REtriever:
String Maching Engine For Publish/Subscribe Systems

Daisy Zhe Wang and Hans-Arno Jacobsen
University of Toronto

October 25, 2004

Abstract

In this paper, we will first present **REtriever**, a DFA-based engine for solving the string matching problem in publish/subscribe systems. For the current stage, REtriever supports a subset of regular expression, which can be used to specify XPath language, subscriptive continuous stream queries and string-based rules. We then will discuss experimental evaluation of the performance of REtriever as compare to the nfa-based and the naive string matching engines.

1 Introduction

The string matching problem is important for a number of application domains, including XML filtering and classification, continuous query over streams, and content-based routing. REtriever aims at solving the string matching problem in the publish/subscribe (pub/sub) systems context, where applications have to manage large numbers of string-based subscriptions. REtriever can be adpated to be used for XML filtering and continuous queries on streams.

The REtriever uses a deterministic finite automata(DFA) based structure and further develops the idea of shared-path-prefix processing in YFilter. It uses keyword matching and recombination techniques to realize the regular expression (RE) matching in a DFA-based structure, and it also developed algorithms to increase the shared processing of overlaps in subscriptions. Our experiments show that REtriever achieves 50 percent of performance enhancement relative to the nfa-based matching engine, which uses an NFA-based structure and only share the processing of prefix overlaps in subscriptions.

The remainder of this paper is organized as follows. Section 2 presents the language models for subscriptions and publications in pub/sub systems, the data structures of REtriever and the two design issues. Then we describe various REtriever algorithms in detail in Section 3. In Section 4, we analyze the experimental results of the performance of different algorithms in REtriever relative to nfa-based and naive approaches.

2 Overview of REtriever

REtriever is designed for string matching in pub/sub systems. Different from any previous work, REtriever uses DFA-based keyword matching and recombination techniques to solve our subscription matching problem, which is given an arbitrary incomming publication p, a set of subscriptions, $S = \{s|s \text{ matches } p\}$ can be retrieved quickly and efficiently. In this section, we first present the language model of an entirely string-based pub/sub system. Then, we present the data structure underlying REtriever. Based on the design principles behind any DFA-based structure, REtriever develops its own novel techniques to further reduce the size of the DFA by *splitting algorithms* to exploit *general overlaps*, which can be

any sub-string in a subscription. We will highlight these new techniques in Section 2.3, deferring the details of our solutions to Section 3.

2.1 Language Models for Pub/Sub System

A string-based subscription language model consists of an alphabet Σ and a set of operators Γ. Σ can be a random set of literals. In the context of this paper, we choose Σ to be the English alphabet. In addition, we have two wildcard operaters: $\Gamma = \{?, *\}$, each of which has its own instantiation. "?" can be instantiated to any one literal over Σ; "*" can be instantiated to any sequance of literals over Σ. A *subscription* is a sequence of literals and operators, and its instantiation is the concatenation of literals and the instantiations of each operators in sequence. For example, "a?b" can be instantiated to $\{aAb|\ A \in \Sigma\}$; "a*b" can be instantiated to {aSb|S is any string over Σ}. The number of instantiations of a subscription can be finite, as in the first example, as well as infinite as in the second example.

The publication language model simply consists of the same alphabet Σ as in the subscription language model. A *publication* is a string of literals over Σ. A publication with finite length is a *string publication*, whereas publication with infinite length is a *stream publication*.

A publication *matches* a subscription when part of the publication is an instantiation of the subscription. For example, publication "aabcbaaabc" matches subscriptions "aaa", "a?b", "b*b", but does not match subscription "bb".

2.2 Data Structure of REtriever

For each subscription S, we tokenize it into **phrases** using operators Γ as deliminators, and insert each phrase into a DFA structure. The transition functions of the DFA structure are literals over Σ. For example, in DFA_1 in Figure 1, subscription "song?blue" is tokenized into phrases "song" and "blue", which are inserted into the DFA saparately. Each node in the DFA has a *state* data structure which consists of the components shown in Table 1. (The concept of *accept* and *half accept* will be introduced in Section 3.2)

Data Structure of a *state*	
component	content
id	unique identifier of the state in DFA
key	transition key to **s**
parent	parent state
shareDegree	number of times **s** is shared by different subscriptions
transitions	a hashtable of (key, nextState) transition pairs
acceptSet	a hashtable of *acceptRecord*s, representing subscriptions that is *accepted* in this state
halfAcceptSet	a hashtable of *halfAcceptRecord*s, representing subscriptions that is *half accepted* in this state

Table 1: Data Structrue of *state*

The *initState* is a special state that represents the root of the DFA, which corresponds to $state_0$ of DFA_1 in Figure 1. All other states can have one or more of the three following runtime state – *shared*, *accepted*, *half accepted*. For example, in DFA_1 of Figure 1, $state_4$ is a halfAcceptState of $subscription_1$ "song?blue", and $state_8$ is the acceptState of the same subscription. Also, $state_5$ is shared by subscription "song?blue" and "berry".

Features of state **s** in DFA of REtriever	
feature	meaning
shared	**s** is shared by two or more subscriptions
accepted	**s** accept one or more subscriptions
half accepted	**s** half accept one or more subscriptions

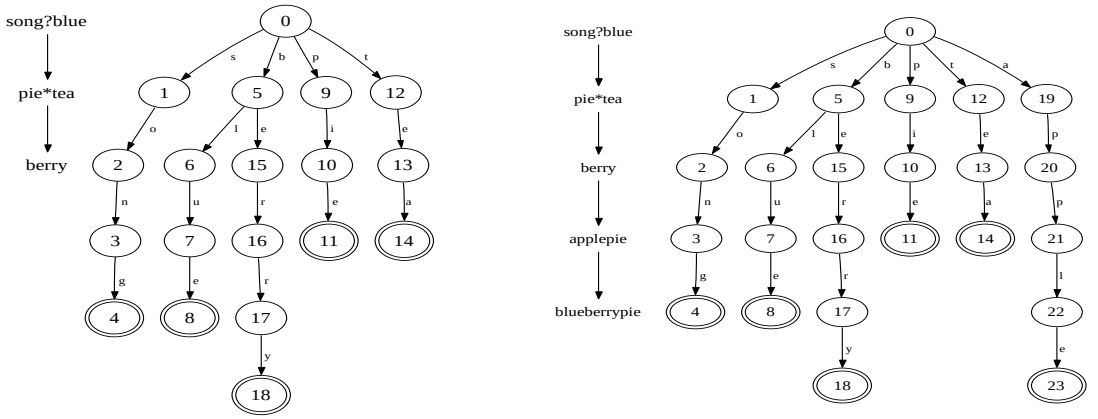


Figure1: REtriever DFAs before and after inserting “applepie” and “blueberrypie”

Table 2: State features in DFA of REtriever

In addition, a subscription hashtable stores all subscriptions as tuples of (id, acceptState, acceptRecord, subscription), and a result hashtable stores all the resulting subscriptions in publication matching.

2.3 Design Issues

At a higher level, the main goal in designing REtriever is to minimize the time to find all subscriptions to which a given publication match. The first design issue is whether to use a DFA-based or nondeterministic finite automata (NFA) based structure. While NFA-based structure takes more time to execute, DFA-based structure theoretically might has exponential number of states. The second design issue is how to make use of the overlaps in subscriptions to share as much processing as possible.

To address the first design issue, we first adapted YFilter to a nfa-based algorithm, which is able to process our subscription language. The nfa-based algorithm, like YFilter, shares the processing of the prefix overlaps in subsriptions by using a combined-NFA structure. Because of multiple active states at one time, the matching time for the nfa-based algorithm is slower than using corresponding DFA. However, if we translate the NFA to DFA using the standard canonical method, we might get exponential number of states. We found that (1) wildcard operators Γ are expensive to evaluate because they introduce multiple active states; (2) NFA built from a keyword, a substring of subscription without wildcard operator Γ, is essentially a DFA. Thus, as introduced in the data structure, our approach tokenizes each subscriptions to phrases using Γ as deliminators. We use DFA-based keyword matching for each phrase matching, and recombine the results to get the matching result of the subscription. The resulting DFA has less number of states than the corresponding NFA. We will present the DFA-based keyword matching and recombination techniques along with the three main operations: insert, match and delete of REtriever in Sections 3.2 and 3.3.

Above described is the no-splitting algorithm in REtriever. In order to reduce the size of the DFA, we want to exploit not only the prefix but all general overlaps in those phrases. We designed two splitting algorithms called: *singleSplit* and *multipleSplit*, which can split each phrases into **words**, whose prefixes are overlaps in the subscriptions. Each words are inserted separately into the DFA, and because they

have common prefix, the size of the DFA is reduced. REtriever is referred to as no-splitting, single-splitting and multi-splitting algorithms, depends on whether we use the splitting algorithm and which splitting algorithms we use. We will discuss the two splitting algorithms as how they split phrases to discover general overlaps in detail in Section 3.1.

3 Algorithms

In this section we will first present two splitting algorithms: *singleSplit* and *multipleSplit*, which exploit the general overlaps in subscriptions to further reduce the size of the DFA. In Section 3.2, new data structures and algorithms for keyword matching and recombination will be described. Lastly, we will present three key operations of REtriever: *insert, delete* and *match* in Section 3.3.

3.1 Splitting algorithms

The goal of our splitting algorithms is to find a way to split a *phrase* into *words*, so that when inserting each words saparately into the DFA, more states are shared. This will reduce the size of the DFA.

3.1.1 Algorithm getScore

Before introducing the two splitting algorithms, function *getScore* should be explained first. *getScore(sub, startPos, endPos)* calculates the *score*, which is the number of states that can be shared in the existing DFA if we insert part of the subscription from startPos to endPos into the DFA.

```
getScore (sub, startPos, endPos)
1   curState ← initState
2   score ← 0
3   for curPos ← startPos . . . (endPos − 1) do
4     if (sub[curPos], nextState) pair is present in the transitions table of curState then
5       score ← score + 1
6       curState ← nextState
7     endif
8   endfor
9   return score
```

For example, the score of "blueberrypie" as a whole with the existing DFA_1 in Figure 1 is 4; while the score of "blueberrypie" from the 5th literal to the end with DFA_1 is 5.

3.1.2 Algorithm singleSplit(phrase)

The *singleSplit* evaluates the score, s_0, of an input phrase with no splitting point, and evaluates the scores, s_i, of the phrase with splitting point at every possible position. The score of a phrase with a single splitting point is the sum of the scores of the first and second part of the phrase. If s_j = *MAX*(s_i) and s_j has a bigger score than s_0, then j is chosen as the splitting point. MIN_WORD_LEN is the minimum length of a word. MIN_SPLIT_SCORE is the minimum increment in score if we decide to split.

```
singleSplit (phrase)
1   if (lenth of phrase) >= (MIN_WORD_LEN * 2) then
2     maxScore ← getScore(sub, 0, ((length of phrase)))
3     splitPos ← 0
4     for i ← MIN_WORD_LEN . . . ((length of phrase) − MIN_WORD_LEN) do
5       score ← 0
6       curState ← initState
7       score ← getScore(phrase, 0, i) + getScore(phrase, i, (length of phrase))
8       if score > (maxScore + MIN_SPLIT_SCORE) then
9         maxScore ← score
10        splitPos ← startPos
```

For example, we want to insert "applepie" into DFA_1 in Figure 1. Because the scores of each possible splitting point in "applepie" are [5, 2, 3, 3, 4, 8, 5, 5, 5], we should split between "e" and "p".

3.1.3 Algorithm multipleSplit(phrase)

The *multipleSplit* is a dynamic programming algorithm, which split a phrase into multiple words. The score of phrase or prefix of phrase, phrase(i), with multiple splitting points is the sum of scores of words generated from splitting the phrase or phrase(i). *multipleSplit* finds out the set of splitting points which results in max score of the input phrase. The algorithm is structured and proved as follows.

1)Define array *splits(i)* and *scores(i)* for $0 < i <$(length of prase) by
phrase(i) = {prefix of phrase from 0 to i}
score(i) = {max score of phrase(i) with multiple splitting points}
splits(i) = {optimum last splitting point before i}
splits(length of phrase) holds the last splitting point of the phrase; if it is not 0, then splits(splits(length of phrase)) holds the second last splitting point of the phrase, etc.

2)Recurrence relationship
For $0 <= i <$MIN_WORD_LEN$*2$
splits(i) = 0; score(i) = getScore(phrase, 0, i);
For $MIN\_WORD\_LEN * 2 <= i <$(length of phrase)
scores(i) = MAX(scores(j)+getScore(phrase, j, i)); (MIN_WORD_LEN$<= j < i$)
splits(i) = j; ((scores(j)+getScore(phrase, j, i)) == scores(i))

3)Proof
For every step i, we try to decide the optimum position of the last splitting point before i. When $0 <= i < MIN\_WORD\_LEN * 2$, we do not split, thus splits(i)=0, and score(i) is score of phrase(i) with no splitting point. When MIN_WORD_LEN$*2 <= i <$(length of phrase), the optimum position j ($j < i$) is where the sum of scores(j), max score of phrase(j) with multiple splitting points, and score of part of phrase from j to i takes on the maximum value.

```
MULTIPLESPLIT (phrase)
1   if (length of phrase) >= (MIN_WORD_LEN * 2) then
2     for i ← 1 ... (MIN_WORD_LEN * 2) do
3       splits[i − 1] ← 0
4       scores[i − 1] ← GETSCORE(phrase, 0, i)
5     endfor
6     for i ← (MIN_WORD_LEN * 2 + 1) ... (length of phrase) do
7       maxScore ← GETSCORE(phrase, 0, i)
8       splitPos ← 0
9       for j ← MIN_WORD_LEN ... (i − MIN_WORD_LEN) do
10        score ← scores[j − 1] + GETSCORE(phrase, j, i)
11        if (score > (maxScore + MIN_SPLIT_SCORE)) or ((score > maxScore) and (splitPos! = 0)) then
12          maxScore ← score
13          splitPos ← j
14        endif
15      endfor
16      split[i] ← splitPos
17      score[i] ← maxScore
```

For example, we want to insert "blueberrypie" into DFA_1 in Figure 1. with MIN_WORD_LEN=2 and MIN_SPLIT_SCORE=2, splits array is [1, 2, 3, 4, 4, 6, 7, 8, 9, 10, 11, 12] and scores array is [0, 0, 0, 0, 0, 4, 4, 4, 4, 9, 9, 9]. Thus the multiple splitting positions are 4 and 9.

3.2 DFA-based Keyword Matching and Recombination

Before we get to the key operations of REtriever, we first present additional data structures that are used in DFA-based keyword matching and recombination.

By *tokenizing* a subscription using operators Γ as delimiters, we get a set of *phrases*. Then, we either do not split, or split each phrases into *words* using *single-splitting* or *multi-splitting* algorithms, and insert each word separately into the DFA. *acceptState* is the accept state for inserting the last word of the subscription; *halfAcceptState* is the accept state for inserting other words of the subscription.

acceptRecord is a tuple of (halfAcceptState, halfAcceptRecord, sub, numQuestion, hasStar) and *halfAcceptRecord* is a tuple of (preHalfAcceptState, preHalfAcceptRecord, sub, numQuestion, hasStar, numMatch, matchEnd, matchStart). *halfAcceptState* and *preHalfAcceptState* point to the accept state of the previous word. *halfAcceptRecord* and *preHalfAcceptRecord* point to the halfAcceptRecord of the subscription in the accept state of the previous word. *numQuestion* is the number of question whildcard and *hasStar* is a Boolean value of whether star wildcards exist between the previous word and this word. *numMatch* is the number of times the word is matched in the publication, and (*matchStart*, *matchEnd*) pairs record the start and end matching position of the word in the publication.

Data Structure of acceptRecord and halfAcceptRecord	
component	content
halfAcceptState/preHalfAcceptState	pointer to the accept state of the previous word
halfAcceptRecord/preHalfAcceptRecord	pointer to the halfAcceptRecord/preHalfAcceptRecord of the subscription in the accept state of the previous word
numQuestion	the number of question whildcard
hasStar	a boolean value of whether star wildcards exist between the previous word and this word
numMatch	the number of times the word is matched in the publication
(*matchStart*, *matchEnd*)	record the start and end matching positions of the word in the publication

Table 3: Data Structure of *acceptRecord* and *halfAcceptRecord*

In order to match, we have to *recombine* the matching result of all the words in a subscription. When a halfAcceptState of a subscription is reached, matching position pair has to be added to all halfAcceptRecords. When the acceptState of a subscription is reached, all previous halfAcceptStates of this subscription are traversed in reverse order, and matching position pairs are checked to see if one of them is the *right matching positions*. By *right matching positions* we mean: the matchEnd of the previous word and the matchStart of this word should be separated by exactly *numQuestion* number of literals if hasStar==0, or by a number of literals that is greater than *numQuestion* if hasStar==1.

Correspondingly, data structure of acceptRecord and halfAcceptRecord has to be established at insert time. Moreover, delete operation cannot be executed in a top down mechanism, but a bottom up mechanism, which will be explained in detail in the next section.

3.3 Algorithms for REtriever Operations

3.3.1 Algorithm insert(sub)

The *insert* operation is done in several steps:
1) tokenize a subscription to phrases
2) either no-splitting the phrases (*noSplit*) or split each phrases to words using *singleSplit* or *multipleSplit*
3) insert each word separtely in to existing DFA
4) increase shareDegree of existing state and set up acceptRecord or halfAcceptRecord for newState

5) set up acceptState in subscription hashtable

```
INSERT (sub)
1   tokenize sub using wildcard operators as delimiters, and generate a list of phrases
2   for each phrase p do
3     noSplit/singleSplit/multipleSplit p to a list of words
4     for each word w do
5       curState ← initState
6       i ← 0
7       while (word[i++], nextState) pair is present in the transitions table of curState do
8         increase shareDegree of curState
9       endwhile
10      for i ← i ... (length of word) do
11        add newState with word[i] as key
12        if (word[i] is the last literal in the last word of the last phrase in sub) then
13          add acceptRecord to newState.acceptSet
14          add (sub, curState) pair to hashtable of subscription
15        else
16          add halfAcceptRecord to newState.halfAcceptSet
```

For example, after inserting the splitted "applepie" and "blueberrypie" , we get DFA_2 in Figure 1.

3.3.2 Algorithm match(pub)

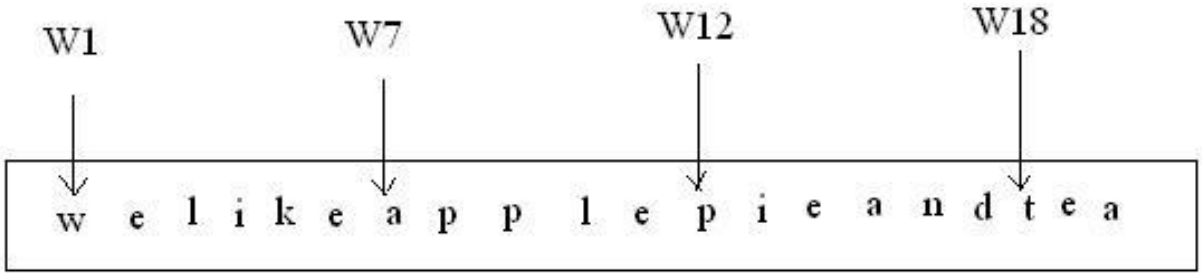
The *match* operation is done by sliding windows one literal to the right each time, for each window prefix match all subscriptions by executing DFA. For each state, the execution of DFA is done as follows.
1) DFA execution begins at the initial state.
2) look up the incomming character c in the transitions hashtable of current state.
3) if transition exist for c in current state

i) add match position pairs to all halfAcceptRecords in current state

ii) for all subscriptions which has acceptRecords in current state, recombine matching positions of all halfAcceptRecord and acceptRecord of the subscription. If *right matching positions* exists in all records, we add the subscription to the result hashtable.

iii) proceed to the next state

```
MATCH (pub)
1   for i ← 1 ... (length of pub) do
2     curState ← initState
3     for j ← i ... (length of pub) do
4       for every halfAcceptRecord hr in curState.halfAcceptSet do
5         increase numMatch and add new match position pairs ((i-1), (j-1)) to hr
6       endfor
7       for every acceptRecord ar in curState.acceptSet do
8         preHalfAcceptRecord ← acceptRecord.halfAcceptRecord
9         hasStar ← acceptRecord.hasStar
10        numQuestion ← acceptRecord.numQuestion
11        sub ← acceptRecord.sub
12        startPos ← (i − 1)
13        while preHalfAcceptRecord! = null do
14          for everymatchpositionpairs(matchStart, matchEnd) do
15            endPos ← 1 + matchEnd + numQuestion
16            if ((hasStar == 1)and(startPos >= endPos))or((hasStar == 0)and(starPos == endPos)) then
17              add sub to result list
18              startPos ← matchStart
19            endif
20          endfor
21          hasStar ← preHalfAcceptRecord.hasStar
22          numQuestion ← preHalfAcceptRecord.numQuestion
```



Graph 1: Illustration of window switching in *match* function

23 $preHalfAcceptRecord \leftarrow preHalfAcceptRecord.preHalfAcceptRecord$

For example, suppose the publication is "welikeapplepieandtea" and the DFA of all subscriptions is DFA_2 in Figure 1. In Window7(see Grpah 1), we matched "apple", and matched position pair is (7, 11); in Window12, we matched "pie", and the matched position pair is (12, 14); in Window18, we matched "tea", and the matched position pair is (18, 20). In addition, in Window12, we get to the acceptState of subscription "applepie", and find the match pair in its halfAcceptState state23 is (7, 11) which is a *right matching position*, and state23 is the only halfAcceptState for subscription "applepie", thus *"recombination"* is successful and "applepie" is matched. Another match is "pie*tea" while the matching positions, (12, 14) and (18, 20) is *right matching positions*, because hasStar==1 and 18>12. On the other hand, even though the acceptState state11 of "blueberrypie" is also reached, there is no *right matching position* in its halfAcceptState state18 and state4, thus it is not a match. So the matching subscription set for publication "welikeapplepieandtea" is "applepie", "pie*tea".

3.3.3 Algorithm delete(sub)

The *delete* operation is done with following steps: (note that all deletions are done in bottom up manner, i.e starting from the accept state of each word to the initState)
1) get acceptState and acceptRecord of the subscription
2) delete all acceptRecord and halfAcceptRecord of the subscription in the accept state for each *word*
3) decrease shareDegree for all states on the path to the accept state of each *word*. If shareDegree of one state drops to zero, delete the state.

```
DELETE (sub)
1   find (sub, acceptState, acceptRecord) pair in hashtable of subscriptions
2   curState ← acceptState
3   while curState! = null do
4      delete acceptRecord or halfAcceptRecord of sub in curState
5      while curState! = initState do
6         decrease shareDegree of curState
7         if shareDegree == 0 then
8            delete curState
9         endif
```

10 **endwhile**
11 $curState \leftarrow preHalfAcceptState$

For example, in order to delete subscription "applepie". First, we find its acceptState – $state1_1$, and decrease the shareDegree of state 11, 10 and 9. Then, we trace to its halfAcceptState, state23, and by decreasing the shareDegree of states 23, 22, 21, 20 and 19, we find that all shareDegree of those states drops to zero. Thus, we delete states 23, 22, 21, 20 and 19, and after deletion of "applepie", DFA_2 in Figure 1 transform back to DFA_1.

4 Experiments and Analysis

In this section, we first present the setup for our experiments to measure the performance of different string matching engines. Secondly, we analyze performance differences between three different algorithms in REtriever – *noSplit*, *singleSplit*, *multipleSplit*, and with the nfa-based and the naive string matching engine.

4.1 Experiment Setup

All experiments were conducted on a 1600MHz Intel Pentium III machine with 1.0 GB memory, 30 GB Hard Disk running Windows XP. We will next introduce our workload generator, which generate subscriptions and publications.

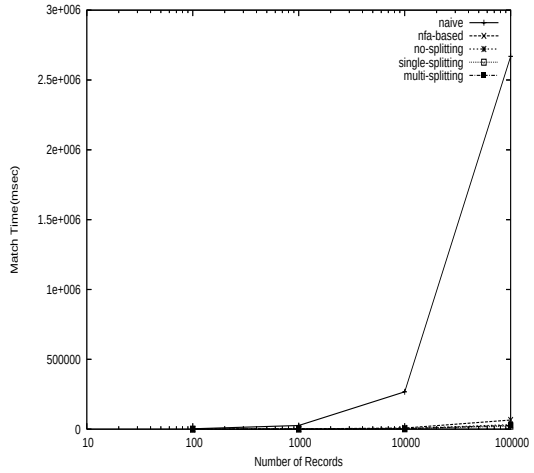
In the workload generator, we define the alphabet to be the English alphabet. The probability of each letter to appear in a subscription or publication is pre-defined. A string consists of *subStrLen* (see Table 4) number of letters. A subscription consists of *numSubStr* strings with *numStar*, *numQuestion* wildcard operators. Among *numSubStr* strings, *numOverlapStr* is drawn from an overlap string pool, which consist of *numOverlap* strings. We can specify the position of overlaps to be prefix, suffix or general. A publication is constructed from *numMatch* number of subscriptions drawn from the *numSubscription* subscriptions we generated. Then replace the wildcards in subscriptions with appropriate literal or string, and concatenate those subscriptions with extra strings in between them.

Parameters for Workload Generator	
parameter	controled value
numSubscription	number of subscriptions to be inserted to DFA
numPublication	number of publication to be matched
numOverlap	number of total overlap strings
numStar	number of star operators in each subscription
numQuestion	number of question operators in each subscription
numOverlapStr	number of overlap strings in each subscription
numMatch	number of subscription matches each publication has
numSubStr	number of strings in each subscription
subStrLen	length of each string
overlapPos	whether overlap is suffix, preffix or general

Table 4: Parameters for workload generator

4.2 Results and Analysis

In this section, we will present a set of preliminary experimental results. In most of these experiments, we vary one parameter of the workload generator, while all other parameters remain the same. With all experiments, the following parameters are fixed values.



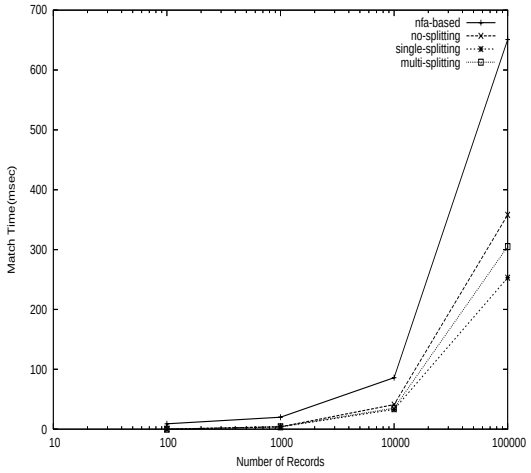


Figure 2-1: Vary Number of Subscriptions (with naive) Figure 2-2: Vary Number of Subscriptions

Parameters with fixed values	
parameter	fixed value
numPublication	100
numOverlap:numSubscription ratio	1:50
numMatch	3
subStrLen	4

Table 5: Parameters with fixed values through out all experiments

4.2.1 Vary The Number Of Subscriptions

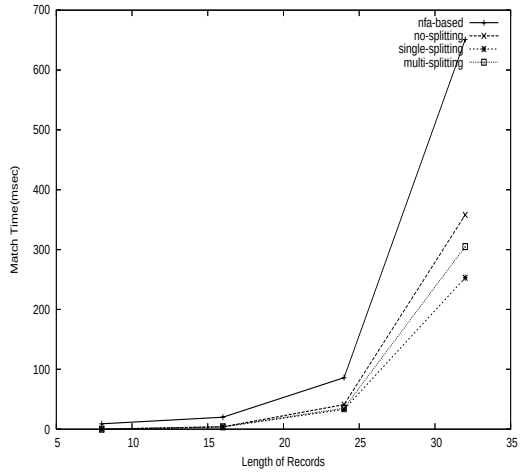
In experiment 1, we increase the number of subscriptions from 100 to 100,000, while each subscription consists of 1 star and 1 question mark operator and 3 strings among which 1 of them are general overlap.

In Figure 2-1, line "naive" denotes the matching time performance for the naive string matching engine. As we can see, the matching time for the naive algorithm increase much faster than all other algorithms when we scale the number of subscriptions. Thus, the naive algorithm is not scalable as compared to the others. In the rest of our presentations, we exclude the performance of this algorithm from the rest of the results.

Figure 2-2 is a zoom-in graph of Figure 2-1, without the naive algorithm. The line "yfilter" denotes the matching time performance for yfilter, while line "no-splitting", "single-splitting", "multi-splitting" denotes the performance for matching engines without splitting, and with singleSplit or multipleSplit algorithms. As we can see, at each data point, the matching time is halved using the REtriever algorithms from the time used for yfilter. The matching time for one publication matching 100,000 subscriptions with length 14 is 650 ms for yfilter, 350 ms for nosplit, 300 ms for multisplit and 250 ms for singlesplit. Results also show that REtriever algorithms are scalable with an increase in the number of subscriptions.

4.2.2 Vary The Length Of Subscriptions

In experiment 2, we increase the length of the subscriptions from consisting of 8 to 32 characters, while keeping numSubscription at 10000, numOverlap at 200, and each subscription has 1 star, 1 question wildcard operator and 2 overlap strings. Results show that the matching time for the nfa-based algorithm is increasingly longer than the others . It is interesting to note the crossing point of performance lines for singleSplit and multipleSplit, which illustrate the tradeoff of more splitting point to share more



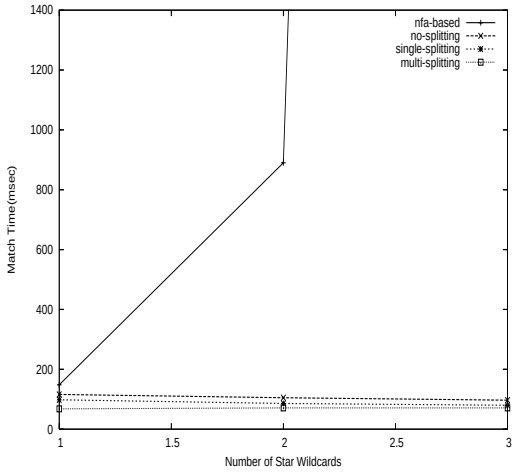


Figure 3: Vary Length of Subscriptions Figure 4: Vary Number of Star Operators in Subscriptions

common processing and the overhead of recombination of matchings. When the subscriptions are short, singleSplit use less time to match, while the length of each subscription is greater than 16, multipleSplit performs much better than singleSplit. This is because, when the subscription is short, a single splitting point is enough to exploit the general overlap strings, and more splitting points which further reduce the number of states, also increase the overhead of recombination, which becomes the dominant factor when the subscription is short. But when the subscription is long, multipleSplit algorithm performs better. (See Figure 3)

4.2.3 Vary The Number Of Star Operators In Each Subscription

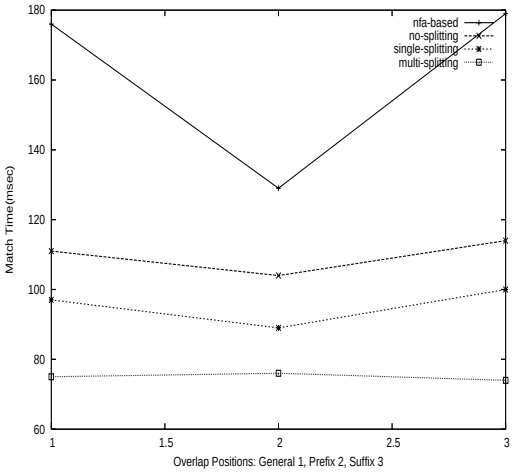
We design this experiment to show the benefit of using DFA-based structure in our algorithms. With numPublication = 100000, numOverlap = 200, and each subscription consists 6 strings with 3 general overlap strings. We increase the number of star operator from 1 to 3, and as seen in Figure 4, yfilter's matching time increases very fast, while the matching time of REtriever algorithms keep roughly the same. This experiment demonstrates that star wildcard operator is expensive in evaluation, which is the key reason why we use keyword matching with recombination technique for our subscription matching.

4.2.4 Vary The Position Of Overlaps In Each Subscription

In this experiment, we vary the positions of overlap strings in each subscriptions: general, prefix, suffix. We set numPublication to 10000, numOverlap to 200, each subscription has 6 strings with 1 overlap string, and 1 star, 1 question wildcard operators. For all three kinds of positions of overlap strings, the REtriever algorithms all outperform the nfa-based engine. While the REtriever algorithms have roughly the same matching time for all overlap positions, the nfa-based engine performs worse in general and suffix overlap compare with prefix overlap, which demonstrates that the REtriever does exploit general overlaps, while the nfa-based engine only exploits the prefix overlap.

4.2.5 Vary The Number Of Overlap Strings In Each Subscription

This experiment aimes to demonstrate the performance variance while increase the percentage of overlap strings in each subscriptions. We generate 200 general overlaps, 10000 subscriptions, each with 5 strings and 1 question mark wildcard operator. As we can see in Figure 6, the matching time for yfilter keeps at 190 ms for one publication, but for singlesplit, it drops from 160 ms to 110 ms, and multisplit drops



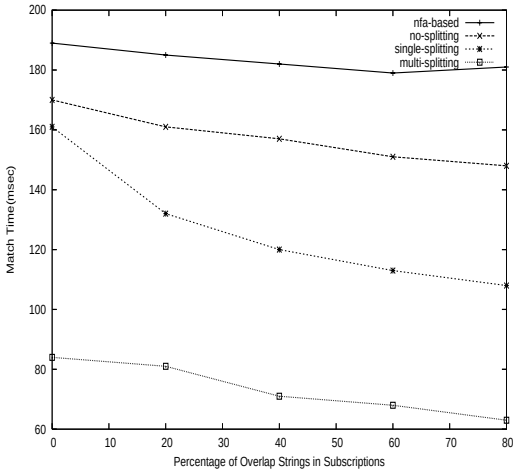


Figure 5: Vary Positions of Overlaps in Subscriptions Figure 6: Vary the Percentage of Overlaps

from 83ms to 62ms. The reason why multisplit perform still very well with zero overlap strings in the subscription is because, shorter overlap strings are prevalent in the subscription even though we specify no overlap string of length=4 in the subscription and multisplit is good in exploiting these short overlaps and thus reduce the machine size and matching time. On the other hand, singlesplit only allows one splitting point, thus with longer overlap strings, it will perform much better.