Efficient Information Extraction over Evolving Text Data

Fei Chen [1], AnHai Doan [1], Jun Yang [2], Raghu Ramakrishnan [3]

[1]*University of Wisconsin-Madison,* [2]*Duke University,* [3]*Yahoo! Research*

***Abstract*— Most current information extraction (IE) approaches have considered only static text corpora, over which we typically have to apply IE only once. Many real-world text corpora however are dynamic. They evolve over time, and to keep extracted information up to date, we often must apply IE repeatedly, to consecutive corpus snapshots. We describe Cyclex, an approach that efficiently executes such repeated IE, by recycling previous IE efforts. Specifically, given a current corpus snapshot U, Cyclex identifies text portions of U that also appear in the previous corpus snapshot V. Since Cyclex has already executed IE over V, it can now recycle the IE results of these parts, by combining these results with the results of executing IE over the remaining parts of U, to produce the complete IE results for U. Realizing Cyclex raises many challenges, including modeling information extractors, exploring the trade-off between runtime and completeness in identifying overlapping text, and making informed, cost-based decisions between redoing IE from scratch and recycling previous IE results. We describe initial solutions to these challenges, and experiments over two real-world data sets that demonstrate the utility of our approach.**

I. Introduction

Over the past decade, the problem of information extraction (IE) has received significant attention. Given a *text corpus* (e.g., a collection of emails, Web pages, etc.), many effective solutions have been developed to extract information from the corpus, and much progress has been made [23], [5], [7], [2].

Most of these IE solutions have considered only *static* text corpora, over which we typically have to apply IE only *once*. In practice, however, text corpora often are *dynamic*, in that documents are added, deleted, and modified. They evolve over time, and to keep extracted information up to date, we often must apply IE *repeatedly*, to consecutive corpus snapshots. Consider for example DBLife, a structured portal for the database community that we have been developing [18]. DBLife operates over a text corpus of 10,000+ URLs. Each day it recrawls these URLs to generate a 120+ MB corpus snapshot, and then applies IE to this snapshot to find the latest community information (e.g., which database researchers have been mentioned where in the past 24 hours). As another example, the Impliance project at IBM Almaden seeks to build a system that manages all information within an enterprise [21]. This system must regularly recrawl the enterprise intranet, and then apply IE to the recrawled data to infer the latest information. See [9], [10], [24] for other examples of dynamic text corpora.

Despite their pervasiveness, no satisfactory solution exists currently for IE over dynamic text corpora. Given such a corpus, the common solution is to apply IE to each corpus snapshot *in isolation, from scratch*. This solution is simple, but highly inefficient, with limited applicability. For example, in DBLife reapplying IE from scratch takes 8+ hours each day, leaving little time left for higher-level data analysis. As another example, time-sensitive applications (e.g., stock, auction, intelligence analysis) often want to refresh information quickly, by recrawling and reextracting, say, every 30 minutes. In such cases applying IE from scratch is inapplicable if it already takes more than 30 minutes. Finally, this solution is ill-suited for *interactive debugging* of IE applications over dynamic corpora, because such debugging often requires applying IE repeatedly to multiple corpus snapshots. Thus, given the growing need for IE over dynamic text corpora, it has now become crucial to develop efficient IE solutions for these settings.

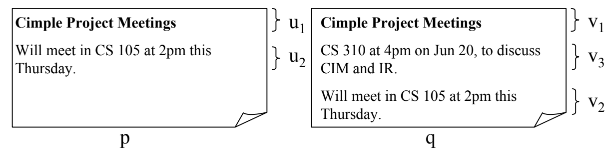

Fig. 1. Two pages of the same URL, retrieved at different times

In this paper we present Cyclex (Recycling Extraction), as a solution. The key idea underlying Cyclex is to recycle previous IE results, given that consecutive snapshots of a text corpus often contain much overlapping data. The following example illustrates this idea:

Example 1. *Consider a tiny corpus of a single URL that lists project meetings. Fig. 1 shows a snapshot of this corpus, which is just a single data page p (of the above URL), crawled today. Suppose that we have applied an extractor E to this snapshot, to extract the tuple (CS 105,2pm) which specifies a meeting. Suppose tomorrow we crawl the above URL to obtain another corpus snapshot, which is the page q shown in Fig. 1. Then to extract meetings from q, current solutions would apply extractor E to q from scratch, and produce tuples (CS 105,2pm) and (CS 310,4pm).*

In contrast, Cyclex *tries to recycle the IE results of p. Specifically, it starts by "matching" q with p, to find text regions of q that also appear in p. Suppose it finds two regions v_1 and v_2 of q that also appear as u_1 and u_2 of p, respectively (see Fig. 1).* Cyclex *then does not apply E to v_1 and v_2, but instead copies over the mentions of u_1 and u_2.* Cyclex *then applies E only to v_3, the sole region of q that does not appear in p. The savings come from not having to apply E to the entire page q.*

While appealing, realizing the above idea raises difficult challenges. The first challenge is that we cannot simply just copy mentions over, e.g., from regions u_1 and u_2 of page p to v_1 and v_2 of page q, as discussed in Example 1. To see why, suppose a particular extractor E is such that it only extracts meetings if a page has fewer than five lines (otherwise it produces no meetings). Then none of the mentions of page p can be copied over to page q, which has more than five lines. In general, which mentions can be copied "safely" depends on certain properties of extractor E. Thus, we must model certain properties of extractor E, so that we can (a) exploit these properties to reuse certain mentions, and (b) prove that reusing will produce the same set of mentions as applying IE from scratch. In this paper we define a small set of such properties, show that many practical extractors exhibit these properties (see Section III), and develop incremental re-extraction techniques by exploiting these properties.

Our second challenge is how to "match" two pages, e.g., p and q in Example 1, to find overlapping text regions. We first develop ST, a powerful suffix-tree based matcher, and prove that this matcher achieves the most complete result, i.e., finds all largest possible overlapping regions. We then show that an entire spectrum of matchers exists, with matchers trading off the completeness of the result for runtime efficiency (see Section V). Since no matcher is always optimal, we provide Cyclex with a set of alternative matchers (more can be added easily), and a way to select a good one, as discussed below.

Since dynamic text corpora can easily contain tens of thousands or millions of data pages, we must also develop efficient solutions for reusing mentions and applying extractor E to non-overlapping text, in the presence of a large amount of disk-resident data. We must also consider how to efficiently interleave these steps with the step of matching data pages (see Section VI).

Finally, addressing the above challenges results in a space of execution plans, where the plans differ mainly on the page matcher employed. Thus, in the final challenge we must develop a cost model and use it to select the optimal plan. Unlike RDBMS settings, our cost model is extraction-specific. In particular, it tries to model the rate of change of the text corpus, and the run time and result size of extractors and matchers, among others (see Section VII).

In summary, we make the following contributions:

- We show that it is possible to exploit past IE work to significantly speed up IE over evolving text. As far as we know, Cyclex is the first solution to this important problem.
- We show how to model certain common properties of information extractors and how to exploit these properties to reuse past IE and to guarantee the correctness of our approach.
- We show that a natural tradeoff exists for finding overlapping text regions. We examine the spectrum of choices and develop a powerful suffix-tree based solution.
- We show how to estimate cost for each of the points in the spectrum, to find an IE plan with minimal estimated time.
- We conduct extensive experiments over two real-world data sets that demonstrate the utility of our approach.

II. Related Work

The problem of information extraction has received much attention (see [23], [5], [2] for recent tutorials). The main focus so far has been on improving the accuracy and runtime of information extractors. But recent work has also started to consider how to manage such extractors in large-scale IE-centric applications [5], [2]. Our work fits into this emerging direction, which is described in more detail in [2].

Once we have extracted entity mentions, we can perform additional analysis, such as mention disambiguation (a.k.a. record linkage, e.g., [16]). Thus, such analyses are higher level and orthogonal to our current work.

While we have focused on IE over *unstructured text*, our work is related to wrapper construction, the problem of inferring a set of rules (encoded as a wrapper) to extract information from *template-based Web pages*. Since wrappers can be viewed as extractors (as defined in Section III), our techniques can potentially also apply to wrapper contexts. In this context, the knowledge of page templates may help us develop even more efficient IE algorithms.

Several recent works have also considered evolving text data, but in different problem contexts. The work [20] considers how to repair a wrapper (so that it continues to extract semantically correct data) as the underlying page templates change, and the work [12] considers how to incrementally update an inverted index, as the indexed Web pages change.

Recent work [11], [14] has also exploited overlapping text data, but again in different problem contexts. These works observe that document collections often contain overlapping text. They then consider how to exploit such overlap to "compress" the inverted indexes over these documents, and how to answer queries efficiently over such compressed indexes. In contrast, we exploit the IE results over the overlapping text regions to reduce the overall extraction time.

The problem of finding overlapping text regions is related to detecting duplicated Web pages. Many algorithms have been developed in this area (e.g., [13], [17], [4]). But when applied to our context they do not guarantee to find *all* largest possible overlapping regions, in contrast to the suffix-tree based algorithm developed in this work. Several suffix tree algorithms have been widely used to find matching substrings in a given input string [8]. Here we have significantly extended these algorithms, to develop one that can efficiently detect *all maximal matching regions* (i.e., substrings) between two given strings, in time linear in the total length of these two strings.

Finally, optimizing IE programs and developing IE-centric cost models have also been considered in several recent papers [22], [19], [3]. These efforts however have considered only static corpus contexts, not dynamic ones as we do in this paper.

III. Problem Definition

Data Sources, Pages, & Corpus Snapshots: Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ be a set of *data sources* considered by an application A. We assume that A crawls these sources at regular intervals to retrieve sets of *data pages*. For example, DBLife considers 10,000+ data sources, each specified with a URL, and crawls these URLs (each to a pre-specified depth) each day to retrieve a set of 14,000+ Web pages. We will refer to P_i — the set of data pages retrieved at time i — as the *i-th snapshot* of the evolving text corpus $\mathcal{S}$.

Entities, Attributes, & Mentions: Data pages often mention *entities*, which are real-world concepts, such as person, paper, and meeting. We represent each entity type e with a set of *attributes* $a_1, \ldots, a_k$, which can be atomic (e.g., meeting room) or set-valued (e.g., topics).

Given a data page p, we refer to a consecutive sequence of characters in p as a *string*, or a *text fragment*, or a *region* (we will use these notions interchangeably). We use $p[i..j]$ to denote the string s that starts with the i-th character and ends with the j-th characters of p. In this case, we will also say $s.start = i$ and $s.end = j$.

A *mention* of an atomic (set-valued) attribute a is then a string in p (a set of strings in p) that refers to a. We can now define an entity mention as follows:

Definition 1 (Entity mention). *Let p be a data page, and $a_1, \ldots, a_k$ be the attributes of an entity type e. Then a mention of an instance of entity type e is a tuple $m = (m_1, \ldots, m_k)$, where each m_i, $i \in [1, k]$, is either a mention of a_i in page p, or the special value "nil," indicating that a mention of a_i cannot be extracted from p. We also define $m.start = \min_{i=1}^k m_i.start$ and $m.end = \max_{i=1}^k m_i.end$.*

Example 2. *Suppose the entity type "meeting" has three attributes: room, time, and topics. Then tuple (CS 310, 4pm, {CIM,IR}) is a mention of "meeting" in page q of Fig. 1. String s = "CS 310" (where $s.start = 25$ and $s.end = 30$) is a mention of attribute "room." "4pm" is a mention of "time," and the set of strings {"CIM," "IR"} is a mention of "topics."*

Extractors: Real-world IE applications extract mentions of one or multiple entity types from data pages. As a first step, in this paper we consider extracting mentions of a single entity type e (e.g., meeting). To extract such mentions, current applications usually employ an extractor E, which is typically a learning-based program, or a set of extraction rules encoded in, say, a Perl script [2]. We assume that E extracts mentions from *each data page in isolation*, e.g., extracting meetings as in Fig. 1. Such per-page extractors are pervasive (e.g., constituting 94% of extractors in the current DBLife, see [2], [22] for many examples). Hence, we start with such extractors, leaving more complex extractors (e.g., those that extract mentions that span multiple pages) for future work. We can now define extractors considered in this paper as follows:

Definition 2 (Extractors). *Let $a_1, \ldots, a_k$ be the attributes of an entity type e. Then an extractor $E : p \rightarrow M$ takes as input a data page p and produces as output a set M of mentions of e in page p, where each mention is of the form $(m_1, \ldots, m_k)$ as described in Definition 1.*

Modeling Properties of Extractors: Recall from the introduction that we must model certain properties of extractors, so that we can reuse mentions and prove the correctness of our algorithm. We now describe two such properties: *scope* and *context*. To motivate scope, we observe that attribute mentions of an entity often appear in *close proximity* in text pages. Consequently, an extractor often starts by extracting attribute mentions, then combines the mentions and prunes those combinations that span more than a maximal length α.

Example 3. *Suppose we apply E to page q in Fig. 1 to extract (room,time). E may start by extracting all room mentions: "CS 310," "CS 105," then all time mentions: "4pm," "2pm." E then pairs room and time mentions, and prunes pairs that are not found within, say, a length of 100 characters. Thus, E returns only the pairs (CS 310,4pm) and (CS 105,2pm).*

Thus, we can formalize the notion of scope as follows:

Definition 3 (Extractor scope). *An extractor E has scope α iff for any mention m produced by E we have $(m.end - m.start) < \alpha$.*

To motivate context, we observe that when extracting mentions, many extractors examine only small "context windows" to both sides of a mention, as the following example illustrates:

Example 4. *Let E be an extractor for (room,time,topics). Suppose E produces string X as a topic if (a) X matches a pre-defined word (e.g., "IR"), and (b) the word "discuss" or "topic" occurs within a 30-character distance, either to the left or to the right of X. Then we say that the context of topic mentions is 30 characters. That is, once E has extracted X as a topic, then no matter how we perturb the text outside a 30-character window of X (on both sides), E would still recognize X as a valid topic mention.*

Let m be a mention produced by an extractor E in page p. Then we formalize the notion of context as follows:

Definition 4 (β-context of mention & extractor context). *The β-context of m (or context for short when there is no ambiguity) is the string $p[(m.start - \beta)..(m.end + \beta)]$, i.e., the string of m being extended on both sides by β characters. Extractor E has a context β iff for any m and p' obtained by perturbing the text of p outside the β-context of m, applying E to p' still produces m as a mention.*

We assume that each extractor E comes with a scope α and a context β. These values can be supplied by whoever implementing E or knowing how E works (e.g., the application builder, after examining E's description or code). As we show in the experiments, α and β do not have to be "tight" in order for us to benefit from recycling IE results. However, the "tighter" (i.e., smaller) these values are, the larger the benefits.

Problem Definition: We can now describe our problem as follows. Let $P_1, \ldots, P_n$ be consecutive snapshots of a text corpus, E be an extractor with scope α and context β, and $M_1, \ldots, M_n$ be the set of mentions extracted by E from $P_1, \ldots, P_n$, respectively. Let P_{n+1} be the corpus snapshot immediately following P_n. Then develop a solution to extract the set of mentions M_{n+1} from P_{n+1} in a minimal amount of time, by utilizing $P_1, \ldots, P_n$, α, β, and $M_1, \ldots, M_n$. In the rest of the paper we describe Cyclex, our solution to this problem.

IV. The Cyclex Solution Approach

To describe Cyclex, we begin with two notions:

Definition 5 (Old region & maximally old region). *A region r in a data page p of snapshot P_{n+1} is an old region if it occurs in a page q of snapshot P_n. r is a maximally old region if it cannot be extended on either side and still remains an old region.*

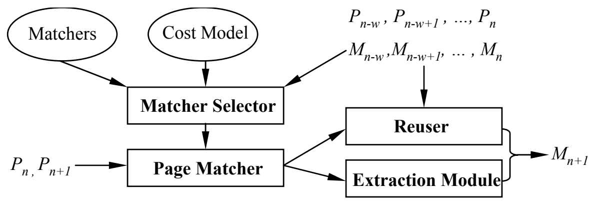

Fig. 2. The Cyclex architecture

To extract mentions from P_{n+1}, Cyclex then considers each page p in P_{n+1} and "matches," i.e., compares p with pages in P_n, to find old regions of p. Next, it uses the old regions to identify *copy regions* and *extraction regions* of p (see Section VI). Cyclex then applies extractor E only to the extraction regions, and copies over the mentions of the copy regions.

Since pages retrieved (in consecutive snapshots) from the same URL often share much overlapping data, to find old regions of p, Cyclex currently matches p only with q, the page in P_n that shares the same URL with p. (If q does not exist, then Cyclex declares that p has no old regions.) Section VIII shows that the choice of matching pages with the same URL already significantly reduces IE time. Considering more complex choices (e.g., matching p with all pages in P_n) is an ongoing research.

We call algorithms that match p and q to find old regions in p *page matchers*. Sections V shows that such matchers span an entire spectrum, trading off result completeness for runtime, and that no matcher is always optimal. For example, the ST matcher described below returns all maximally old regions, thus providing the most opportunities for recycling past IE results. But it may also incur more runtime than matchers that return only some old regions. So, a priori we do not know if it would be better than these other matchers.

The above result leads to the Cyclex architecture in Fig. 2. Given snapshot P_{n+1}, the matcher selector employs a cost model (that utilizes statistics computed over the past w snapshots) to select a page matcher from a library of matchers. The page matcher then finds old regions of pages in P_{n+1}. Next, the extraction module applies extractor E to extraction regions of pages in P_{n+1}, and the reuser copies over mentions of the copy regions. Cyclex then combines the results of both the extraction module and the reuser to produce the final IE result for P_{n+1}. The next three sections describe the matchers, the reuser and extraction module, and the matcher selector in detail.

V. The Page Matchers

Recall from Section IV that a page matcher compares pages p and q to find old regions of p. We have provided the current Cyclex with three page matchers: DN, UD, and ST (more matchers can be easily plugged in as they become available). DN incurs zero runtime, as it immediately declares that page p has no old region. Cyclex with DN thus is equivalent to applying IE from scratch to P_{n+1}.

UD employs an Unix-diff-command like algorithm [15], which splits pages p and q into lines, then employs a heuristic to find common lines. Thus, UD is relatively fast (takes time linear in $|p| + |q|$), but finds only some old regions. We omit further description for space reason, but refer the reader to [15].

ST is a novel suffix-tree based matcher that we have developed, which finds *all maximal old regions* of p using time linear in $|p| + |q|$. ST and DN thus represent the two ends of a spectrum of matchers that trade off the result completeness for runtime efficiency, while UD represents an intermediate point on this spectrum.

In the rest of this section we describe ST in detail. Roughly speaking, ST inserts all suffixes of q and p into one suffix tree T [8]. As we insert each suffix of p, T helps us identify the longest prefix of this suffix that also appears in q. To realize this intuition, however, we must handle a number of intricacies, so that we can locate all maximal old regions without slowing down ST to quadratic time.

A. Suffix Tree Basics

The suffix tree for a string q is a tree T with $|q|$ leaves, each describing a suffix of q. T must satisfy the following: (1) Each non-root internal node has at least two children. (2) Each edge is labeled with a nonempty substring of q, and no two edges out of a node can have labels beginning with the same character. (3) The *path label* of a node is the concatenation of all edge labels on the path from the root to this node; each suffix of q corresponds to the path label of a leaf. (4) Each non-root internal node with path label λu (where λ is a single character and u is a string) has *a suffix link* to the node with path label u; the root has a suffix link to itself. Fig. 3(a) shows the suffix tree for "ababbabaab$," where symbol $ terminates the string. Suffix links are showed as dotted lines.

To construct a suffix tree for q, we insert all suffixes of q one by one into an initially empty tree. For example, the suffixes of "ababbabaab$" are "ababbabaab$," "babbabaab$," "abbabaab$," ..., "b$." Let s_i denote $q[i..|q|]$, the i-th suffix of q. Conceptually, to insert s_i, we first look up s_i, matching s_i against edge labels as we go down the tree until no more characters can be matched. If lookup stops at a node, we insert s_i as a leaf below that node; if lookup stops in the middle of an edge, we add a new node to split the edge right before the point where it diverges from s_i, and then insert s_i as a leaf of the new node.

Unfortunately, if we insert every s_i by starting the lookup from the root, we would end up with a quadratic-time algorithm. The secret to more efficient suffix-tree construction is to exploit the suffix links, which allow us to leverage the matching work we have already done when inserting s_{i-1}. We now sketch the construction algorithm below.

Suppose we have just inserted s_{i-1} as a leaf child of node α_{i-1}; note that α_{i-1} is the only possibly new internal node created during the insertion of s_{i-1}. Next, we want to insert s_i into the suffix tree, and ensure that α_{i-1}'s suffix link is properly set up. To this end, we follow a series of **up**, **across**, and **down** moves in the suffix tree. Suppose α_{i-1}'s path label is λu, where λ is a single character; note that u is a prefix of s_i. First, we go **up** from α_{i-1} to its parent θ, whose path

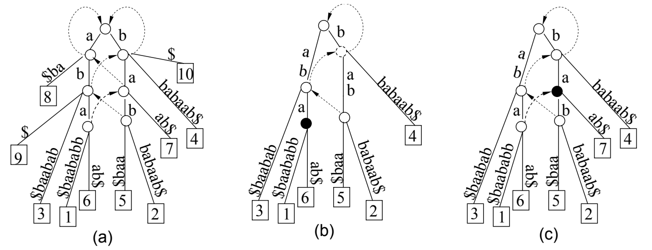

Fig. 3. An example of inserting a suffix

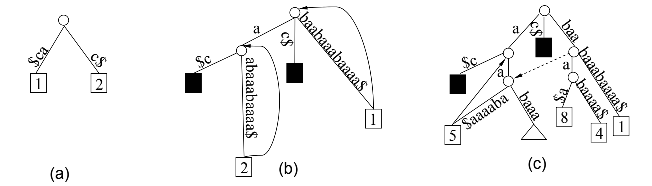

Fig. 4. An example of prefix links

label is $\lambda u'$, where u' is a prefix of u. Then, following the suffix link of θ, we go **across** to θ', whose path label is u'. Next, starting from θ', we go **down** the tree, matching $u - u'$, the substring of u that follows u'. We end up with node β with path label u, to which we set the suffix link of α_{i-1}. If β does not currently exist in the tree, we create β by splitting the edge right where the matching of $u - u'$ stops; we then add s_i (which, as we recall, begins with u) as a child of β. On the other hand, if β already exists in the tree, we continue to go **down** the tree from β, matching $s_i - u$, the substring of s_i that follows u, and insert s_i at the point where matching stops; this process may create a new internal node. It can be shown that this construction algorithm is linear in the size of the string [8].

Fig. 3.b shows the suffix tree before inserting s_7 of "ababbabaab$." The only new internal node in the tree now is α_6 (the dark node). The path label for the dark node is "aba" and u is "ba." First, we go **up** from the dark node to its parent θ. Then we follow the suffix link of θ and go **across** to θ' (the dotted node). Notice that we skip looking up the first "b" in s_7 by following the suffix link. Next, from the dotted node, we go **down** the tree, matching the substring of u that follows "b." The matching stops in the middle of the edge with label "ab" out from the dotted node, which leads to splitting the edge and creating a new node β. In Fig. 3.c, β is the dark node. We then insert the leaf corresponding to s_7 as the child of β. Finally, we set up the suffix link from α_6 to β.

B. ST: The Suffix-Tree Matcher

ST starts by building a suffix tree T for q, the old page, as described in Section V-A. Next, it inserts the suffixes of p, the new page, one by one, into T, and reports each maximal old region as soon as it is detected. To carry out this second step, we make important extensions to both the insertion procedure and the suffix tree structure. First, we augment suffix-tree nodes with *prefix links*, which are crucial to finding old regions efficiently. We also show how to set up these links during construction. Second, we show how to detect *maximal* old regions without introducing additional performance overhead. We describe these two extensions next.

Finding Old Regions Using Prefix Links: By inserting s'_i, the i-th suffix of p, into T, we can easily find the longest common prefix between s'_i and any suffixes that have been already inserted. Let h_i denote this string, which corresponds to node α'_i, the parent of the leaf corresponding to s'_i. On the other hand, what we are looking for, r_i, is the longest common prefix between s'_i and any suffix of q, the old page. Unfortunately, r_i may not be the same as h_i, because the suffix tree at this time additionally contains suffixes $s'_1, \ldots, s'_{i-1}$ of p, inserted earlier than s'_i.

However, it is not difficult to see that r_i must be a prefix of h_i, because h_i by definition cannot be shorter than any common prefix between s'_i and suffixes of q. To find r_i, we need to locate the last node on the path from the root to α'_i with at least one descendant leaf corresponding to a suffix of q. Efficiently finding this node, which we denote by δ_i, turns out to be quite tricky. One might think that we should encounter δ_i as we go down T when inserting s'_i. However, recall from Section V-A that we use suffix links to avoid quadratic-time construction; thus, we reach α'_i without starting from the root, and possibly without passing through δ_i.

To ensure the efficiency of locating δ_i, we add a *prefix link* for each node of T. The prefix link of node γ, denoted $L_p(\gamma)$, points to its lowest ancestor with at least one descendant leaf corresponding to a suffix of q. If γ itself has at least one descendant leaf corresponding to a suffix of q, we do not explicitly store a prefix link, but we implicitly understand that $L_p(\gamma)$ points to γ itself.

We construct prefix links as follows. Suppose we have created the suffix tree T for q. Then there are no explicit prefix links yet (i.e., every node's prefix link implicitly points to itself) because every node leads to a suffix of q. Now, for every new leaf γ we create (for a suffix of p), we let $L_p(\gamma)$ point to the same node as γ's parent's prefix link. For an internal node γ created by splitting an edge pointing to node γ', if $L_p(\gamma')$ points to γ' itself, we let $L_p(\gamma)$ point to γ itself; otherwise, we set $L_p(\gamma) = L_p(\gamma')$. For example, Fig. 4.(a) shows the suffix tree for q = "ac$." Fig. 4.(b) shows the prefix links (in solid arrows) after we insert the first two suffixes of p = "baabaaabaaaa$." The black leaves are corresponding to the suffixes of q. For those nodes which have a prefix link to itself, we do not show the links.

With prefix links, we now show how to find the longest common prefix between a suffix s'_i of p and any suffix of q, while inserting s'_i into the suffix tree. After a leaf has been created for s'_i, we check the node δ_i pointed to by the prefix link of the leaf's parent. The path label of δ_i gives us the largest possible old region matching a prefix of s'_i. For example, Fig. 4.(c) shows the state of the suffix tree before we inserting s'_9, the ninth suffix of p, "aaaa$." We omit the irrelevant part of the tree (in triangle) and links from the figure. Following the standard suffix-tree construction algorithm, we first use the suffix link (in dotted arrow) of the parent node of α_8 to go **across** to θ'. Then we go **down** the tree and match

Algorithm 1 ST

Input: old data page q, new data page p
Output: all maximal old regions R in p
$T \Leftarrow$ buildSuffixTree(q)
//initialization
$R \Leftarrow \emptyset$
$\alpha'_0 \Leftarrow T.root$
for each suffix s'_i of p **do**
  //locate the node corresponding to the longest common prefix of s'_i and any suffixes in T and set up the suffix link of α'_{i-1}
  $\alpha'_i \Leftarrow longestCommonPrefix(s'_i, T, \alpha'_{i-1})$
  if α'_i is a new node created by splitting an edge pointed to γ **then**
    //set up the prefix link of α'_i
    if $L_p(\gamma) = \gamma$ **then**
      $L_p(\alpha'_i) \Leftarrow \alpha'_i$
    else
      $L_p(\alpha'_i) \Leftarrow L_p(\gamma)$
    end if
  end if
  Insert leaf η'_i as a child of α'_i
  $L_p(\eta'_i) \Leftarrow L_p(\alpha'_i)$
  //find r_i, the longest common prefix of s'_i and any suffix of q, using prefix link of α'_i
  $r_i \Leftarrow p[i..i + pathLength(T.root, L_p(\alpha'_i)) - 1]$
  //compare the ending positions of r_i and r_{i-1} to check if r_i is a maximal old region
  if $r_i.end > r_{i-1}.end$ or $i = 1$ **then**
    $R \Leftarrow R \bigcup \{r_i\}$
  end if
end for

the substring of u = "aaa" that follows "aa." The matching stops in the middle of the edge with label "abaaaa\$," which leads to splitting the edge and creating a new internal node α'_9 with path label "aaa." The leaf for s'_9 is then inserted below α'_9. The prefix links of α'_9 and the leaf point to the same node pointed to by the prefix link (in solid arrow) of leaf 5. We then use the prefix link of α'_9 to find "a," the longest common prefix between s'_9 and any suffix of q.

Detecting Maximal Old Regions: So far, we have seen how to find, for each suffix of p, the longest common prefix between it and all suffixes of q. However, these prefix matches are not necessarily maximal old regions (cf. Definition 5). Although such matches cannot be extended any further to the right, it may be possible to extend them to the left. How do we then find the globally maximal old regions?

We make two observations. First, any maximal old region must be the longest common prefix between some suffix of p and suffixes of q. The second observation is captured by the following lemma:

Lemma 1. *Let $p[i-1..j]$ be the longest common prefix between s'_{i-1}, the $(i-1)$-th suffix of p, and any suffix of q. Let $p[i..k]$ be the longest common prefix between s'_i and any suffix of q. Then, $p[i..k]$ is a maximal old region if and only if $k > j$.*

The above observations lead to a simple, efficient method for identifying all maximal old regions in a streaming fashion while we process suffixes of p one by one. After processing the i-th suffix of p and finding the longest common prefix r_i between it and q's suffixes, we compare the end position of r_i with that of r_{i-1} (identified while processing the $(i-1)$-th suffix of p). As long as the end position has advanced, we output r_i as a maximal old region.

The complete psudocode for ST is listed in Algorithm 1.

Runtime Complexity: We conclude this section by stating the complexity of our suffix-tree matching algorithm in the following theorem. The dominating cost, in terms of both time and space, comes from standard suffix tree construction. Our implementation uses balanced search trees to manage parent-child relationships in the suffix tree, which implies that an additional time cost factor $c = \mathrm{O}(\log A)$, where A is the size of the alphabet. Other alternatives with $c = \mathrm{O}(1)$ also exist, but we have found our implementation to work well when A is very large. This is probably because suffix trees with balanced search trees to manage parent-child relationships take smaller space and thus lead to fewer cache misses.

Theorem 1. ST *takes $\mathrm{O}((|p| + |q|)c)$ time and $\mathrm{O}(|p| + |q|)$ space, where c is the cost of looking up a child of a node in the suffix tree.*

VI. The Reuser + Extraction Module

Suppose Cyclex has selected a page matcher M (see Section IV). We now describe how M works in conjunction with the reuser and the extraction module to recycle mentions and extract new ones. We face two key challenges. First, since corpus snapshots often are large, we must handle disk-resident data efficiently. Second, we must employ scope α and context β to identify precise text regions from which it is "safe" to copy mentions or to apply extractor E. To address these challenges, we proceed in the following three steps.

1. Find Copy Regions: We begin by reading pages from disk-resident P_{n+1} in a sequential manner. For each page p, we find $q \in P_n$ which shares the same URL with p. (If no such q exists, we simply apply extractor E to p.) Next, we apply M to p and q (in memory) to find old regions (see Section V).

Not all mentions in old regions (if we find any) are safe to be copied. This is illustrated by the following example.

Example 5. *Let q = "Dr. John Doe is a CS prof.". Suppose extractor E declares string n to be a person name if it is two capitalized words preceded by "Dr. ". Then E has context $\beta = 3$, and produces "John Doe" as a mention of q. Now consider p = "John Doe is a CS professor". Suppose M declares o = "John Doe is a CS prof" to be an old region of p. Then since "John Doe" is a mention (of q) in o, we may think that it will also be a mention of p. However, this is incorrect because applying E to p would produce no mention.*

In general, we can copy a mention only if both the mention (e.g., "John Doe") and its context (e.g., "Dr.") are contained in an old region. Specifically, if $p[c..c+k]$ is an old region because it matches $q[c'..c'+k]$, then we copy a mention m only if it is contained in the region $q[c'+\beta..c'+k-\beta]$. We refer to such regions, from which it is safe to copy mentions, as *copy regions*. We now describe finding copy regions, distinguishing two cases: disjoint old regions, and overlapping old regions.

- *Old regions are disjoint:* Let $r_1, \ldots, r_k$ be old regions of p (discovered by matcher M). We represent each r_i as a tuple $(id_p, id_q, s_p, s_q, l)$, where id_p and id_q are IDs of p and q, s_p and s_q are the start positions of the old region in p and q, respectively, and l is the length of the old region.

Suppose old regions represented by $r_1, \ldots, r_k$ are disjoint. Then we simply construct for each r_i a copy region h_i which

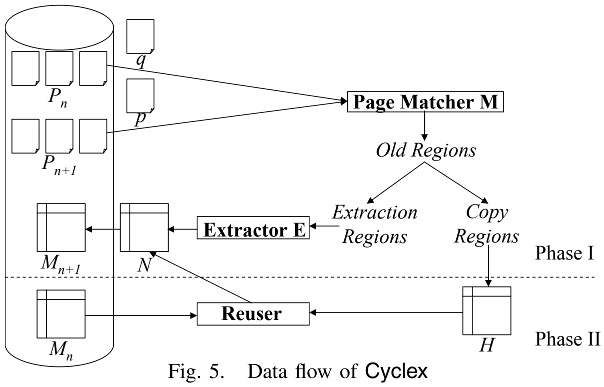

Fig. 5. Data flow of Cyclex

is a tuple $(id_p, id_q, s'_p, s'_q, l')$, where $s'_p = s_p + \beta, s'_q = s_q + \beta$, and $l' = l - 2\beta$. Next, we insert h_i into a memory-resident table H.

- *Old regions are overlapping:* In this case we extend the above algorithm so that we copy each mention in the overlapping regions only once. First, we construct a set of copy region candidates by chopping β characters at both ends of each old region, as we described in the disjoint case. Let the resulting set of regions be $r'_1, \ldots, r'_k$. This step gives us a set of regions where we are sure that if a mention is contained in one of those regions, it will be extracted by E from p, and thus it can be safely copied. However, since regions $r'_1...r'_k$ can overlap, a mention can be contained in more than one region and copied more than once. The following two steps ensure that any mentions contained in at least one of $r'_1...r'_k$ will be copied exactly once.

Let a and b be two overlapping regions from $r'_1, \ldots, r'_k$. Then a corresponds to a copy region candidate $p[i..j]$ and b corresponds to another copy region candidate $p[k..l]$ such that $i < k < j < l$. Then we discard a and b and generate instead the following regions: (1) regions c, d, e that corresponds to $p[i..k-1], p[k..j], p[j+1..l]$, respectively. These regions are created so that we can avoid copying mentions in region d twice. (2) regions f, g that corresponds to $p[k-\alpha..k+\alpha], p[j-\alpha..j+\alpha]$, respectively. These regions are created to catch any mention that may cross the splitting points k and j and thus is not contained in any of the above regions.

We insert the tuples corresponding to these regions into table H. Fig. 5 shows the data flows of Cyclex for the step of finding copying regions in phase I.

2. Find Extraction Regions & Apply Extractor E: Let $c_1, \ldots, c_t$ be the copy regions of p, identified as in Step 1. We now find *extraction regions*, those regions of p on which we must apply extractor E, to ensure the correctness of Cyclex.

To obtain extraction regions, at a first glance, it appears that we can simply remove copy regions from p. However it is not difficult to construct examples where this would "remove too much," thus dropping mentions that we should have found for p. In general, we can prove that if $p[c..c+k]$ is an old region, then it is safe to remove only region $p[c+\gamma..c+k-\gamma]$, where $\gamma = 2\beta+\alpha-1$. We now describe finding extraction regions for two cases: disjoint old regions, and overlapping old regions.

- *Old regions are disjoint:* Let R be the set of disjoint old regions of p. We begin by initializing c, the start position of the next extraction region, to 1. Then we scan regions of R sequentially, in increasing value of their start positions. For each $r \in R$, we create $p[c..(r.s_p - 1 + \gamma)]$ as an extraction region. Then we update $c = r.s_p + r.l - \gamma$. The last extraction region ends at position $|p|$.
- *Old regions are overlapping:* In this case, the extraction regions identified by the above algorithm might not be minimal in the sense that if we remove some parts of the extraction regions, we can still guarantee correctness of Cyclex. Hence,we waste the time of applying E over the additional regions.

To ensure that an identified extraction region is not contained in any old region, we extend the algorithm for disjoint old regions case as follows. First, we repeatly concatenate any two overlapping old regions $p[i..j]$ and $p[k..l]$ if the length of the overlapping part is larger than γ. Without loss of generality, suppose $i < k < j < l$. Since $j - k \geq \gamma + 1$, the maximal length of the β-context of any mention extracted by E, the β-context of any mention across the two old regions $p[i..j]$ and $p[k..l]$ is either contained in $p[i..j]$ or $p[k..l]$, and thus the mention will be copied. Hence, we can ignore the adjacent boundaries of $p[i..j]$ and $p[k..l]$ when identifying extraction regions. We refer to the concatenated regions as *super old regions*. Let the set of super old regions be R'. Any mention such that both itself and its context is contained in a region $r' \in R'$ will be copied.

Next, we create a set of extraction regions to catch any mention that will not be copied. For each r' corresponding to $p[i..j]$ in R', we create a *removal* region $p[i+\gamma..j-\gamma]$. Since the length of the overlapping part of any two regions in R' is at most γ, the removal regions created at this step are disjoint. Let the set of removal regions be D. Finally, we remove D form p and the remaining set of regions are the extraction regions.

Once we have identified all extraction regions of a page p, we apply extractor E to these regions. To guarantee correctness of Cyclex, among all extracted mentions, we only retain those such that both the mentions and their contexts are contained in an extraction region. We then insert the retained mentions into a memory-resident table N. N is flushed to the disk-resident table M_{n+1} (which stores all mentions extracted from P_{n+1}) whenever it is full. Fig. 5 shows the data flow of Cyclex for the step of finding extraction regions and applying extractor E in phase I.

3. Copy Mentions from Copy Regions: We repeat step 1 and step 2 until we have processed all pages p in P_{n+1}. At this point, we have extracted mentions from all extraction regions. We have also stored all copy regions (actually, only the start- and end-positions of these regions, not the regions themselves) in table H. Now we must copy to N any mention that (a) exists in M_n (the IE result over the previous snapshot P_n) and (b) can be found in a region stored in H.

Since M_n can be large, we assume it is on disk. Furthermore, since each application may want to store the mentions in a particular order (for further processing, e.g., mention disambiguation), we do not assume any particular order for mentions in M_n. Rather, we proceed as follows. We perform a sequential scan of M_n. For each mention m of M_n, we immediately probe m against regions of table H (implemented as a hash table, with key id_q, s_q and l). In case of a hit, m appears in one of the copy regions, thus, we construct an appropriate mention m' of p (that correspond to m), then insert m' into table N. Fig. 5 shows the data flow of Cyclex for the step of copying mentions in phase II.

The following theorem states the correctness of Cyclex:

Theorem 2 (Correctness of Cyclex)**.** *Let M_{n+1} be the set of mentions obtained by applying extractor E from scratch to snapshot P_{n+1}. Then Cyclex is correct in that when applied to P_{n+1} it produces exactly M_{n+1}.*

VII. The Cost-Based Matcher Selector

We now describe how the matcher selector employs a cost model to select the best matcher (one that minimizes Cyclex's runtime).

Our cost model captures the three execution steps of Section VI. We model the elapsed time of each step as a weighted sum of I/O and CPU costs. The weights are measured empirically, allowing us to account for varying execution characteristics across steps, implementations, and platforms. With the weights, we can reasonably capture completion times of highly tuned implementations that overlap I/O with CPU computation (in this case, the dominated cost component will be completely masked and therefore have weight 0) as well as simple implementations that do not exploit parallelism.

Let m be the number of pages in P_{n+1}, m_b be the total size of P_{n+1} on disk (in blocks), and l be the average page size (in bytes). Let n be the number of mentions in the previous mention table M_n, and n_b be the total size of M_n on disk (in blocks). Let b be the number of buckets in the in-memory hash table H (cf. Section VI). We model the completion time of a Cyclex plan on P_{n+1} as:

$$\hat{w}_{1,\mathrm{IO}} \cdot m_b \cdot \hat{f} + \hat{w}_{1,\mathrm{mat}} \cdot m \cdot l \cdot \hat{f} + \hat{w}_{1,\mathrm{ex}} \cdot m \cdot l \cdot \hat{f} \cdot \hat{g} \quad (1)$$

$$+\hat{w}_{2,\mathrm{IO}} \cdot n_b + \hat{w}_{2,\mathrm{find}} \cdot n \cdot \frac{m \cdot \hat{f} \cdot \hat{h}}{b} \quad (2)$$

$$+\hat{w}_{3,\mathrm{IO}} \cdot m_b(1-\hat{f}) + \hat{w}_{3,\mathrm{ex}} \cdot m \cdot l \cdot (1-\hat{f}), \quad (3)$$

where $\hat{f}$ is the fraction of pages in P_{n+1} with a match in P_n; $\hat{g}$ measures, on average, what fraction of the text within a matched page still needs re-extraction; and $\hat{h}$ is the average number of tuples inserted into hash table H per matched page. The $\hat{w}$'s are weights, whose numeric subscripts reflect which phases incur the associated costs.

Line (1) models the completion time of the first execution step. This includes I/O cost of reading in matching pages from P_{n+1} and P_n, CPU cost of matching the pairs of pages to identify copy regions, and the CPU cost of applying E to extraction regions. Line (2) models the second step. This includes I/O cost of reading in M_n, and CPU cost of probing H to determine whether to copy each mention. The term $\frac{m \cdot \hat{f} \cdot \hat{h}}{b}$ estimates the number of hash table entries per bucket. Finally,

Data Sets	DBLife	Wikipedia
# Data Sources	980	925
Time Interval	1 day	21 days
# Snapshots	30	20
Avg # Page per Snapshot	10155	3038
Avg Size per Snapshot	180M	35M

Extractors for DBLife	α	β
researcher (first name, mid name, last name)	32	3
affiliation (researcher name, organization)	93	7
talk (speaker, time, location, topics)	400	10

Extractors for Wikipedia	α	β
actor (first name, mid name, last name)	35	3
play (actor name, movie)	96	4
award (actor name, award, movie, role)	250	10

Fig. 6. Data sets and extractors for our experiments

Line (3) models I/O cost of reading in unmatched pages in P_{n+1}, and CPU cost of applying E to them. In all three steps, we ignore the cost of writing out mentions in P_{n+1}, since this cost is the same for all matcher choices.

As a special case for DN, which simply runs E over the entire P_{n+1}, Lines (1) and (2) are always 0, and $\hat{f} = 0$ on Line (3). For UD and ST, $\hat{f}$ is the same. In general, however, the hatted parameters $\hat{f}$, $\hat{g}$, $\hat{h}$, and $\hat{w}$'s need be estimated, and their values may differ across alternatives. On the other hand, unhatted parameters do not need to be estimated, because their exact values are directly available from either the corpus metadata (for m, m_b, l, n, and n_b) or the execution context (for b).

We estimate the parameters using a small sample S of P_n as well as the past k snapshots, for a pre-specified k. For space reasons, we do not discuss parameter estimation further, but refer the reader to the full paper that is available online [6]. Section VIII demonstrates empirically that small $|S|$ and k are sufficient for our applications of Cyclex, meaning that parameter estimation and cost-based plan selection adds very little overhead to the overall cost.

VIII. Empirical Evaluation

We now empirically evaluate the utility of Cyclex. Fig. 6 describes two real-world data sets and six extractors used in our experiments. DBLife consists of 30 consecutive one-day snapshots from DBLife system [18], and Wikipedia dataset consists of 20 consecutive snapshots obtain from Wikipedia.com. The DBLife extractors extract mentions of academic entities and their relationships, and the three Wikipedia extractors extract mentions of entertainment entities and relationships (see the figure).

We obtained extractor scopes and contexts by analyzing the extractors. For example, "talk" extractor detects speakers, time and topics by matching a set of regular expressions. The length of extraction context for these attribute is 0. Then "talk" detects location attribute by (a) detecting a set of keywords such as "Location: ," "Room: " etc., and (b) extracting 1-2 capitalized words immediately following the detected keyword as the location. We thus set the context β of "talk" to be the maximal length of all keywords.

Runtime Comparison: For each of the above six extraction tasks, Fig. 7 shows the runtime of Cyclex vs. DNplan, STplan, and UDplan, three plans that employ matchers DN, ST, and UD, respectively, over all consecutive snapshots (the X axis).

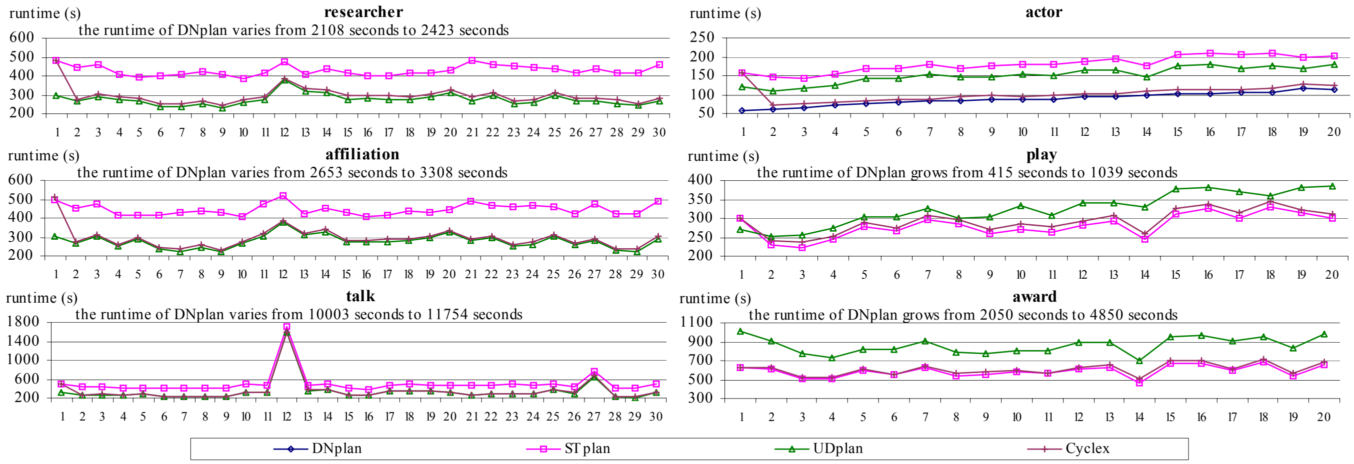

Fig. 7. Runtime of Cyclex versus the three algorithms that use different page matchers

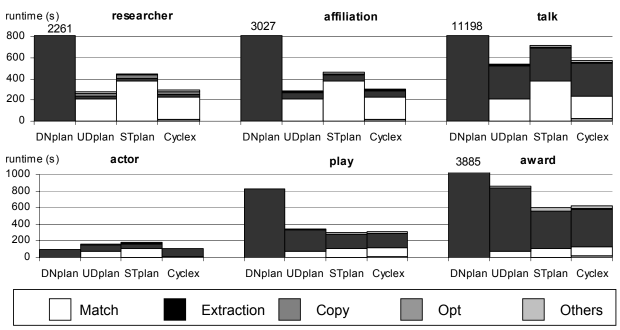

Fig. 8. Runtime decomposition of different plans

The runtimes of DNplan are significantly higher than those of the other three plans. Hence, to clearly show the differences in the runtimes among all plans in one figure, we only plot the curves of STplan, UDplan, and Cyclex, and summarize the trends of the curves of DNplan. Note that for each snapshot, Cyclex employs a cost model to pick and execute the best among the above three plans. Cyclex's runtime includes statistic collection, optimization, and execution times.

The results show that in all cases except "actor," UDplan, STplan, and Cyclex drastically cut runtime of DNplan (which always applies extraction from scratch to the current snapshot), by 50-90%. This suggests that recycling past IE efforts can be highly beneficial.

Next, the results show that none of DNplan, STplan, and UDplan is uniformly better than the others. For example, for "actor," where the changes between two consecutive snapshots are substantial and the extraction cost is fairly low, DNplan outperforms UDplan and STplan. In contrast, for "play" and "award," where the change of data is still substantial but extraction is very expensive, STplan is the winner. For DBLife cases, where the consecutive snapshots change little and matching regions detected by UD and ST are quite similar, UDplan is the winner.

The above results underscore the importance of optimization to select the best plan for a particular extraction situation. They also show that Cyclex handles this optimization well. It successfully picks the fastest plan in all six cases, while incurring only a modest overhead of 4-13% the runtime of the fastest plan.

Contributions of Components: Fig. 8 shows the decomposition of runtime of various plans (numbers in the figure are averaged over five random snapshots per IE task). "Match" is time to match pages, "Extraction" is time to apply IE, "Copy" is time to copy mentions, "Opt" is optimization time of Cyclex, and "Others" is the remaining time (to read file indices, doing scoping, etc.).

The results show that matching and extracting dominate runtimes, hence we should focus on optimizing these components. The suffix-tree matcher ST clearly spends more time finding old regions than the Unix-diff matcher UD. However, the figure shows that this effort clearly pays off in certain cases, such as "play" and "award," where IE is expensive and the consecutive snapshots change substantially. Here, STplan saves significant time avoiding IE. Finally, the results show that the overhead of Cyclex (statistic collection, etc.) remains insignificant compared to the overall runtime.

We also found that DNplan (i.e., applying IE from scratch) incurs very little IO time in most tasks (less than 3% of total runtimes, numbers not shown due to space reasons) Thus, it is important to optimize CPU time, as we do in this work.

Sensitivity Analysis: Finally, we examined the sensitivity of Cyclex wrt the main input parameters: k and $|S|$, the number of snapshots and size of sample used in statistic estimation, and the scope and context values.

Fig. 9.a plots the "accuracy" of Cyclex as a function of k, where "accuracy" is the fraction of snapshots where Cyclex picks the correct (i.e., fastest) plan. We show results for "affiliation" and "play" only, results for other IE tasks show similar phenomenons.

Fig. 9.b-d plots the "accuracy" of Cyclex in a similar fashion against changes in the sample size $|S|$, scope α, and context β, respectively.

The results show that Cyclex only needs a few recent snapshots (3) and a small number of sample size (30 pages) to do well. Regarding scope and context, the results show that for "affiliation," Cyclex performs well even when we increased α and β significantly, by 5 and 100 times, respectively. For "play," Cyclex performs well until α was increased by 4 times. As α increases, the difference between the fastest plan, STplan,

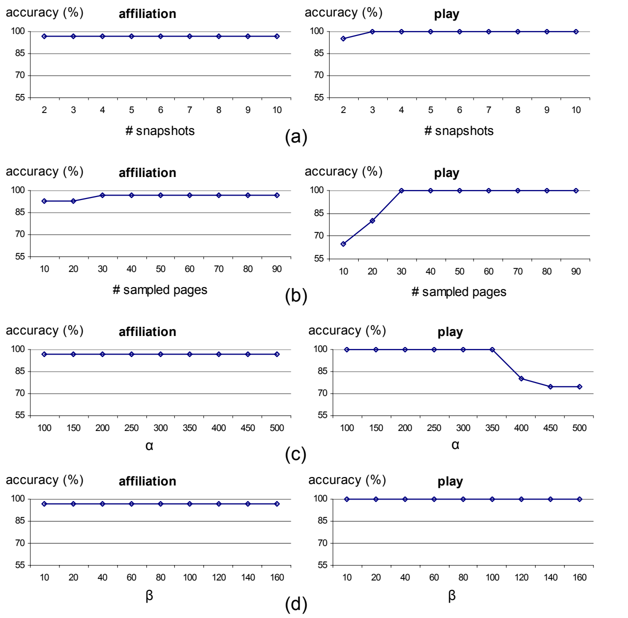

Fig. 9. Accuracy of cost models as a function of (a) number of snapshots k, (b) sample size $|S|$, (c) α, (d) β

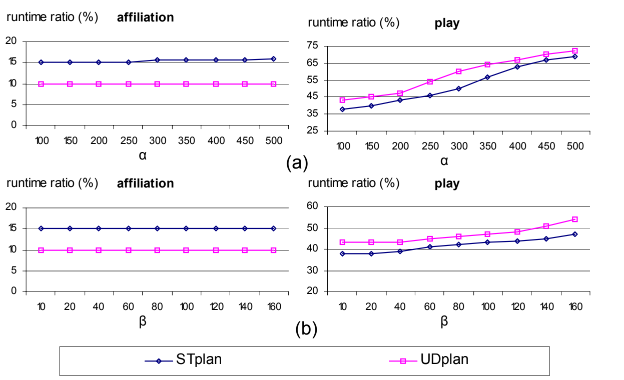

Fig. 10. Ratio of runtimes as a function of α and β

and the second fastest plan, UDplan, becomes smaller and smaller, thus causing the optimizer to mistakenly select the second fastest plan on certain snapshots.

In the final experiment, Fig. 10 shows the runtime ratio of STplan and UDplan as a function of α and β. The runtime ratio is the ratio of the runtime of these plans over the runtime of DNplan. The results show that this ratio changes only slowly, as we increase α and β. This suggests that a rough estimation of α and β does increase the runtime of the various plans, but only in a graceful fashion.

IX. Conclusions & Future Work

A growing number of real-world applications must deal with IE over dynamic text corpora. We have shown that executing such IE in a straightforward manner is very expensive, and have developed Cyclex, an efficient solution that recycles past IE results. As far as we know, Cyclex is the first in-depth solution in this direction. It opens up several interesting research directions that we are planning to pursue. These include (a) how to handle multiple extractors, in these cases it is yet unclear how to extract copy and extraction regions of a page, and (b) how to handle extractors that extract mentions across multiple pages.

Acknowledgment

This work is funded by NSF grant ITR IIS-0326328 to the fourth author and NSF CAREER grant IIS-0347903 and a Sloan fellowship to the second author.

References

[1] A. Chandel, P. C. Nagesh, and S. Sarawagi,"Efficient batch top-k search for dictionary-based entity recognition," in *Proc. ICDE'06*.

[2] A. Doan, R. Ramakrishnan, and S. Vaithyanathan,"Managing information extraction: state of the art and research directions (tutorial)," in *Proc. SIGMOD'06*. Available: http://pages.cs.wisc.edu/∼anhai/papers/ie-tutorial06-final.ppt

[3] A. Jain, A. Doan, and L. Gravano, "SQL queries over unstructured text databases," in *Proc. ICDE'07*.

[4] A. Z. Broder, "Identifying and filtering near-duplicate documents," in *Proc. CPM'00*.

[5] E. Agichtein and S. Sarawagi, "Scalable information extraction and integration (tutorial)," in *Proc. KDD'06*.

[6] F. Chen, A. Doan, J. Yang, and R. Ramakrishnan, "Efficient information extraction over evolving text data," Available: http://www.cs.wisc.edu/∼fchen/cyclex-report.pdf

[7] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan, "An algebraic approach to rule-based information extraction," in *Proc. ICDE'08*.

[8] D. Gusfield, *Algorithms on strings, trees, and sequences: computer science and computational biology*, Cambridge : Cambridge University Press, New York, NY, USA, 1997.

[9] J. Cho, and H. Garcia-Molina, "Effective page refresh policies for web crawlers," *TODS*, 28(4):390–426, 2003.

[10] J. Edwards, K. McCurley, and J. Tomlin, "An adaptive model for optimizing performance of an incremental web crawler," in *Proc.WWW'01*.

[11] J. Zhang and T. Suel, "Efficient search in large textual collections with redundancy," in *Proc. WWW'07*.

[12] L. Lim, M. Wang, S. Padmanabhan, J.S. Vitter and R. Agarwal, "Dynamic maintenance of web indexes using landmarks," in *Proc. WWW'03*.

[13] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman, "Computing iceberg queries efficiently," in *Proc. VLDB'98*.

[14] M. Herscovici, R. Lempel and S. Yogev, "Efficient indexing of versioned document sequences," in *Proc. ECIR'07*.

[15] E. Myers, "An O(ND) difference algorithm and its variations," in *Algorithmica*, 1(1):251–266, 1986.

[16] N. Koudas, S. Sarawagi, and D. Srivastava, "Record linkage: similarity measures and algorithms (tutorial)," in *Proc. SIGMOD'06*.

[17] N. Shivakumar and H. Garcia-Molina, "SCAM: A copy detection mechanism for digital documents," in *Proc. DL'95*.

[18] P. DeRose, W. Shen, F. Chen, Y. Lee, D. Burdick, A. Doan, R. Ramakrishnan, "DBLife: A community information management platform for the database research community (demo)," in *Proc. CIDR'07*.

[19] P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano, "To search or to crawl?: Towards a query optimizer for text-centric tasks," in *Proc. SIGMOD'06*.

[20] R. McCann, B. AlShebli, Q. Le, H. Nguyen, L. Vu, and A. Doan, "Mapping maintenance for data integration systems," in *Proc. VLDB'05*.

[21] B. Bhattacharjee, V. Ercegovac, J. Glider, R. Golding, G. Lohman, V. Markl, H. Pirahesh, J. Rao, R. Rees, F. Reiss, and E. Shekita,"Impliance: A next generation information management," in *Proc. CIDR'07*.

[22] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan, "Declarative information extraction using datalog with embedded extraction predicates," in *Proc. VLDB'07*.

[23] A. W. Cohen and A. McCallum, "Information extraction from the World Wide Web (tutorial)," in *Proc. KDD'03*.

[24] Y. Cai, X. L. Dong, A. Y. Halevy, J. M. Liu, and J. Madhavan, "Personal information management with SEMEX," in *Proc. SIGMOD'05*.