine which clusters should be verified for a given event, clusters are also grouped in a cluster family. Clusters are grouped in the same family if they share the same access predicate\(^5\). Those cluster families are then indexed in a vector, called cluster vector. They are indexed by the access predicate of the clusters contained in them. This means that the \(n^{th}\) position of the cluster vector contains a list with all clusters that store subscriptions with the predicate of id \(n\).

3.3 Matching Algorithm

3.3.1 Document Parsing

The first step is to parse each incoming XML document and convert it into a set of internal events as described in Section 3.1. This is done using a SAX parser that will fire an event for each start node, containing the node name, the attributes and the position in the document at which that node appeared and an end node event containing the node name. With this event-based parsing, we know that we’ve reached the end of a path (internal event) when we have an end node event followed by start node event.

Each internal event consists in a nodename, attributes, position 3-tuple for each node in that path. Then two processing phases are performed for each internal event: Predicate Evaluation and Subscription verification.

3.3.2 Predicate Evaluation

Each nodename, attributes, position 3-tuple contained in the internal event will be converted to a set of predicate ids\(^6\) through the predicate indexes detailed in Section 3.2.1. Then, using that predicate id, we set the bit position (position is the depth of the node) to 1 in the result entry predicate id of the predicate vector, as shown in Section 3.2.2. This denotes that a match has occurred.

\(^5\)Note that there are several clusters with the same access predicate but different size.

\(^6\)One for each node name and one for each attribute.
3.3.3 Subscription Verification

Recall that, after the Predicate Evaluation phase, all the predicates verified by the current internal event have their bit-chain entries in the predicate vector set to 1 at the depth where they appeared. Hence, the next step is to verify which subscriptions are matched. To do this, we start by selecting the clusters that may contain subscriptions that will be matched, the candidate clusters. This cluster selection consists in selecting, using the cluster vector, clusters that store subscriptions with size equal or lesser than the depth of the current path and have an access predicate that has a value different from zero. The clusters whose access predicate is zero do not need to be considered since they concern subscriptions that refer at least one element (represented by the access predicate) that is not present in the path being processed. An illustrative example can be seen on Figure 3.6.

![Diagram of Path Being Processed and Predicate Vector]

### Cluster of size 3

- **p1**
- **p4**
- **p1**
- **p0**
- **p2**
- **p3**
- **p0**
- **p3**
- **p3**

### Subscriptions

- **S1** = {a/b/c}
- **S2** = {x/*/c}
- **S3** = {a//b/c}

**Figure 3.6:** A sample of a possible execution using the previous examples.

Then, we must evaluate each of the subscriptions contained in the previously selected candidate clusters. All the subscriptions in each candidate cluster must be checked, either until they are matched or until one of their predicates does not match the respective operator, at which point we can skip the remaining predicates of that subscriptions and proceed to the next subscription in the cluster.

The specific algorithm to check each subscription of each cluster takes care of determining which operators it should apply between the predicates of a subscription based on the previously set flags. We implemented this operators as the following set of bitwise operations.

\[ \text{pos}_b \text{ parent } \text{pos}_c \left(../b/c\right) \rightarrow \text{This operation must return the positions where predicate } \text{pos}_b \text{ was verified right before predicate } \text{pos}_c. \]  

\[ \text{So, we implemented it as } \text{pos}_b < < 1 \& \text{pos}_c. \]

\[ ^7 \text{Note that, having at least one 1 in the predicate result respective bit-chain means the access predicate was satisfied by at least one node in the path and as such the subscriptions in that cluster must be thoroughly verified.} \]
3.3. MATCHING ALGORITHM

\( pos_b \) ancestor \( pos_c \) (.//b//c) \( \rightarrow \) This operation must return the positions where predicate \( pos_c \) was verified at any position after the predicate \( pos_b \). So, we implemented it as \( pos_c \& BinaryMask(pos_b) \), where binary mask returns a 64-bit number that has the most significant bits set to 1 until the first set position in \( pos_b \) and the rest of bits are set to 0. This mask basically removes all the occurrences of node c before node b and leaves unchanged all the occurrences of node c after node b.

\( pos_a \) verifyAttribute \( pos_i \) \( da(a[@id = 7]) \) \( \rightarrow \) In this case we just need to verify if the attribute was matched at the same depth as the node, so it’s implemented as \( pos_a \& pos_i \).

Algorithm1: checkCluster() The following sample code shows how each cluster’s subscriptions are verified:

```plaintext
01: for(j=0; j < clustersize; j++) { //Iterate through subscriptions
02: res = predicatevector[cluster[0][j]]; //Store the result of the first predicate
03: for (i=1; res != 0 && i < subsize; i++) { //Iterate through the remaining predicates
04: predid = cluster[i][j]; //Get the predicate id of the predicate currently being evaluated
05: if (predid > 0) {
06: next = predicatevector[predid];
07: if (next > 0)
08: res = (res << 1) & next; //regular case
09: else {
10: res = res & ~next; //attribute case
11: }
12: else { //ancestor-descendant case
13: next = predicatevector[-predid];
14: res = next & getBinaryMask(res);
15: }
16: } // end of second for
17: } // end of first for
```

3.3.4 Optimizing Document Parsing and Predicate Evaluation

By following the aforementioned approach, we would be doing the redundant work of first parsing each node into an internal event and then convert each node from the internal event into a predicate. Not only that, but each internal event would share some nodes and we would be computing the predicates satisfied by those shared nodes and respective attributes several times. In order to compute the predicates of each XML node only once we’ve modified the parser.

The parser was modified to combine the Predicate Evaluation phase with the parsing of a document. Basically, as we parse a document we already know to which internal event each node belongs. All parsed nodes belong to an internal event until an end node event is fired followed by a start node. Recall that, this represents the end of the current path. Therefore, every time a start node event is fired we can immediately convert that node and attributes into the respective predicate ids and add the respective matches to the predicate vector. Then, every time we find the end of a path, we immediately begin the Subscription Verification phase.

To prevent having to compute the predicates satisfied by the nodes shared between internal events, whenever we handle an end node event, we remove the corresponding match from the respective predicate vector entry immediately. This way, the matches on the predicate ids of the shared nodes are only removed when all the paths that share that predicate have been processed.
3.3.5 Optimizing Access Predicate Indexing

Even tough this is an optimization already present in LeSubscribe, it has a very significant impact on the performance and as such it was relevant to mention. Basically we use an additional vector, the access predicate vector, that stores all the predicates that were matched in the current internal event. Therefore, to determine which clusters should be verified, there is no need to traverse all the entries on the predicate vector and check if they are different from 0. Instead, we just need to traverse the access predicate vector and get the clusters from the cluster vector for each of the predicates in the access predicate vector, as long as those clusters size is equal or lower than the current event depth.

3.3.6 Optimizing Events through Compression

As previously shown, WFilter breaks down each event (XML document) into several internal events, each composing a unique path from the root to a leaf node. Even though this handling allows us to greatly simplify the incoming events, it does so at the cost of performing redundant computations when processing each internal event. Despite the increased cost of having to perform redundant computations, it still is a worthy trade-off. However, finding a way to skip or reduce the redundant computation was clearly an effective optimization.

To achieve a reduced set of events to process, we found a way to compress certain events. In this case, events that have all the nodes in common except the leaf node. This compression can be done only if those leaf nodes do not have a common name, otherwise, there would be a risk of matching false positives when dealing with attribute predicates separated by an AND. Assuming that one leaf node satisfied one part of the AND expression and the other satisfies the other, it would evaluate that expression to true, while that is not correct, since an AND operator indicates that a single node satisfies both the conditions specified by the AND operator.

This optimization can reduce the amount of clusters that must be verified. Since the leaf nodes of each branch are processed simultaneously, we spare the redundant computations of having to process each of the nodes in the branch once for each of its leaf nodes.
Figure 3.7: An example of the result of compressing events
Evaluation
4 Performance Evaluation

The greater danger for most of us lies not in setting our aim too high and falling short; but in setting our aim too low, and achieving our mark. – Michelangelo

4.1 Summary

We performed a set of experiences in order to evaluate the performance of WFilter. We wanted not only to analyze how our algorithm performs in nearly all possible scenarios, but also, how it compares to the current state-of-the-art. Therefore, we’ve chosen to compare our performance with YFilter. We’ve chosen YFilter because, to date, when dealing with subscriptions containing all kinds of XPath operators, YFilter is still arguably the most efficient algorithm. Despite other proposed approaches being able to perform better on certain scenarios, YFilter has an equivalent performance overall, with the differences being just a matter of tradeoffs. Besides, the way these algorithms are implemented can have a very high impact in their performance. Thus, this renders the approach of implementing them for comparison an unfair option. However, unlike other algorithms, YFilter authors made the source code, executables and testing environment available for download.

Our experiments were performed using the YFilter testing package, including the XPath query generator, the documents of the nitf dataset and the XPath query parser to parse the subscriptions into our systems. By using the same parsers we ought to guarantee that the comparison is as fair as possible.

4.1.1 Configurations and Environment

YFilter has two strategies to perform attribute-expression matching. We configured the YFilter executable to always use the SP (Selection-Postponed) approach for evaluating predicates, as it was already shown to be their fastest approach.

For WFilter, we wanted to choose the most selective possible predicate to work as access predicate for the subscriptions, so we set the following rules for access predicate:

1st choice Predicates that represent attribute expressions as they are more unique.

2nd choice In case a subscription doesn’t contain any attribute expressions, its access predicate will be the last node that it’s not a wildcard.

3rd choice When a subscription is only composed of wildcards, their access predicate is the special wildcard predicate.

We conducted all our experiments in a 2.3 GHz Intel Core i5 processor with a L2 cache of 256 KB and a L3 cache of 3 MB and 4 GB of 1333 MHz DDR3 ram memory.
4.2 Loading Time

First, since these algorithms are designed to be employed in a dynamic environment, we wanted to measure how fast each of them can load a new subscription to its index. To do this, we’ve generated a high amount of subscriptions and measured the time each algorithm took to load all those subscriptions into its indexes. These experiences are shown in Figures 4.1 and 4.2.

![Diagram showing loading time for increasing amount of subscriptions with no attribute expressions]

Figure 4.1: Loading time for increasing amount of subscriptions with no attribute expressions

In the case where there are no attributes, the difference between the two algorithms is noticeable. WFilter performs roughly between two to three times faster. WFilter’s conversion of location steps to predicates is done through hash indexes which perform in nearly negligible time. Instead, YFilter has to traverse their current NFA for each of the subscription’s location steps and for new location steps, generate their respective state.

Besides determining the predicates for each subscriptions, WFilter also needs to determine the access predicate of a subscription, which usually can be done implicitly as the subscription is parsed. The process of determining in which cluster should a subscription be stored, based on its access predicate, is equivalent to a perfect hashing function.

The consequences of the aforementioned processes can be seen in the presented graphs. For WFilter, a constant increase in the amount of subscriptions causes a roughly constant increase in time, since most of the times WFilter performs the same operations. In the case of YFilter, as the amount of indexed subscriptions increases, the probability of finding commonality between a new subscription and already indexed subscriptions also increases causing YFilter to scale increasingly better as the amount of indexed subscriptions increase.

When dealing with attributes, YFilter has to create a more complex NFA node structure
4.3 MATCHING TIME

The second kind of measurement we took and the most relevant was the measurement of the matching time. For these experiments we used 100 XML documents (events), so the matching time is the time that goes from the moment when the first document starts being parsed to the moment where the algorithm has determined all the matched subscriptions for the 100 documents. The document size ranged from the 4 KB to 12 KB with the average being roughly 8 KB.
4.3.1 Varying Amount of Subscriptions

Our first matching experiment intended to compare how both algorithms in an environment as close to a real case as possible. To do this, we attempted to generate a set of subscriptions with a distribution of operators that could be balanced enough to closely emulate a possible real case. We ran a comparison test for a set of subscriptions without attribute expressions and another with attribute expressions.

Wildcards probability: 10%
Max depth: 10
Ancestor-descendant probability: 20%
Attribute probability: 0%

Figure 4.3: Matching time for increasing amount of subscriptions with no attribute expressions

When dealing with only structural subscriptions which is the main concern of YFilter, our algorithm shown an improvement in the order of 60% to 100%. Both the algorithms scale in a constant manner and as such, increasing the amount of subscriptions without compromising their diversity, will have no impact on their scaling, as shown in Figure 4.3.

Just by adding a small amount of attributes to the subscriptions, the performance gap becomes much larger. YFilter has to perform an additional complex phase to process the attributes, while WFilter processes them almost in the same way as structural expressions. Moreover, WFilter also takes advantage of a more efficient clustering. In cases where subscriptions contain at least one attribute expression, the respective predicate will be used as access predicate. Resulting in an increase in the number of clusters, reducing the average number of subscriptions per cluster and increasing the average selectivity of the access predicates. These results are shown in Figure 4.4.

4.3.2 Impact of Wildcards

The wildcard (*), as an additional operator that can be matched by any node in a document can have a potential slow down in the efficiency of the matching process. It corresponds to
4.3. MATCHING TIME

![Diagram showing matching time for increasing amount of subscriptions with attribute expressions]

Figure 4.4: Matching time for increasing amount of subscriptions with attribute expressions

An evaluation that will be true for many of cases and can highly increase the solution space when compared to a regular location step. Usually, the increase of wildcards in subscriptions is directly related to a noticeable increase in the time spent computing the matches.

As shown in Figure 4.5, both algorithms behave similarly when the percentage of wildcards in subscriptions increase. Up to a certain percentage there is a tendency to increase the matching time. Since the wildcard is verified by any node, it will almost always lead to the need to proceed evaluating. However, YFilter explores commonality between subscriptions, and WFilter indexes subscriptions based on their predicate that is not a wildcard. Therefore, a very high percentage of wildcards will lead to a reduction in the processing time. This is due to YFilter being able to explore a lot of commonality between a set of subscriptions that consists mostly of wildcards. Also, for WFilter, increasing the wildcard percentage past a certain threshold will amplify the selectivity of WFilter access predicates. This is because increasing the percentage of wildcards means a reduction in the amount of regular location steps which are used as access predicates.

Looking at the results, WFilter, performs around 50% slower in the presence of wildcards compared to a situation where there are either no wildcards or too many. For YFilter, as the percentage of wildcards changes from 0% to 10% it immediately causes a 300% slowdown. That slow down reaches up to nearly 500% around 50% wildcards.

4.3.3 Impact of Ancestor-Descendant

Ancestor-Descendant operator ("//") addresses nodes at variable depth which forces an algorithm in some way to keep state and, either wait until that operator is satisfied or until the event is entirely processed. This always causes a performance downgrade, so any matching
CHAPTER 4. PERFORMANCE EVALUATION

Number of Subscriptions: 500000
Max depth: 10
Ancestor-descendant probability: 20%
Attribute probability: 0%

solution must be able to scale well when dealing with ancestor-descendant operators.

In this case of Figure 4.6, where there are no attributes, despite YFilter being slower, both algorithms scale similarly well. There is a noteworthy difference from no ancestor-descendant at all to any percentage, but both algorithms have a stable small decrease in performance of around 10% slower for each extra 10% ancestor-descendant operator distribution.

As shown already, the ancestor-descendant is an operator that has a significant influence in the overall performance of the matching process. However, it is expected to be less significant when in a more expressive environment, namely one including attributes, as there are more operators to be handled. Therefore, it is expected that an efficient matching algorithm can either sustain its scaling factor or even reduce the impact of the ancestor-descendant operator. The results of such experiment are presented in Figure 4.7.

In the case where subscriptions can contain attributes, both algorithms are able to scale much better. This is because the processing time for the ancestor-descendant("//") is roughly constant in both approaches. Due to this, the additional time taken to process the attributes overshadows the differences in performance when dealing with the ancestor-descendant operator. So, despite YFilter being much slower in the presence of attributes, both algorithms seem to be nearly unaffected by the presence of the ancestor-descendant operator if the subscription set contains some subscriptions with attributes.

4.3.4 Impact of Depth

We’ve also decided to analyze the importance of the depth of the subscriptions in the total time taken to determine the matched subscriptions. To test this case, we’ve variated the depth between 2 and 13.

The graph of Figure 4.8, shows that both algorithms are practically unaffected by the increase in depth of the subscriptions when dealing with a diverse environment (subscriptions
4.3. MATCHING TIME

**Number of Subscriptions:** 500000  
**Max depth:** 10  
**Wildcards probability:** 10%  
**Attribute probability:** 0%

![Graph](image)

Figure 4.6: Impact of ancestor-descendant on both WFilter and YFilter using subscriptions without attribute expressions containing attributes. The spike in the lower depths happens because by generating so many different subscriptions with so few location steps, the probability of those subscriptions sharing very few commonalities is very high. Thus, rendering the commonality exploitation of both WFilter and YFilter not very effective.

### 4.3.5 Impact of Attributes

As previously shown, attribute expressions have a huge impact on how fast the algorithms perform. Therefore, we decided to analyze the direct impact of increasing the probability of subscription’s location steps containing attribute expressions. We’ve varied the probability of attribute expressions from 0 to 60%. The results of this experience are shown on Figure 4.9.

The initial spike from 0% to 10% attribute expressions is expected for both algorithms. It happens, because in the first case, attribute expressions don’t need to be handled at all. While in the 10% case, the additional techniques to handle attribute expressions must be used. As the amount of attribute expressions increases, the two algorithms have drastically different performances. In the case of YFilter, the more attribute expressions there are, the more work that must be done, hence the increase in the processing time. For WFilter, the increase of attribute expressions actually improves the performance. This is due to attribute expressions being used as access predicates. Attribute expressions provide a very high selectivity and so, the more attribute expressions, the more the average selectivity of each access predicate.
CHAPTER 4. PERFORMANCE EVALUATION

Number of Subscriptions: 500000
Max depth: 10
Wildcards probability: 10%
Attribute probability: 10%

Figure 4.7: Impact of ancestor-descendant on both WFilter and YFilter with subscriptions containing attribute expressions

Number of Subscriptions: 500000
Wildcards probability: 20%
Ancestor-Descendant depth: 25%
Attribute probability: 15%

Figure 4.8: Impact of varying the depth on the subscriptions
Number of Subscriptions: 500000
Wildcard probability: 15%
Max Depth: 10
Ancestor-Descendant depth: 15%

Figure 4.9: Impact of increasing the amount attribute expressions on the subscriptions
5.1 Conclusions

We are in the Information age: the amount of information available to a user is huge and it increases each day. In this context, being able to disseminate the right information to the interested users is crucial from the point of view of users. This way, they do not have to browse through information to find the one that they are interest in. Publish/Subscribe systems are an approach for the dissemination of information. In these systems, entities that publish information are called publishers and users interested in that information are called subscribers. When a publisher publishes information in the system it’s called an event.

Despite being suited for the same problem it is impossible to fairly compare the algorithms shown in Chapter 2 since they do not support the same range of XPath expressions or employ completely different paradigms that bring a lot of trade-of’s and outperform each other depending on the scenario on which they are employed. However, most of them focus on structure matching, supporting any type of structural XPath expressions or just a subset, not handling expressions concerning attributes at all.

Only a few support XPath expressions that can contain both structural expressions and expressions that concern the attributes of a XML node. Taking into account that no algorithm supports the full syntax of XPath and some of them, either don’t support predicates or are better suited for specific cases, it becomes impossible to select one as the best solution for a general case. So, we will consider the ones that apply efficient techniques to a larger subset of the XPath language while still handling a high volume of input documents per second, a very high number of subscriptions and can handle adding and removing subscriptions during runtime, the most suitable solutions for a publish/subscribe system. As of currently, the most complete algorithms that attempt to provide a solution to the same problem we propose to, are YFilter, pFiST and Predicate-based filter as they include at least one indexing technique for subscriptions, support structural matching and also include some level of value-based predicate filtering.

In this context, when dealing with a high amount of users/subscriptions and a high rate of incoming events, an important problem that must be solved is the matching problem: How to determine the subscriptions that are matched by each incoming event. Besides, as the subscription languages increase in expressiveness, allowing the users greater ability to express their interests, they also increase the complexity of the respective matching process.

Considering that XML is currently considered almost as a standard to represent information, it makes sense to use it as the format to represent the information submitted to the publish/subscribe systems. Moreover, and considering that now the information/events are represented as XML documents, then the subscription language should be a query language suited for XML so that users can specify their interests with precision. Thus, XPath as the most com-
mon language to query XML Document paths, it also makes sense that in this context it is used as a subscription language.

Predicate-Filter implements a very similar solution to ours also based on Le Subscribe. It also handles both structural and attribute expressions. However, verifying the predicates is not a simple process and its performance is equivalent to that of YFilter.

In this thesis, I presented an efficient solution for the matching problem. Being able to handle subscriptions consisting of a vast subset of the XPath language, including structural location steps and attribute expressions.

Not only is WFilter able to handle subscriptions encompassing a large subset of the XPath language, but it is able to do so while efficiently supporting a high rate of incoming events and large number of subscriptions. WFilter achieves this performance by converting all of the subscription’s location steps into predicates and fully exploring commonality between them: each unique predicate is evaluated only once. Subscriptions are indexed in clusters allowing us to reduce the possible solution space by a noteworthy amount. Finally, we can quickly perform the mapping between predicates, subscriptions and clusters through a set of indexes.

The experiences we’ve done show that our solution can, not only scale well in the presence of any operator, but also, severely outperform YFilter when dealing with both structural and attribute expressions combined in a diverse subscription environment.

5.2 Future Work

This work could be further improved by increasing the expressiveness of the subscription language by allowing nested path expressions. Currently, it would be simple to split each nested path expression into two different subscriptions, one for each path, and store them as separate subscriptions. Then, by using a map, we could map each subscription containing nested paths to the subscriptions resulting of the splitting. Through this map, all that would be needed would be a post-processing step to analyze if all the subscriptions resulting of the splitting were matched. However, not only does this require a post-processing step, it also goes against our strategy of matching all kinds of operators in the same process by using a single approach.

Other possible way of achieving an improvement is to implement find a way to increase processing speed even further. Taking into account the behavior and the data structures of our matching algorithm, this algorithm can be easily parallelized. This would allow us to improve the performance of WFilter since by parallelizing it, we could take advantage of the currently very common multi-core processors. Since our measurement showed that the cluster subscription verification phase takes much more time than the XML parsing and predicate evaluation phase, the best approach to parallelize this algorithm would be to parallelize the cluster subscription verification phase. For instance, we could have several threads (based on the number of cores) that would be responsible for processing the whole set of candidate subscription clusters for a given event. In this approach, each thread would process a disjoint subset of the candidate subscription clusters in a distinct core.

Besides increasing processing speed, language processing or both by introducing additional techniques, one could also focus on improving the current ones. WFilter as shown in experiments is highly dependent on the selectivity of the chosen access predicates. For our experiments, we tried to chose access predicates that would provide an efficient performance
for the most cases, even on the worst-case scenario. However, the clustering mechanism and access predicate criteria could be further studied in order to attain improvements that could be significant.


