Universität des Saarlandes
Max-Planck-Institut für Informatik
AG5

Efficiently Identifying Interesting Time Points in Text Archives

Masterarbeit im Fach Informatik
Master's Thesis in Computer Science
von / by

Vinay Setty

angefertigt unter der Leitung von / supervised by

Prof. Dr. Gerhard Weikum

betreut von / advised by

Dr. Srikanta Bedathur

Klaus Berberich

begutachtet von / reviewers

Dr. Srikanta Bedathur

Prof. Dr. Gerhard Weikum

Februar / February 2010

Declaration of Authorship

I, Vinay Setty, declare that this thesis titled, 'Efficiently Identifying Interesting Time Points in Text Archives' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master's degree at this University.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed: _____

Date: _____

"Time will explain it all. He is a talker, and needs no questioning before he speaks,"

Euripides, BC 480-406, Greek Tragic Poet

Abstract

Large scale text archives are increasingly becoming available on the Web. Exploring their evolving contents along both text and temporal dimensions enables us to realize their full potential. Standard keyword queries facilitate exploration along the text dimension only. Recently proposed *time-travel keyword queries* enable query processing along both dimensions, but require the user to be aware of the exact time point of interest. This may be impractical if the user does not know the history of the query within the collection or is not familiar with the topic.

In this work, our aim is to *efficiently identify interesting time points* in Web archives with an assumption that we receive a result list for a given query in standard relevance-order from an existing retrieval system. We consider two forms of Web archives: (i) one where documents have a publication time-stamp and never change (such as news archives), and (ii) the archives where documents undergo revisions, and are thus versioned. In both settings, we define interestingness as the change in top-k result set of two consecutive time-points. The key step in our solution is the maintenance of top-k results valid at each time-point of the archive, which can then be used to compute the interestingness scores for the time-points. We propose two techniques to realize efficient identification of interesting time points: (i) For the case when documents once published never change, we have a simple but effective technique. (ii) For the more general case with versioned documents, we develop an extension to the segment tree which makes it rank-aware and dynamic. To further improve efficiency, we propose an early termination technique which is proven to be very effective. Our methods are shown to be effective in efficiently finding interesting time points in a set of experiments using the New York Times news archive and the Wikipedia versioned archive.

Acknowledgements

First and foremost, I would like to thank my advisors Dr. Srikanta Bedathur and Klaus Berberich, for their invaluable guidance. Their views and advise was crucial in fruitful completion of my thesis. The discussions with them were always fruitful and insightful.

A special note of thanks to Prof. Gerhard Weikum for giving me the opportunity to pursue this thesis under his supervision and for his valuable inputs.

I am very thankful to my friends and classmates for their company and moral support. I learnt a lot from them in last 2 years which is a very important factor for successful completion of my thesis.

Finally, I would like to thank IMPRS-CS for the assistance and financial support that I received from them.

Contents

Dedicated to Tobias Tykvart

Chapter 1

Introduction

1.1 Motivation

The World Wide Web is constantly evolving. With proliferation of news sites, blogs, wikipedia and social networks the World Wide Web has become an important source of information and an effective medium of communication. Recently, there have been many efforts to digitize and archive the information and make it searchable on the web. Large corporations such as Google and Microsoft have also made efforts to digitize newspaper articles and periodicals spanning over a long period of time and make them searchable. Apart from digitization of newsworthy events, the netizens document the real world events and their opinions in the form of wikis, blogs and tweets (micro-blogs). Also, web-based encyclopedias like Wikipedia are constantly evolving over time and their revision history is preserved. As a result large document collections of news articles such as the New York Times [1], Wikipedia version history [2] spanning over a large period of time, are becoming increasingly available. These evolving collections have temporal information associated with documents in the form of publication time, version history and temporal references etc. Exploring their evolving contents along both text and temporal dimensions enables us to realize their full potential.

In text search, the user provides a set of keywords indicating a topic of interest to the search engine. In a typical search engine, frequency-based measures (e.g. tf-idf) [3] are used to process the keyword queries. But in a document collection spanning over a large period of time, frequency-based measures like tf-idf do not consider the temporal dimensions of the document collection. If the query is restricted to a specific date, then the queries cannot be answered using simple frequency-based measures. Recently proposed "time-travel keyword queries" by Berberich et al. in [4] enable query processing

in both text and temporal dimensions. The "time-travel keyword queries" are keyword queries along with a temporal context, and they can be used to get relevant documents alive at a particular time point for the topic of interest.

Use case: Suppose a curious voter wants to know what Sarah Palin's [1] political policies were, when she became the Governor of Alaska. If the voter is aware of the fact that she became Governor of Alaska on 4^{th} of December 2006, then queries like "Sarah Palin"@04/12/2006 can be used to retrieve documents related to "Sarah Palin" alive on 04/12/2006. Notice that, this requires prior knowledge about "Sarah Palin". But if the voter is unaware of "Sarah Palin" it is impossible to compose such queries. In that case, the voter will be left with a huge number of time points to explore. There is a need to enhance web archive exploration by assisting the user not only by identifying interesting time points for a given query but also by quickly providing top-k relevant documents at the identified interesting time points.

A majority of the existing work to identify important events from a document collection is inspired by the Kleinberg's work on bursty and hierarchical structure of streams [5]. A significant amount of work in this direction has been done based on burst-detection methods [6, 7]. While these are proven techniques for burst detection, they do not address efficiency aspects to identify important events from a large document collection like New York Times news archive. Works which propose query based burst event detection [8, 9] give little or no emphasis on efficiency. Furthermore, these approaches do not consider changes in top relevant documents to the query.

Use Case: Assume, a certain news article is on top rank for the query "Sarah Palin" until 03/12/2006 and on 04/12/2006 the existing top news article is not available anymore, and some other recent news article replaces its position. In this case, even though there is no significant difference in the frequency of the terms "Sarah" and "Palin", there is a significant change in the top documents relevant to the query. To the best of our knowledge we are the first to address this problem.

[1] Republican nominee for Vice President of the United States in 2008

1.2 Problem Statement

Consider a scenario in which we are given,

1. A document collection over a period of time with temporal information like date of publication or version history etc.
2. A text retrieval system, which can effectively and efficiently answer relevant documents for keyword queries (e.g. Lucene or any search engine like Google).
3. A keyword query.

The goal of this work is to efficiently identify interesting time points for the given query and also provide top-k relevant documents at those time points.

Use Case: Given a news archive like the New York Times Annotated corpus [1] and a query like "Sarah Palin" we propose a system which identifies interesting days relevant to the query "Sarah Palin" and at the same time provide top-k documents alive at time points like 04/12/2006 (assuming 04/12/2006 is an interesting time point identified for the query "Sarah Palin").

1.3 Contributions

We make the following contributions during the course of our work.

1. We define quantitative measures to compute the "interestingness of a time point" for a given query.
2. We propose an efficient interval index structure based on the segment tree [10], which can be used to identify interesting time points. It can also index the top-k relevant documents at all time points, to compute the interestingness of all the time points. The same index structure can be used to process top-k results for "time-travel keyword queries" efficiently.
3. For further improvement in efficiency, we propose an "early termination technique", to identify top-m interesting time points for a given keyword query, based on our interestingness measure.

4. Finally, we propose an end-to-end system prototype which can take a document collection like the New York Times annotated corpus [1] and a keyword query as input and produces a visual representation of identified interesting time points as output.

1.4 Outline

The remainder of the thesis is structured as follows. In Chapter 2, we discuss related work done and some important ideas used in the thesis. In Chapter 3, we explore the existing interval index structures for managing time intervals that support time point queries efficiently and propose a dynamic and rank-aware extension of the segment tree [10]. In Chapter 4, we introduce notion of interestingness and design an early-terminating algorithm to identify the most interesting time points. We also propose an end-to-end system architecture to identify interesting time points. In Chapter 5, we present the experimental setup and results to evaluate our approach. Finally conclusions and future work are presented in Chapter 6.

Chapter 2

Related Work

In this thesis, we focus on temporal aspect of web archives. There is a significant amount of work which exploits the temporal dimension of the evolving document collections. In this chapter we try to analyze them and list their limitations as approaches to explore web archives. In Section 2.1 we discuss how "time-travel keyword queries" can be processed. In Section 2.2 and Section 2.3 we discuss several approaches which are directly relevant to our work as they identify bursty or important events from an evolving document collection. Finally, in Section 2.4 we introduce the conventional top-k algorithms and explain how our early termination algorithm is inspired by them.

2.1 Time-Travel Text Search

Berberich et al. [4] introduced the concept of "time-travel search". They consider text-search over time versioned data collections. This new concept supports "time-travel keyword queries" which answers all relevant documents for a given query existed at a given time point. They facilitate text search based on temporal context by maintaining an index referred to as the Time-Travel Index (TTIX). The TTIX builds on the standard inverted index the workhorse of Information Retrieval (IR). TTIX extends index entries by validity-time intervals. The index entries thus have the following form

$$< did, t_b, t_e, tf >,$$

where *did* is a document identifier, $[t_b, t_e]$ is the validity-time interval and tf is the term frequency. Query processing over TTIX involves traversing these index lists which are typically large. To address this problem, the authors use techniques called temporal coalescing and sublist materialization to improve query processing efficiency. Temporal

coalescing reduces the number of postings by coalescing temporally adjacent version payloads into one unified virtual entry. Thus many temporally adjacent entries which have similar scores are considered as one unified entry over a longer period engulfing the lifetimes of all the participants.

In addition, TTIX partitions the time axis for each term separately, thus yielding multiple sub-lists per term, each responsible for an associated time interval. The index list covering time range from t_i to t_j thus contains all index entries for a single term whose validity-time interval overlaps with $[t_i, t_j]$. Such a partitioning of the time axis introduces extra storage costs, since index entries are replicated across index lists. If their validity time interval overlaps with more than one of their associated time intervals, in [4] Berberich et al. proposed technique called sublist materialization, that determine temporal partitioning of individual inverted lists that trade-off extra storage-costs and query-processing gains. These give rise to two approaches: the Performance Guarantee (PG) approach and Space Bound (SB) approach.

Even though time-travel searching facilitates web archive exploration in temporal domain, the user has to be the subject matter expert to use such a system. But the idea of treating documents as line segments with end points as begin time and end time and using them to explore temporal dimension of the document collections is inspired by this work.

2.2 Burstiness Detection in Text Streams

2.2.1 Kleinberg Model

Kleinberg [5] has discussed burst-detection in the context of text streams. Kleinberg model is based on modeling the stream using an infinite state automaton. This approach is computationally expensive, because it requires computing the minimum cost state sequence that involves the application of a forward dynamic programming algorithm on a Hidden Markov Model [11]. The states of Hidden Markov Model correspond to frequency levels for individual terms. The state transitions (bursts) correspond to points in time around which the frequency of a term changes significantly. Given the frequency sequence of a term, a dynamic programming approach is used to fit the most possible state sequence that is likely to have generated the given frequency sequence. The state assigned to each interval will serve as its burstiness score. This approach is related to

our work because [5] detects bursty events in text streams, which is similar to identifying interesting time points. But it does not fulfill our requirements for following reasons:

1. Dynamic programming makes it computationally expensive.
2. Since Kleinberg model is a frequency-based approach, it is effective for burst-detection. But if we want to identify the change in top relevant documents, this model is not appropriate.

2.2.2 Blogscope

Bansal and Koudas [9] have presented a system called BlogScope (www.blogscope.net) which analyzes blogs. The system crawls and indexes the Blogosphere and extracts information for popularity analysis. BlogScope supports temporal burst detection for keywords, identifies correlated sets of keywords, etc. In this work they model the popularity x of a query as the sum of a base popularity μ and a zero mean Gaussian random variable with variance σ^2.

$$x \sim \mu + N(0, \sigma^2)$$

They compute the exact popularity values $x_1, x_2, ..., x_w$ for the last w days by materialized statistics. Then they estimate the parameters mean (μ) and variance(σ) from this data using the maximum likelihood.

$$\mu = \frac{1}{w}\sum_{i=1}^{w} x_i \text{ and } \sigma^2 = \frac{1}{w}\sum_{i=1}^{w}(x_i - \mu)^2$$

From the standard normal curve, the probability of the popularity for some day being greater than $\mu + 2\sigma$ is less than 5%. They consider such cases as outliers and label them as bursts. Therefore, the i^{th} day will be identified as a burst if the popularity value for the i^{th} day is greater than $\mu + 2\sigma$. Even though this approach is query based, it is not clear how efficient it is for large web archives, because in [9] there is no experiments to test its efficiency and scalability. Also this approach is frequency-based and therefore it is oblivious to the change in top relevant documents. For example, if top-k documents at time point x_i and time point x_j are completely different even though the frequency remains same, this approach ignores such cases.

2.2.3 Burstiness-Aware Search for Document Sequences

Lappas et al. [8] define burstiness based on the concept of discrepancy. The discrepancy concept is generally used to describe the deviation of a situation from the expected behavioral baseline. They directly incorporate the burstiness information in the indexing and ranking of documents. They propose a search framework for documents that considers term burstiness in the indexing and ranking process. Furthermore, they use an extension of the well known TA algorithm [12] and analyze documents with burstiness score on temporal axis to identify top ranked overlapping intervals. But this approach is again based on the deviation in the frequency of the term. To the best of our knowledge, our work is the first that analyzes top documents at every time point of the system to identity interesting time points.

2.3 Content-Based Approaches

The approaches described so far analyze statistics like term frequency, term burstiness, frequency discrepancy, and frequency deviation. In this section we will look at two approaches which focus on event extraction based on the content of the documents.

2.3.1 Temporal Profiles

Jones and Diaz in [13] look at the temporal information provided by each of the top-k documents relevant to the query Q and weigh this information according to the document's probability of relevance, $P(Q|D)$. They look at the first R documents retrieved, as a representative for the set of relevant documents, and weight each by the estimated relevance. Their temporal query model is defined as

$$\tilde{P}\left(t|Q\right) = \sum_{D \in R} \tilde{P}(t|D) \frac{P(Q|D)}{\sum_{D' \in R} P(Q|D')}$$

where, R is the set of top-k documents. The first factor in the summation represents the temporal information about the document. The second factor is the normalized retrieval score. While this is an interesting approach they do not focus on efficiency. They mainly concentrate on identifying the temporal classes and classifying the queries to those classes.

2.3.2 Query-Based Event Extraction Along a Timeline

Chieu et al. [14] extract events relevant to a query from a document collection. Each event is represented by a sentence extracted from the document collection based on the assumption that "important" events are widely cited in many documents for a period of time within which these events are of interest. A timeline of sentences containing the given query is extracted. The interestingness of a sentence s is number of sentences which report the same event as s in the entire corpus. The burstiness of an event is based on the number of reports of an event within and outside a date duration. This query-based approach not only extracts important time points but also the corresponding events in the form of sentences. Even though it is an interesting method, it is not suitable for exploring huge news archives like New York Times because of following reasons:

- It is not efficient because of its content lookup. The original experiments considered only a small corpus.
- A sentence containing the query may not always represent an event.

2.4 Top-k Algorithms

Top-k queries based on ranking elements of multidimensional datasets are a fundamental building block for information retrieval. The best known general purpose algorithm for evaluating top-k queries is Fagin's threshold algorithm (TA) [12]. In Algorithm 2.1 we describe the a variant of TA called NRA (No Random Access) algorithm and discuss how these algorithms inspired our early termination technique.

Algorithm 2.1 Fagin's NRA algorithm

Input: m score sorted lists of objects

Output: top-k items based on the given aggregation function $aggr(q, d)$ and scoring function $s(q, d)$

1: top-k $\leftarrow \varnothing$; candidates $\leftarrow \varnothing$; min-k $\leftarrow 0$;
2: scan all lists $L_i (i = 1..m)$ in parallel:
3: consider item d at position pos_i in L_i;
4: $E(d) = E(d) \cup \{i\}$;
5: $high_i = s_i(q_i, d)$;
6: $worstscore(d) = aggr\{s_\nu(q_\nu, d) | \nu \in E(d)\}$;
7: $bestscore(d) = aggr\{aggr\{s_\nu(q_\nu, d) | \nu \in E(d)\}, aggr\{high_\nu | \nu \notin E(d)\}\}$;
8: **if** $worstscore(d) >$ min-k **then**
9: remove $argmin_{d'}\{worstscore(d') | d' \in$ top-k$\}$ from top-k;
10: add d to top-k
11: min-k $= min\{worstscore(d') | d' \in$ top-k$\}$;
12: **else if** bestscore(d) $>$ min-k **then**
13: $candidates = candidates \cup \{d\}$;
14: **end if**
15: $threshold = max\{bestscore(d') | d' \in candidates\}$;
16: **if** threshold $\leq$ min-k **then**
17: exit;
18: **end if**

The key idea of the NRA algorithm is estimating the upper bound on the score an unseen document can achieve (step 7 of the Algorithm 2.1), and at each iteration of the algorithm, compare the min-k score so far with the estimated upper limit (step 16 of the Algorithm 2.1). If the estimated min-k is more than the estimated upper bound, then we know that we already have k objects with best score. In our work, we use the similar idea to early terminate the identification of interesting time points. If the interestingness measure is dependent on the score of the documents and if the documents are served in the descending order of their score, then we can use the same technique to facilitate the early termination.

Chapter 3

Index Structures for Ranked Intervals

3.1 Introduction

A natural way to represent documents according to their temporal dimension is using line segments. Each line segment parallel to the time axis represents the lifetime of a document. But if we also want to take their relevance score into consideration and order them according to score, then we need another dimension to represent the ranks. So, it is clear that we need to have a time range in one dimension and score in another dimension. In this work, we model the lifetime of documents from a web archive as line segments parallel to the time axis. We need an in-memory index structure for efficiently answering stabbing queries in their score order. Additionally, one of our requirements is to maintain top-k results at every time point in the archive lifetime using the same index structure. This is required to compute the interestingness of every time point in the archive (described in Chapter 4 in detail). And the most important requirement is that when the data is provided incrementally in score order we should be able to do the same operations efficiently. We require this because the ranked documents are incrementally received in score order.

In Section 3.2 we describe the existing interval index structures that support stabbing queries efficiently, but they are neither rank aware and nor support dynamic insert operation and top-k maintenance . We will analyze why these index structures do not meet our requirements and identify index structure which can meet our requirements with minimal changes. Because of the special requirements we have, we need to design a

novel data structure. In Section 3.4 we extend the segment tree to meet our requirements. Before we describe the data structures, we define some notations required.

Definitions

1. $\mathcal{I}$ is the list of all intervals we want to index and each interval contains a begin point and and end point. The intervals in $\mathcal{I}$ are score ordered.

$$\mathcal{I} = \{< b, e, score >\}$$

   where b is the begin point, e is the end point and the score is some weight associated with the interval, which can be used to rank it. For example, consider the case of documents from a web archive, they have a begin time and an end time and the relevance score of the document can be used to rank them.

2. Stabbing query: The stabbing query asks for all intervals that contain a given point and they should be reported in their score order. Given a query point x the stabbing query returns

$$I_s = \{\forall I \in \mathcal{I} \mid I.b \leq x \wedge I.e > x \wedge \forall i = 1, 2..., m\ I_i.score \leq I_{i+1}.score\}$$

   where m is the number of intervals containing the point x.

3. Rank-awareness of a datastructure: The data structure preserves or respects the rank order of the intervals (defined by their scores) when inserted so that results for stabbing query are reported in score order.

4. Top-k results at every time point: A subset of $\mathcal{I}$ which can be used to answer top-k stabbing queries at every time point in the list of intervals $\mathcal{I}$. Let $\mathcal{T} = \bigcup_{I \in \mathcal{I}} \{I.b, I.e\}$, then top-k intervals I_t at all the time points are defined as

$$I_t \subset \mathcal{I}, I = \bigcup_{t \in \mathcal{T}} (top_k(t)),$$

   where $top_k(t)$ is the top-k intervals(based on their score) that contain point t.

3.2 Existing Approaches

There are many data structures available which support interval indexing, but they are far from meeting our requirements. In all cases we assume that we have a list of intervals $\mathcal{I}$ and each interval is a pair of left end point and right end point. The goal of the data

structures presented in this section is to efficiently answer stabbing query over these intervals. Two of the well known data structures which can efficiently process stabbing queries are interval trees and segment trees. In this section we briefly describe the key ideas behind them. These approaches do no support dynamic insert operation. They require all the intervals to be known in advance.

3.2.1 Interval Trees

Construction Before we can start building interval tree, we must sort all the intervals according to their begin time. An interval tree over all the intervals in $\mathcal{I}$ is constructed in a recursive manner as follows. We begin by picking the mid-point x_{mid} [1] of the interval collection, and let $\mathcal{I}_{center}$ denote the subset of intervals in our collection that are stabbed by x_{mid}. Also, let $\mathcal{I}_{left}$ and $\mathcal{I}_{right}$ be the subset of intervals completely to the left and right of the mid-point x_{mid}. The intervals stabbed by $\mathcal{I}_{left}$ and $\mathcal{I}_{right}$ are then recursively divided in the same manner until no intervals are left. The intervals in node with key $\mathcal{I}_{center}$ are stored in a separate data structure linked to the node in the interval tree. This data structure consists of two lists, one containing all the intervals sorted by their begin points, and another containing all the intervals sorted by their end points. In the end we get a binary search tree with each node containing a mid-point x_{mid}, pointers left and right children which contain all the intervals completely to the left of x_{mid} or completely to the right of x_{mid}. And all intervals overlapping the x_{mid} sorted by begin point and end point are in separate lists.

Example for Interval Tree Figure 3.1 shows a set of ten intervals (labeled as a tuple <identifier, begin point, end point>) in interval 0 to 100 with mid-point 50. This partitions the intervals into the set of intervals stabbed by 50, {1, 3, 5, 6, 10}, the set of intervals completely to the left of 50, {8, 9}, and the set of intervals to the right of 50, {2, 4, 7}. Then we recursively proceed and find the mid-point of the intervals {8, 9} and {2, 4, 7} which are 25 (mid-point of interval from 5 to 45), and 75 (mid-point of interval from 65 to 85). Now we stop because we have no intervals which are not stabbed by points 50, 25 and 75. And in the end we can expect a tree with the root node with key 50 and containing intervals {1, 3, 5, 6, 10}. To its left there is a node with key 25 and contains {8, 9}. And to the right of 50 there is another node with key 75 and containing intervals {2, 4, 7}.

[1] it can also be the center point of the interval covered but in practice median end point is chosen to keep the tree relatively balanced

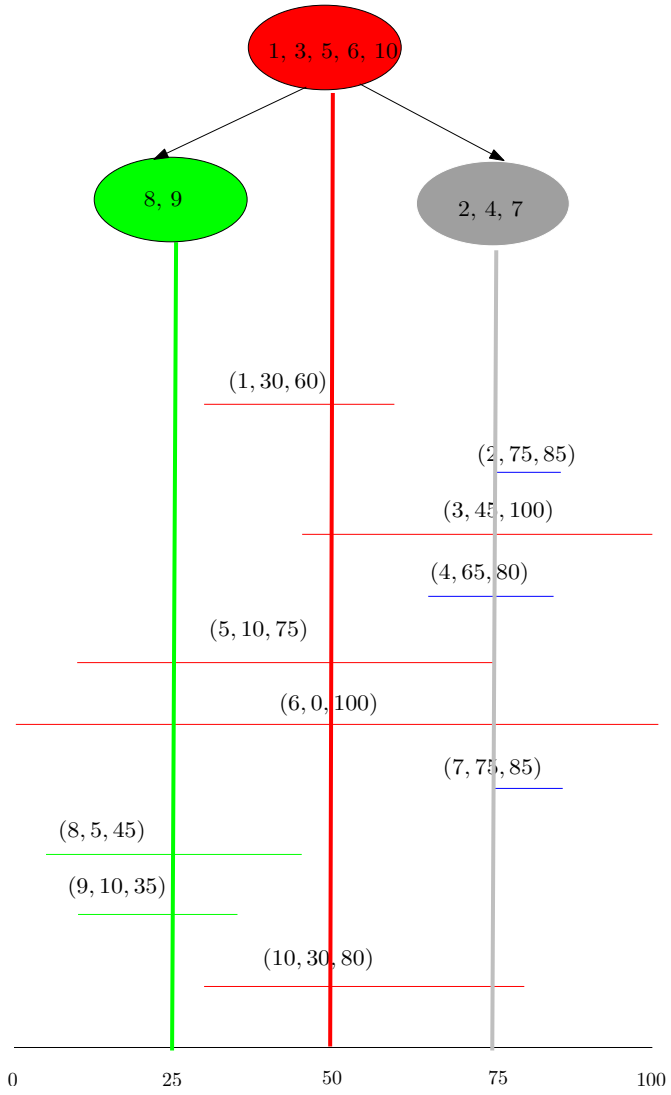


Figure 3.1: Example of Interval Tree

Stabbing Query The task is to find all intervals in the tree that contain a given point x. The tree is traversed recursively in a binary search tree fashion but at each node we check the intervals intersected by the mid-point. At each tree node N, x is compared to $N.x_{mid}$, the mid-point of the interval. If x is less than $N.x_{mid}$ then, the leftmost set of intervals, $N.\mathcal{I}_{left}$, is considered. If x is greater than $N.x_{mid}$ then, the rightmost set of intervals, $N.\mathcal{I}_{right}$, is considered. As we traverse the tree from the root to a leaf, the ranges in its $N.\mathcal{I}_{center}$ are processed. If x is less than $N.x_{mid}$, we know that all intervals in $N.\mathcal{I}_{center}$ end after x. Therefore, we need only to find those intervals in $N.x_{mid}$ that begin before x. We can scan the lists of $N.\mathcal{I}_{center}$ which are sorted by begin and end point. Now since only begin points of the intervals matter, we compare the list which is sorted according to the begin point of the intervals and select all those intervals with begin point less than x. It is easy to see that all these intervals must overlap x because,

they begin before x and end after x. Similarly we can process the end points and answer the intervals overlapping with x, if x is greater than $N.x_{mid}$. For example suppose we want to process a stabbing query with x = 60, we start with the root with mid-point value 50 and we see that its right node has key 75, so 60 must be enclosed in root node. And since 60 lies to the right of 50 we can expect that all intervals stored at this node begin before 50. We then just look for intervals which are ending on or after 60. which are in {1, 3, 5, 6, 10}.

Analysis

- One of the advantages of the interval tree is that its space complexity is linear in the number of intervals i.e. $O(n)$ where n is the number of intervals indexed.
- The runtime complexity of construction of the tree is $O(n \log n)$, because we need $O(n \log n)$ to sort the intervals according to their begin point and we need another $O(n)$ to recursively partition them into center, left and right parts. So effectively it is $O(n \log n)$, but it in practice it is $O(n \log n + n)$.
- Stabbing query takes $O(\log n + m)$ where n is the number of intervals and m is the number of results being reported.

Since interval trees support one dimensional ranges, they are oblivious to the ranks of the intervals being indexed. Later in Section 3.3.1 we will see that they can be made rank-aware. Notice that each node of the interval tree spans an interval, and these interval are not disjoint. The overlapping intervals of the nodes, make it difficult to maintain top-k at each time point, which is one of our requirements.

3.2.2 Segment Trees

Construction: Segment tree [10] partitions intervals into a collection of disjoint and atomic segments and then indexes these segments using a binary-tree structure. The atomic segments are simply derived by sorting the collection of all end points(begin point and end point) in the input set of intervals. If there are n intervals with $2n$ distinct end points then we can have at most $2n + 1$ atomic segments. The segment tree over these sequence of atomic segments is a balanced binary tree. Each node N of the segment tree can be described by a single extent interval $interval(N)$ which is the union of all atomic segments under N's subtree. Each node N also has a unique key which is one of the endpoints of the intervals in $\mathcal{I}$. Each $interval(N)$ contains a time interval spanned between lower bound lb and upper bound ub. Let a node N of segment tree contains all

intervals $I \in \mathcal{I}$ such that $interval(N) \subseteq \mathcal{I}$ but $interval(N.Parent) \not\subseteq \mathcal{I}$. In other words each node N of the segment tree contains intervals which are covered by its extent interval $interval(N)$ and does not contain anything which is covered by $(interval(N.Parent))$. The stabbing query processing in segment tree is greatly simplified by the fact that the atomic interval partitions the underlying domain. This implies that at each level of the tree from root to leaf only one extent interval will be stabbed by the query. So query processing is simply moving down the tree from root to leaf following the nodes of the tree whose extent interval is stabbed by the query.

Example for Segment Tree Figure 3.2 shows a set of 10 intervals, each of which are again represented with tuple tuple <identifier, begin point, end point>. It is easy to see that they create 8 atomic segments with 9 unique end points <0, 10, 30, 45, 60, 75, 85, 90, 100>. Once we define the atomic intervals, we build a binary tree on these intervals. Then we insert the intervals one after another, and the intervals are copied at different nodes maintaining the invariant $interval(N) \subseteq \mathcal{I}$ but $interval(N.Parent) \not\subseteq \mathcal{I}$. For example consider interval 3 which spans 45 to 100. It is copied at a leaf node whose interval spans from 45 to 60, because none of its parents overlap any part of interval completely. Also the interval 3 is copied at non leaf node whose interval spans from 60 to 100 because it completely overlaps with the part of the interval 3 and hence it is not copied at any of its child nodes. The structure of the segment tree after inserting all intervals is shown in Figure 3.2. Now suppose we want to run a stabbing query at point 65, then we start with the root node with $interval(N) = (0, 100)$. Since it overlaps with the query 65, all the intervals in this node qualify as results to the query. Now we inspect the $interval(N.left)$ and $interval(N.right)$. Then we choose the one which overlaps with the query, i.e. $N.right$. We recursively do this until we reach a leaf node. And at the end of it we have the intervals which intersect point 65. This makes the complexity of stabbing query $O(\log n + m)$, where m is the number of intervals stabbed by query point.

Analysis

- The construction of the tree still costs $O(n \log n)$, as we need $O(n \log n)$ to sort the intervals according to their begin point to define atomic segments. We need another $O(n \log n)$ to copy them in the nodes of the segment tree with invariant $interval(N) \subseteq \mathcal{I}$ but $interval(N.Parent) \not\subseteq \mathcal{I}$. So effectively it is $O(2 \cdot n \cdot \log n)$ but asymptotically it is still $O(n \log n)$.
- Similar to interval trees, stabbing queries can be answered in $O(\log n + m)$, where m is the number of results for the stabbing query.

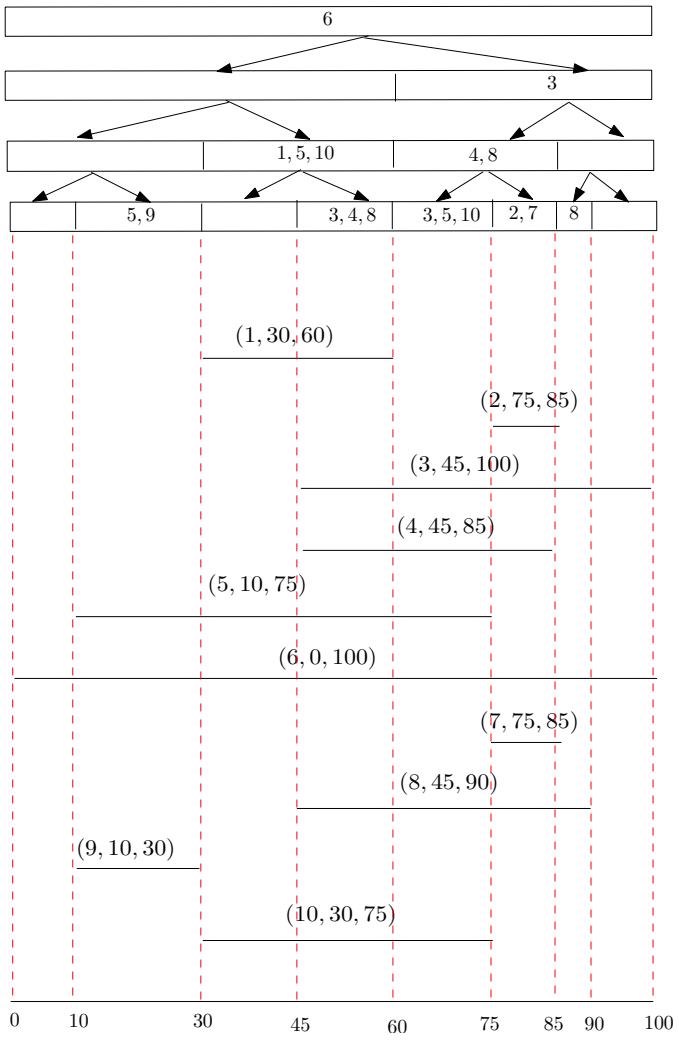


Figure 3.2: Example of Segment Tree

- However, the segment tree exhibits worse space complexity of $O(n \log n)$, because we split the intervals when inserted. In the worst case each interval could be copied at $\log n$ nodes making its space complexity $O(n \log n)$.

Even though the data structure is expensive in terms of space usage when compared to interval trees, the main advantage of segment tree is the property that intervals covered by the nodes are disjoint. This property helps in making segment tree support dynamic insertions. Later, in Section 3.4, we will show how the segment tree can be extended to support dynamic insertions.

3.3 Rank-Aware Interval Index Structures

All the data structures described in Section 3.2 support stabbing query, but none of them are rank aware, that is, they are oblivious to the rank of the documents. In this section we explore the extensions which can be used to turn the data structures see in Section 3.2 to support rank-awareness.

3.3.1 Rank-Aware Interval Trees

One way to make interval trees rank aware is to sort the intervals stored at each node according to their score, instead of their begin and end points. However, this results in complete scan of the lists stored at the node. Alternatively, we can keep track of all the results returned for a query and sort them in the end. For more details refer to [15]. But what we really need is top-k results at every time point in the intervals $\mathcal{I}$. This is not so trivial. Each node in an interval tree has an extent interval, it is enough if we can make sure that each extent interval has at least k results. But since there are overlaps in extent interval we end up accumulating more than k results at some points. As a result we end up storing more than what we need in the tree. But again the major disadvantage is that interval trees are not dynamic in nature.

3.3.2 Rank-Aware Segment Trees

Unlike the interval tree, segment trees have atomic segments which are non-overlapping intervals. This feature of segment trees not only makes stabbing query efficient but also facilitates rank awareness. Because of the non overlapping nature of atomic segments, all internal nodes also have non overlapping intervals. This means all the intervals stored at a node overlapping with stabbing query point are relevant to the query. So all we need to do is to keep the intervals sorted according to their score at each node. And when a stabbing query comes, we can just aggregate the results merging the sorted lists at $\log n$ nodes (from root to leaf).

Theorem 1. top-k stabbing query of rank-aware of segment tree costs $O(\log n + \log n \cdot \log \log k)$

Proof. The tree is traversed from root to the leaf which costs $O(\log n)$ and we maintain a global priority queue to accumulate the results read from each node. Each of the visited $\log n$ nodes we do $\log n$ insertions. Each insertion costs $O(\log \log n)$ but if we restrict to store only top-k results at each node we just need $O(\log \log k)$. So in total we need $O(\log n + \log n \cdot \log \log k)$ for a stabbing query. □

Apart from being rank aware we also want to make segment tree to be able to maintain top-k results at each time point. We know that when all atomic intervals have at least top-k results then we have top-k results at all the time points. This maintenance can be done when we insert the intervals in the segment tree. Whenever top-k results for an interval have been completely determined, we can propagate the check to see if its parent also has top-k results. In this way in the best case when all intervals have top-k results we mark root also as having top-k results and stop inserting more intervals to the tree. Note that rank-aware segment tree is still not dynamic. In Section 3.4 we will show how we can turn rank-aware segment tree into a dynamic data structure that supports insertions.

3.4 Rank-Aware Dynamic Segment Trees

3.4.1 Approach

Now with an extension to make segment tree rank aware, we are left with making it dynamic. With support for insert operation rank-aware segment tree meets all our requirements. Berg et al. in [10] give an idea that we can make the segment tree dynamic by replacing binary tree which is used to index the atomic intervals by a self balancing binary tree like red-black tree or AVL tree. The idea is to keep the binary tree balanced as we insert new intervals by rotation. But we have to be careful when we rotate because rotations cause the changes in intervals covered by the nodes. We should rearrange the intervals in the nodes so that the intervals of the nodes is respected. In this section we propose a method using which we can make the rank-aware segment tree dynamic while still maintaining the performance guarantees of the conventional segment tree.

Dynamization of Segment Tree Making segment tree dynamic to support insertion and still maintaining the benefits of the traditional segment tree is quite simple. To make the atomic segments $I \in \mathcal{I}$ (defined earlier) searchable in logarithmic time we can use any balanced binary search tree like AVL [16] tree or red-black tree [16]. But red-black trees are believed to be faster compared to AVL trees by a constant factor [16]. So we choose red-black tree for indexing atomic segments. And then to store the intervals (in the nodes of the tree) we need a min-heap at each node v of the red-black tree. As we insert new intervals it will create new atomic segments and eventually the tree could become out of balance. We can simply use red-black tree rotations to re-balance the tree to retain its asymptotic query complexity. The algorithms to insert an interval in a rank-aware segment tree with rebalancing using rotations are given in Algorithm 3.1 and Algorithm 3.3

3.4.2 Insert Operation

Algorithm 3.1 Algorithm to insert an interval in rank-aware dynamic segment tree

Input: Set of score sorted intervals to be inserted in segment tree

Output: Top-k results at each time point

```
1: while There are more intervals to insert or top-k results are not present at all time
   points retrieved do
2:    get the next top interval s to be inserted
3:    find least common ancestor LCA node for s
4:    if LCA is a leaf node then
5:       insert the end points of s at LCA
6:       if tree goes out of balance then
7:          RebalanceSegmentTree(s)
8:       end if
9:    else
10:      CopySegment(s, LCA)
11:      insert the end points of s at leaf nodes at right and left sub-tree of LCA
12:      if tree goes out of balance then
13:         RebalanceSegmentTree(s)
14:      end if
15:   end if
16: end while
```

Algorithm 3.2 CopySegment(linesegment, LCA)

Input: Least Common Ancestor(LCA) to the interval being inserted

Output: Copy the interval in all valid atomic segments

```
1: if interval(LCA).lb ≥ linesegment.b and interval(LCA).ub <= linesegment.e then
2:    LCA.add(linesegment)
3:    if priority-queue.contains(LCA) == false then
4:       add LCA to priority-queue
5:    end if
6:    if interval(LCA) has k results then
7:       propagate it to parent recursively
8:       if  interval(parent) of all nodes have k results then
9:          return
10:      end if
11:   end if
12: else if interval(LCA).left.lb >= linesegment.b and interval(LCA).left.ub <= line-
    segment.e  then
13:   CopySegment(linesegment, LCA.left)
14: else if interval(LCA).right.lb >= linesegment.b and interval(LCA).right.ub <=
    linesegment.e  then
15:   CopySegment(linesegment, LCA.right)
16: end if
```

3.4.3 Rotations

Rotations for rank-aware dynamic segment tree involve changing colors and node pointers which are same as that in the original red-black tree and is thus straightforward. However, we have an extra overhead to rearrange the intervals from the nodes which are being rotated to respect the intervals.

Algorithm 3.3 RebalanceSegmentTree(Node)

Input: Unbalanced segment tree

Output: Balanced segment tree

```
1: if Node.color == RED and Node.Parent.color == RED then
2:    if Node < Node.Parent and Node < Node.GrandParent then
3:       Rotate(Node, Node.GrandParent)
4:    end if
5:    Rotate(Node, Node.GreatGrandParent)
6:    Node.color = BLACK
7: end if
```

Algorithm 3.4 Rotate(Node)

Input: Node for rotation

Output: Tree after rotation

```
1: if Node < Node.parent then
2:    if Node < Node.Parent.left then
3:       rotate left child of Node with left child of parent
4:    else
5:       rotate right child of Node with left child of parent
6:    end if
7: else if Node < Node.Parent.right then
8:    rotate left child of Node with right child of parent
9: else
10:   rotate right child of Node with right child of parent
11: end if
```

The rotations are required to keep the rank aware dynamic segment tree balanced. It is easy to see that the rotations spoil the properties of segment tree. To restore the properties of segment tree, we need to rearrange intervals. The process of rearranging intervals is shown in Figure 3.3. Algorithm 3.5 gives the steps required to do right rotation. Left rotation is symmetric and hence we will not show it here. Consider a sub-tree of segment tree as shown in Figure 3.3. In the left part of the picture we see the sub-tree rooted at node b. When we apply the right rotation on this sub-tree we will get a new root a' which was node a before rotation. And when we do right rotation, the $interval(a')$ will be from $interval(b)$ and $interval(b')$ will be $interval(x').lb$ to $interval(y').ub$ where lb and ub are lower and upper bound of the intervals of y' and x' respectively. And it is important to note that $interval(x')$, $interval(y')$, $interval(z')$ remain same as $interval(x)$, $interval(y)$, $interval(z)$. Another important point to note is that the rotations are local to the nodes being rotated. So we have constant number of rotations at each insert [16].

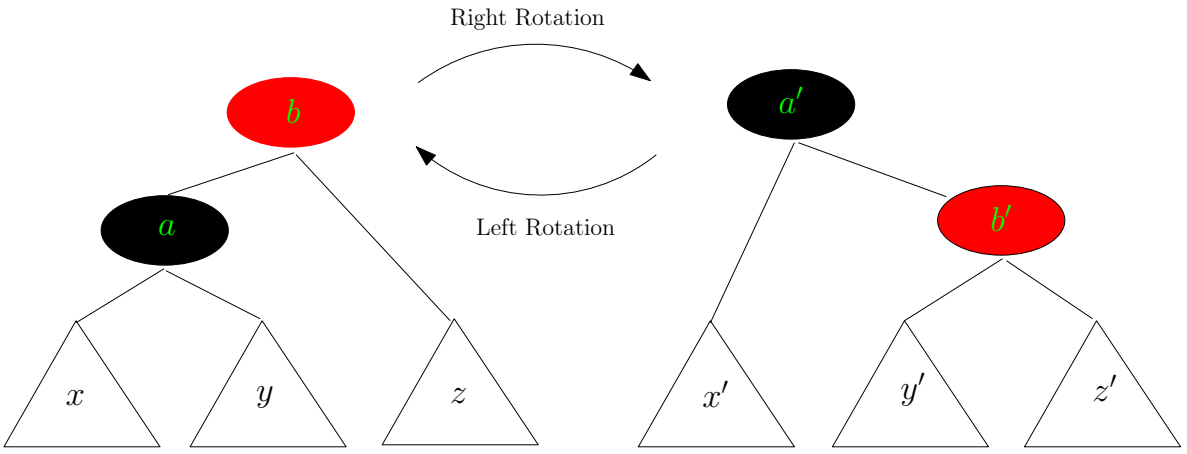


Figure 3.3: Right rotation and left rotations in segment tree

Algorithm 3.5 RearrangeSegmentsForRightRotation

Input: Nodes a and b for rotation as shown in figure 2.

Output: Nodes a' and b' after rearrangement of intervals

1: $a' = b$
2: $temp = y \cap z$
3: $b' = temp$
4: $z' = x - temp$
5: $y' = y - temp$
6: $y' = y' \cup z'$
7: $x' = x \cup b$

3.4.4 Stabbing Query

The stabbing query in rank-aware dynamic segment tree is similar to that in traditional segment tree. The only change needed is that we want to preserve the rank order of the intervals when reporting the result. The Algorithm 3.6 answers the results for a given stabbing query.

Algorithm 3.6 Stabbing query for rank-aware dynamic segment tree

Input: Root node of rank-aware dynamic segment tree and stabbing query point q

Output: List of intervals overlapping query point q in their score order

```
1: result ← ∅
2: node ← root
3: while node is not leaf do
4:    result ← result ∪ node.segments
5:    if q < node.key then
6:       node ← node.left
7:    else
8:       node ← node.right
9:    end if
10: end while
```

3.4.5 Asymptotic Analysis

We now summarize the complexity analysis of rank-aware dynamic segment tree.

- The space complexity still remains $O(n \log n)$
- The cost of building the tree now involves insert operation (Algorithm 3.1) and rotations (Algorithm 3.3). The insert algorithm first finds the LCA which costs $O(\log n)$ in the worst case. Once the LCA is found, if it is leaf node the intervals are inserted with a constant cost of rotation R. If the LCA is not a leaf, then we copy the interval in left and right sub-tree of LCA in the nodes where the interval being inserted is completely overlapping with the interval of node. This is asymptotically $O(\log n)$. But we have an extra overhead to keep track of top-k at every time point. This is done in $O((\log n)^2)$. So the total cost to build rank-aware dynamic interval tree is $O(n \cdot (R + (\log n)^2))$.
- The run time complexity of stabbing query is still $O(\log n + m)$ where m is the number of results being reported.

Chapter 4

Efficient Identification of Interesting Time Points

4.1 Introduction

Given a keyword query, how can we analyze the query and efficiently identify the most interesting time points related to the query? Before we can answer this question, we must define what makes a time point to qualify as interesting for the given query? And how can we measure the interestingness of a time point relevant to the given keyword query. In this work, we analyze temporal information (publication time or version history) from the relevant documents retrieved from web archive. A certain time point in the lifetime of the archive is interesting if the data collection underwent a significant change at that time point. In news archives, these time points often would correspond to real world events. Our approach to quantify the change at any time point is to measure the change in top-k relevant documents of its neighboring time point. The goal of our work is to efficiently identify the most interesting time points relevant to the given query in the entire lifetime of the web archive.

The main challenge in realizing efficient identification of interesting time points is to process the large result set for the query. Since we analyze the lifetimes of the documents that are relevant to the given keyword query, in a large web archive like New York Times news archive [1] or wikipedia version history [2], there could be millions of documents which qualify for the query. It is very expensive to analyze all relevant documents. To identify the most interesting time points quickly, we should access as few documents as possible, but at the same time we should make sure that we do not miss any interesting time points. The assumption that we receive the result documents in their score order

can be leveraged to facilitate early termination in reading the relevant documents for the query. We design NRA (No Random Access) based early termination algorithm. This algorithm can facilitate early termination if the interestingness function is based on the relevance score or rank of the documents. We apply this algorithm to different data models which we will define later in the chapter. Depending on the data model, there are two approaches and they will be described in Section 4.6 and Section 4.7. For generic data model we make use of the rank-aware dynamic segment tree proposed in Chapter 3 that makes sure we have top-k results for all the time points and at the same time identify most interesting time points in an efficient manner. For one of the special data models we propose a simple but effective algorithm, which measures and identifies interesting time points efficiently.

In Section 4.2 we formally define our data model. Then we formally define interestingness of a time point based on change in top-k at each time point and also discuss several variants of interestingness functions, and the rationale behind them in Section 4.4. Finally, we zero in on an interestingness function based on ranks of the relevant documents. In Section 4.5, Section 4.7, and Section 4.6. we present different approaches for all the data models in question. Finally, in Section 4.8, we present a working prototype for the end-to-end system to identify interesting time points.

4.2 Data Model

The document model $\mathcal{D}$ contains each document with a begin time and an end time, spanning the lifetime of the document. The line segment spanning between the begin time point and the end time point represents the document creation, its life time, and its deletion. We have two special cases of data model (1) $\mathcal{D}'$ in which the end time of each document is a constant '*now*', where '*now*'is the present day, (2) $\mathcal{D}''$ The life time of the document is constant, that is end time = begin time + τ where τ is a constant greater than 0.

4.2.1 Generic Data Model: $\mathcal{D}$

If we consider Wikipedia-like archives, each article/document evolves over time. Each of these edits forms a version. In this work we treat each of these versions as a different document with a validity time associated with it. So each document in the collection has two timestamps, begin time and end time which indicate the lifespan of the version respectively. Each document $d \in \mathcal{D}$ is a tuple $< did, t_b, t_e >$ where did is the document

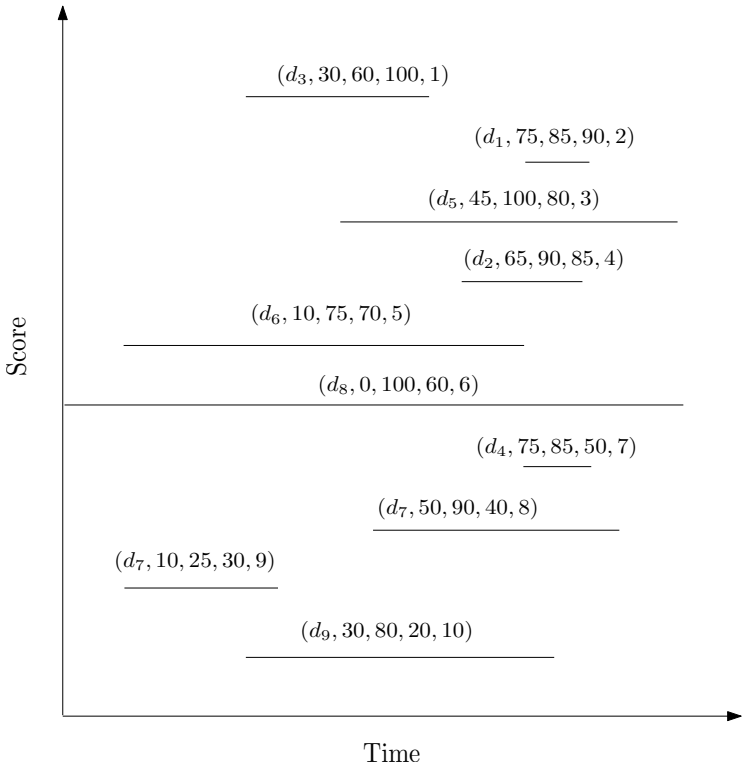


Figure 4.1: Graphical visualization of documents from a generic data model $\mathcal{D}$ retrieved as a result list for a query given by user

identifier, t_b and t_e are the timestamps of the begin and end time of the version. Each document has a relevance score and a rank with respect to a query. A document d is then $< did, t_b, t_e, s_q, r_q >$ where s_q is the relevance score (e.g., determined using Okapi BM25[3]) and r_q is the rank of the document d relative to all the relevant documents for the given query.. It can be graphically visualized as shown in Figure 4.1, For example, the line segment with tuple $< d_6, 10, 75, 70, 5 >$ represents a document with document id d_6, begin time 10, end time 75, relevance score of 70 and its relative rank is 5. Also note that there are 2 line segments with same document id d_7. These represent the different versions of the same document, so they have different begin and end times and also different relevance score and rank.

4.2.2 Special Case I: $\mathcal{D}'$

In a document collection like news archive collection once an article is created it never changes. That is, each article/document is only published once and it is never changed or deleted after that. So each document has a timestamp called begin time (usually publication time of the document) associated with it. Each document $d \in \mathcal{D}'$ is a tuple (d_{id}, t) where did is the document identifier, t is the timestamp or the begin time of the

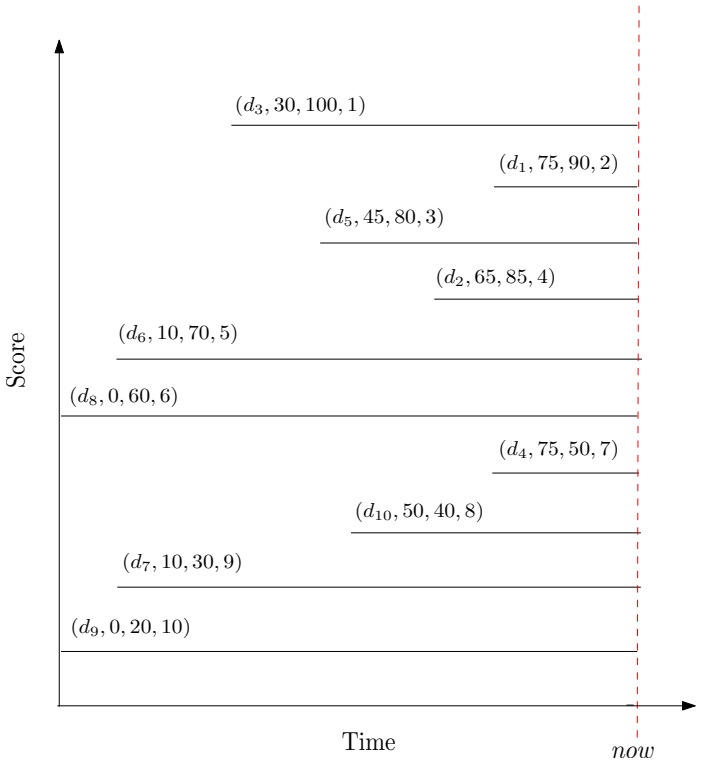


Figure 4.2: Graphical visualization of special data model $\mathcal{D}'$

document. When a document d is referenced in the context of a query, then it will also have corresponding relevance score and the rank associated with it. Given a query q, each document $d =< did, t, s_q, r_q >$ is a tuple of document id, begin time, relevance score and rank. Note as a difference to previous model, the end time is always a constant so it is not part of the tuple. It can be graphically visualized as shown in Figure 4.2, the x-axis in Figure 4.2 represents the time and y-axis repsresents the relevance score. Then each document in the archive is a line segment with its y-coordinates determined by its relevance score. The begin time t_d makes left x-coordinate and right x-coordinate is simply the current time. For example, the line segment with tuple $< d_6, 10, 70, 5 >$ represents a document with document id d_6, begin time 10, relevance score of 70 and its relative rank is 5.

4.2.3 Special Case II: $\mathcal{D}''$

This data model is a special case of the generic data model $\mathcal{D}$ in which the life time of each document is a constant. Formally, we have each document $d \in \mathcal{D}''$ $d =< did, t_b, (t_b + \tau) >$ where did is document id, t_b is begin time and constant $\tau > 0$. When a query is processed in this data model, the documents in the result set are represented by tuple $d =< did, t_b, (t_b + \tau), s_q, r_q >$ where s_q and r_q are respectively relevance score and rank

for the query q. An example for this special data model is the case in which New York Times articles were freely accessible only for a limited time after their publication.

4.3 Problem Definition

We are given a user query q and the corresponding list of query results R. The list of query results R consists of tuples $d \in \mathcal{D}$ [1]. The set of query results R can either be known a priori, or, can be obtained in an incremental fashion (e.g., as the output of a rank join operator). In both cases we assume that R is sorted in descending order of the $d.s_q$ component of the document tuple. In addition, we may be given, as an explicit input, a set of time points $\mathcal{T} = \{t_1, ..., t_n\}$ to which we can limit our attention. If this is not the case, $\mathcal{T}$ consists of all time points occurring as boundaries of the valid-time intervals in R, i.e., it is implicitly defined as

$$\mathcal{T} = \bigcup_{d \in \mathcal{D}} \{d.t_b, d.t_e\}.$$

For all $t \in \mathcal{T}$, we define the top-k results (where k is a parameter) as the set of results alive at t having the highest k scores. We denote set of top-k results at time point t_i as $top_k(t_i)$.

The goal is to read the set of query results R for the given query q and compute $\forall t \in \mathcal{T}$, $top_k(t)$ and at the same time determine the top-m time points with highest interestingness value represented as $top_m(q)$.

$$top_m(q) = \{\{t_1, t_2, ...t_m\} \mid t_i \in \mathcal{T} \wedge \delta(t_i) \geq \delta(t_m)\},$$

where $\delta(t_m)$ is the interestingness of the m^{th} highest interesting time point in $\mathcal{T}$. And the most important requirement is to do all this efficiently.

4.4 Interestingness Model

We consider a time point $t_i \in \mathcal{T}$ interesting if its top-k results differ significantly from the top-k result of its predecessor $t_{i-1} \in \mathcal{T}$. There are several measures that we can use to quantify the degree of difference in top-k results of a time point from its predecessor. In this section, we will present several top-k based approaches and also one approach

[1] Ideally it can be also $d \in \mathcal{D}'$ and $d \in \mathcal{D}'$ for simplicity we just use generic data model

which does looks at cardinality of relevant results at any time point instead of the change in top-k.

4.4.1 Frequency-Based Approach

One simple measure is to count the number of documents alive at any given time point. This approach is in contrast to looking at top-k results at every time point. Higher the number of relevant documents that are alive at that time point the more interesting the time point for the query is. Interestingness function is defined as below

For generic data model $\mathcal{D}$ archive,

$$\delta_f(t_i) = |\{d \in \mathcal{D} : d.t_b \leq t_i \wedge d.t_e \geq t_i\}|.$$

For special data model $\mathcal{D}'$ archive,

$$\delta_f(t_i) = |\{d \in \mathcal{D} : d.t \leq t_i\}|.$$

Even though this approach is very simple, it has the following disadvantages

- In order to get the frequency distribution at all time points, we need to read all relevant documents for the given query. For example, at a certain time point t there might be hundreds of documents alive but are at the bottom of the score sorted result list. Unless we read all the relevant documents for the given query we cannot identify t as an interesting time point.
- If at some time point a large number of low scoring documents are present, then it makes the time point interesting.

These problems can be resloved by analyzing change in top-k relevant documents at every time point instead of all the relevant documents.

4.4.2 Interestingness Based on Top-k Set Similarity

- **Jaccard Similarity**[17] One way to quantify interestingness is by measuring the change in top-k set at any time point. It is quite natural to use set similarity techniques [1] like Jaccard similarity of the top-k result sets at two consecutive time

[1] We can also use comparison of top-k lists based on Kendall's tau, footrule distance etc. [18] to quantify the change in top-k

points t_i and t_{i-1}. Interestingness measure for a time point t_i is defined as below

$$\delta_j(t_i) = 1 - \frac{|top_k(t_i) \cap top_k(t_{i-1})|}{|top_k(t_i) \cup top_k(t_{i-1})|},$$

where $d.score$ is the relevance score of document to query q and $top_k(t)$ is set of top-k relevant results for the given query at time point t.

Intuitively, the more new documents are seen in top-k at any time point the more interesting it becomes for the give query.

- **Weighted Jaccard Similarity**[19] Weighted Jaccard similarity is an extension of Jaccard similarity which can be used to measure the difference in top-k result sets of two consecutive time points. It is defined as below.

$$\delta_w(t_i) = 1 - \frac{\sum_{d \in \{top_k(t_i) \cap top_k(t_{i-1})\}} d.score}{\sum_{d \in \{top_k(t_i) \cup top_k(t_{i-1})\}} d.score}$$

where $\delta_w(t_i)$ is maximum i.e 1.0 when $top_k(t_i) \cap top_k(t_{i-1})$ is $\varnothing$ and it is minimum i.e 0 when $top_k(t_i) \cap top_k(t_{i-1}) = top_k(t_i)$. And the higher the new high scoring documents appear in $top_k(t_i)$ when compared with $top_k(t_{i-1})$ the more the weighted difference $\delta_w(t_i)$

4.4.3 Interestingness Based on Sum of Document Weights

We define the interestingness of a time point t_i as

$$\delta_r(t_i) = \sum_{d \in \{top_k(t_i) - top_k(t_{i-1})\}} w(d),$$

where $w(d)$ can be any weight based on the relevance score of the document to the given query.
Examples for $w(d)$ are:

1. $$w(d) = \frac{1}{d.r_q},$$

   where $d.r_q$ is the position of the document d in the score sorted result list relevant to given user query.

2. $$w(d) = d.score_q,$$

   where $d.score_q$ is the relevance score of document d for query q.

Intuitively, the interestingness at a time point b is simply the sum of document weights of all the documents that belong to $top_k(t_i)$ and they do not belong to $top_k(t_{i-1})$. This function in a way captures the weighted count of high scoring new documents created at each time point. In other words, the more documents with higher r appear in $\delta_r(t_i)$, the more interesting that time point t_i will be. It is easy to see that $\delta_r(t_i)$ defined above is monotonically decreasing when compared to frequency based or top-k similarity based interestingness. It is easy to see that $\delta_r(t_i)$ is a monotonically decreasing for the decreasing values of $d.r_q$. Assuming $d.r_q$ is always positive, $\delta_r(t_i)$ is always decreasing with $d.r_q$. .

In our setting, we are reading relevant results for a given query in their score order and then we explore their time dimension. To make this process efficient, we need to read as few relevant results as possible. We can exploit the fact that the result list for the given query is already in score order to get a bound on the interestingness. If we want to facilitate early pruning while computing interestingness, there are certain properties of interestingness function which must be met. One important property is the ability to upperbound on the interestingness value. This property allows early pruning of documents which we need to process without exactly knowing their relevance scores. A monotone interestingness function can largely facilitate upper bound computation. A function F , defined on predicates $p_1, ..., p_n$, is monotone if $F(p_1, ..., p_n) \leq F(p_1^{'}, ..., p_n^{'})$ whenever $p_i \leq p_i^{'}$ [12]. One such function which satisfies monotonicity is a function which is based on relevance score or rank of the documents defined above.

4.5 Naïve Approach

If we want top-k results at all time points in $\mathcal{T}$, a simple approach would be to accumulate $top_k(t_i)$ at these time points as we read the result set R. We can stop reading the result set R once we are sure that we obtained $top_k(t_i)$ for all $t_i \in \mathcal{T}$. Pseudocode for the same is shown in Algorithm 4.1

4.5.1 Naïve Algorithm

Algorithm 4.1 Algorithm to compute $top_k(t_i)$ at all time points in $\mathcal{T}$ and also determine $top_m(q)$ for a given query q

Input: Score sorted result list R for keyword query q

Output: Top-m interesting time points with interestingness defined as above

1: $A \leftarrow Matrix[|\mathcal{T}|][k]$, top-$m \leftarrow \varnothing$, priority-queue $\leftarrow \varnothing$
2: **while** There are more results in R or not all time points have k results **do**
3: Read the document d in top of the list R
4: Append the document d at columns $A[d.t_b]$ to $A[d.t_e]$. If $d \in \mathcal{D}$ then append from column $A[d.t]$ to the last column of the matrix A. Skip a column if it already has k documents.
5: Update $\delta(t_i)$ in priority-queue, if $d \notin top_k(t_{i-1})$ where $t_b \leq t_i \leq t_e$
6: **end while**
7: top-$m \leftarrow$ m most interesting time points from priority-queue

4.5.2 Algorithmic Analysis

The approach is very simple but has following limitations.

1. The algorithm runs n times in the worst case. And in each iteration, the document can be inserted at $|\mathcal{T}|$ time points in the worst case. This makes the algorithm $O(n \cdot |\mathcal{T}|)$. If we don't know $|\mathcal{T}|$ in advance it doesn't give any deterministic guarantee on run time.

2. There is lot of memory being wasted because of duplication. In the worst case, each document can be replicated $|\mathcal{T}|$ times. This makes the space complexity of the algorithm $O(n \cdot |\mathcal{T}|)$, again making its performance non deterministic.

3. The major limitation is that if the granularity of time points in $\mathcal{T}$ are reduced to milliseconds, then $|\mathcal{T}|$ becomes extremely high making the algorithm impossible to run.

In Chapter 5, we experimentally confirm the above limitations. And on the other hand, obtaining the $top_k(t)$ for any time point t can be done in O(1).

4.6 Approach for Generic Data Model $\mathcal{D}$

For Wikipedia-like archives [2] each document version has a begin time and and end time. In this work we treat each version of a document as a different document, i.e, each document can be visualized as an interval of time (time segment) which spans from, $d.t_b$ to $d.t_e$ of the time axis. And each time segment has a global ranking for a given query. It is a 1.5 dimensional (1 dimension for time and 0.5 dimension for rank) range data. In this Section we propose a new approach to retrieve interesting time points for generic data model $\mathcal{D}$. Also we materialize the top-k results at the interesting time points for a given query and make them searchable in logarithmic time. This is done using the Rank-Aware Dynamic Segment Tree introduced in the previous chapter. Also in oder to facilitate early pruning of the result list, we must integrate the NRA technique used in Algorithm 4.5 with Rank Aware Dynamic Segment Tree. In this section we present the algorithm to do the same and analyze the performance of the algorithm. Note that solution for special data model II $\mathcal{D}''$ is just a special case of this approach.

Challenges According to the problem definition, we need top-m interesting time points for a given query by analyzing the result list for the query given by a retrieval system. Also we should be able to answer the top-k results at these time points efficiently. This becomes slightly complicated with the generic data model $\mathcal{D}$. Some of the challenges to be addressed are listed below.

1. Since each document dies after an arbitrary amount of time, the top-k results are affected at the point where the document dies
2. A document d does not make it to the top-k set of all the time points between $(d.t_b, d.t_e)$. At time points where there are already top-k results it is not added
3. Since the interestingness is not just affected at the begin time of the document but also at the time points where the document makes it to top-k set
4. Maintaining top-k at all time points is not trivial

4.6.1 Extensions to Rank Aware Dynamic Segment Tree

In order to facilitate early termination using NRA algorithm we need some extensions to the Rank-Aware Dynamic Segment Tree. Firstly, we need to track number of results we have seen for all the time points in $\mathcal{T}$. Secondly, we should be able update interestingness of the time points affected by the result document in consideration.

Maintaining Top-k for Every Time Point

Again the very fact that the whole interval domain $\mathcal{I}$ is partitioned into disjoint atomic intervals by the segment-tree can be used to maintain top-k results at every time point. As we insert the time-intervals into the tree, we can compute the number of segments available for the extent interval $interval(Node)$ of a child of node N by the recursive function

$$sum(Node) = Node.\#segments + sum(Node.left) + sum(Node.right)$$

where $Node.\#segments$ is the number of segments stored at node N

when sum(N.left) and sum(N.right) both are k for a node N then we say that all time points in $interval(N)$ have k results. Although the sum is defined recursively it can be implemented in such a way that the sum is readily available at each node while inserting. Also when an interval $interval(N)$ is filled with k results we propagate the message upwards to its parent recursively, if even the parent realizes that $interval(N.Parent)$ is filled with k (by checking the number of results at the interval of other child) the message propagates further up. In the best case the message propagates till root then we know that we have top-k results at all time points and we can terminate reading more results.

Incrementally Computing and Updating Interestingness For Time Points

According to the definition of change, we can expect change only at the time points of the nodes(end points of the atomic intervals) in the tree. And the interestingness can be computed incrementally whenever a segment is inserted. It is easy to see that if b is the begin time of a segment, and is not yet k-complete then the interestingness at b is increased by $1/r$ (where r is the rank of segment being inserted). But it may so happen that the document might disappear from the top-k set of a time point p and appear back in top-k at q where $q > p$. To take care of this case, at each node N we have to make decision

Algorithm 4.2 UpdateInterestingness(segment, N)

1: **if** interval(N) has $<$ k results and interval(N).$lb - 1$ has k results **then**
2: $r \leftarrow$ segment.$rank$
3: interestingness at interval(N).lb will be incremented by $1/r$
4: $l \leftarrow interval(N).size$
5: update best-interestingness(interval(N).lb) $+= \sum_{i=1}^{k-l} \frac{1}{d.r_q+i}$
6: **end if**

4.6.2 NRA Algorithm

Since interestingness is monotonically decreasing, we can evaluate the best possible interestingness for all the time points. And with a separate priority queue we can easily get the best interestingness possible at any instant of the algorithm and compare it with the m^{th}-highest interestingness we have seen so far. If the best interestingness is less than the worst interestingness seen so far then we can terminate. Assuming the interestingness is updated using the method in Algorithm 4.2 , it is easy to integrate the NRA style early termination with scored dynamic segment tree. Algorithm 4.3 gives an idea of how interestingness can be computed while we are inserting the segments and terminate early if NRA condition is met.

Algorithm 4.3 NRA algorithm for retrieving top-m interesting time points from generic data collection

Input: Score sorted result list R for keyword query q

Output: Top-m interesting time points with interestingness defined as above

top-$m \leftarrow \varnothing$; min-$m \leftarrow 0$; priority-queue $\leftarrow \varnothing$;
while There are more results or top-m interesting time points are not retrieved **do**
  read the next top document d from R with begin time $d.t_b$, end time $d.t_e$ and rank $t.r_q$
  insert nodes with keys $d.t_b$ and $d.t_e$ in tree
  if tree goes out of balance **then**
    RebalanceSegmentTree(b)
  end if
  find least common ancestor LCA node for time range ($d.t_b$, $d.t_e$)
  priority-queue $\leftarrow$ CopySegment(d, LCA)
  best-interestingness $\leftarrow$ max{best-interestingness(t) | t $\in$ priority-queue}
  worst-interestingness $\leftarrow$ priority-queue.m^{th}-highest interestingness
  if worst-interestingness $\geq$ best-interestingness and priority-queue.size $\geq m$ **then**
    top-$m \leftarrow$ m most interesting time points from priority-queue1
    exit
  end if
end while

Algorithm 4.4 CopySegment(segment, LCA)

Input: Least Common Ancestor(LCA) to the segment being inserted

Output: Copy the segment in all valid atomic intervals and update the interestingness of affected time points

```
1: top-m ← ∅; min-m ← 0; priority-queue ← ∅;
2: if interval(LCA).lb ≥ segment.b and interval(LCA).ub <= segment.e then
3:    LCA.add(segment)
4:    UpdateInterestingness(LCA, segment.r)
5:    if priority-queue.contains(LCA) == false then
6:       add LCA to priority-queue
7:    end if
8:    if interval(LCA) has k results then
9:       propagate it to parent recursively
10:      if interval(parent) of all nodes have k results then
11:         return priority-queue
12:      end if
13:   end if
14:   return
15: else if interval(LCA).left.lb >= segment.b and interval(LCA).left.ub <= segment.e then
16:   CopySegment(segment, LCA.left)
17: else if interval(LCA).right.lb >= segment.b and interval(LCA).right.ub <= segment.e then
18:   CopySegment(segment, LCA.right)
19: end if
20: return priority-queue
```

Consider the example of a result set from $\mathcal{D}$ archive for a given query as shown in Figure 4.1. For simplicity again we will assume $f(d.r_q) = d.r_q$. Also let us assume the parameter k = 3 and we need top-2 interesting time points. After inserting documents from rank 1 to 6 i.e. $\{d_3, d_1, d_5, d_2, d_6\}$, the segment tree looks like in Figure 4.3. Now the next document to be inserted is $(d_8, 0, 100, 60, 6)$, because these time ranges already have top-3 documents alive. Document d_8 spans the entire time range of the time points we have seen so far. But it is important to note that document d_8 does not make it to the top-k at time ranges $(45, 60), (65, 75), (75, 85)$. So apart from interestingness at time point 0 being affected, the interestingness at time points 60 and 85 are also affected. Also since the time range $(0, 100)$ is already covered by the existing nodes, no new node will be inserted. Instead the document d_8 will be copied at nodes $(0, 30), (30, 45), (60, 65), (85, 100)$.

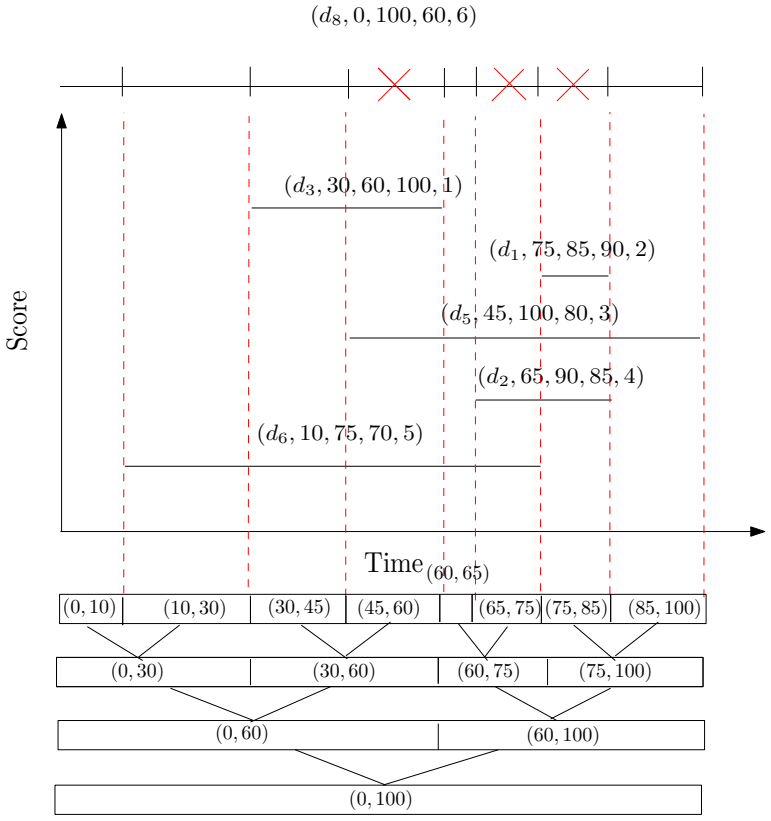


Figure 4.3: Intermediate step of Algorithm 4.3

Early Termination When inserting a document, at most two new potential interesting time points are introduced in the system. For example, at phase 1 of the algorithm time points 30 and 60 are introduced and their interestingness according to definition are 1.0 and 0 respectively. So time point 30 becomes top-1 time point while time point 60 becomes a candidate for interesting time point. Whenever an interval is inserted, we estimate the best possible interestingness we can achieve from the candidates and compare it with the interestingness of the top-2 time point. This comparison fails until phase 5. In phase 5 when we insert $(d_8, 0, 100, 60, 6)$, the candidate time points 45 and 65 will no more be candidates because they already got k = 3 highest documents. So the best candidate we have is time point 10 with interestingness 0.2, which means the best possible interestingness we can achieve from unseen documents is 0.465, which is less than our worst interestingness in top-2 interesting time points. So we terminate the algorithm after reading 6 documents. All the phases are depicted in Table 4.1.

Table 4.1: Different phases of Algorithm 4.3

Phase	Top-2 time points	Candidates	Best Unseen Interestingness	Worst Interestingness Seen
1	$\{(30, 1.0)\}$	$\{(60, 0)\}$	1.08	1.0
2	$\{(30, 1.0), (75, 0.5)\}$	$\{(60, 0), (85, 0)\}$	0.78	0.5
3	$\{(30, 1.0), (75, 0.5)\}$	$\{(45, 0.33), (60, 0), (85, 0), (100, 0)\}$	0.78	0.5
4	$\{(30, 1.0), (75, 0.5)\}$	$\{(45, 0.33), (65, 0.25), (60, 0), (85, 0), (100, 0)\}$	0.78	0.5
5	$\{(30, 1.0), (75, 0.5)\}$	$\{(10, 0.2), (60, 0), (85, 0), (100, 0)\}$	**0.465**	**0.5**

4.6.3 Algorithmic Analysis

The algorithm involves following major steps.

1. Inserting the begin time and end time of the document in the tree if they don't exist already.
2. Find the Least Common Ancestor(LCA) of the time range for which the document is valid.
3. Copy the segment recursively in left sub tree and right sub tree of the LCA recursively.

Assuming that there are n relevant documents in the corpus for the given query, step 1 costs $O(\log n)$. And in step 2 in the worst case LCA can be a leaf node, which makes its cost also $O(\log n)$. Analysis of Step 3 is not so trivial. It is easy to show that a segment being inserted in the segment tree is copied at maximum two nodes at the same depth. For proof please refer to [10]. And since there are $O(\log n)$ levels, the copy operation is done in $O(\log n)$. Also we have a constant number of rotations at each insertions. And each rotation costs a constant time (For proof refer to Chapter 3). But we have extra cost because of the extensions we have to the Rank Aware Dynamic Segment Tree. After each insertion, we have to update the count of documents to maintain top-k at every time point. This process in the worst case can propagate from leaf to the root, costing $O(\log n)$. And there is also cost of maintaining data structures to facilitate early termination. In the worst case each segment inserted can have distinct time point, making $n - k$ time points candidates for the interesting time points. So we can expect the size of the candidates to grow up to $n - k$, which means the cost of maintaining it is $O(\log n)$. Also if we want to maintain the documents sorted according to their score order at each node, then there is an additional cost of $O(\log \log k)$. In summary the total

cost of an insertion in Rank-Aware Dynamic Segment Tree is $O(n \cdot (C \cdot \log n + \log \log k))$, which is asymptotically still $O(n \log n)$.

4.7 Approach for Special Data Model $\mathcal{D}'$

For newspaper archives such as the New York Times Annotated Corpus [1] as described in Section 4.2, each document has a publication time and no end time. For such a data collection we have a simple and and efficient algorithm. Using this algorithm we can compute and store top-k results for any given keyword query at all time points and also identify interesting time points in the process. Unlike generic data model $\mathcal{D}$ in $\mathcal{D}'$ we have only begin time, so it becomes very simple to identify interesting time points. The algorithm works in the following way.

1. For a given query, retrieve the relevant results in their relevance score order using any top-k retrieving algorithms like TA or NRA.
2. Read the documents retrieved by top-k algorithm one at a time in their score order and update the interestingness at the begin time of the documents.
3. Estimate the best possible interestingness that any time point can get. The time point with the best possible interestingness must be the one which we have already seen.
4. Check if the estimated best interestingness is worse than the worst interestingness we have so far, if this is the case, then we terminate reading otherwise ask top-k algorithm for next best documents and repeat steps 2 to 4 until all the relevant documents are processed.
5. After termination, the top-m time points with the best interestingness are the most interesting time points.

For pseudocode please refer to Algorithm 4.5. One important point to be noted here is that to be able to terminate early we need to have interestingness function based on the rank or the score.

4.7.1 NRA Algorithm for Early Termination

Algorithm 4.5 NRA algorithm for retrieving top-m interesting time points from Special Data Model $\mathcal{D}'$

Input: Score sorted result list R for keyword query q

Output: Top-m interesting time points with interestingness defined as above

top-$m \leftarrow \varnothing$; candidates $\leftarrow \varnothing$; buffer[k] $\leftarrow \varnothing$; hash-map $\leftarrow \varnothing$;
while There are more results or not all time points have k results **do**
  worst-interestingness = m^{th} most interestingness in top-m
  while size of buffer $< k$ **do**
    read next top document d with begin time $d.t$ from R add it to buffer
    for all time points t in candidates **do**
      if $d.t \leq t$ **then**
        size(t)++, l = k - size(t)
        update best-interestingness(t) = interestingness(t) + $\sum_{i=1}^{l} \frac{1}{d.r_q+i}$
      end if
    end for
    if $d.t$ does not exist in candidates **then**
      add $d.t$ to candidates with interestingness($d.t$) = $1/f(d.r_q)$ and size($d.t$) = 1
    else
      update interestingness($d.t$) = interestingness($d.t$) + $1/f(d.r_q)$
      l = k - size($d.t$)
      update best-interestingness($d.t$) = interestingness($d.t$) + $\sum_{i=1}^{l} \frac{1}{d.r_q+i}$
    end if
    if interestingness($d.t$) > worst-interestingness **then**
      remove last time point in top-m and and add it to candidates
      add $d.t$ to top-m and remove $d.t$ from candidates
    end if
  end while
  insert an entry in hash-map with key t_{max} and value as buffer
  Let the documents with max begin time t_{max} in buffer be $\mathcal{D}_{max}$
  threshold = max(best-interestingness in candidates)
  if best-interestingness(t_{max}) < worst-interestingness **then**
    remove t_{max} from candidates
  end if
  delete $\mathcal{D}_{max}$ from buffer
end while
return top-m

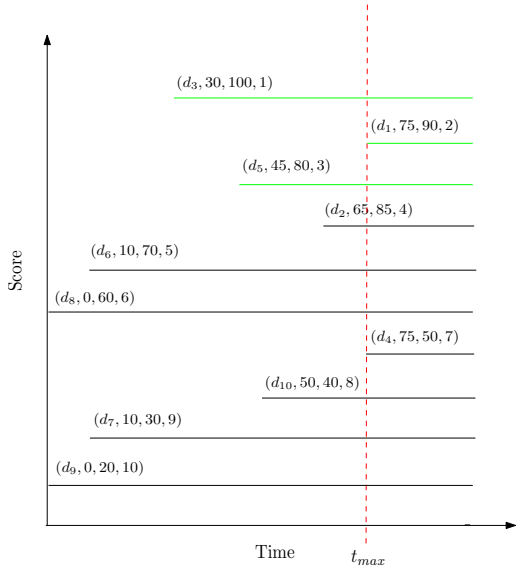


Figure 4.4: Phase 1 of Algorithm 4.5

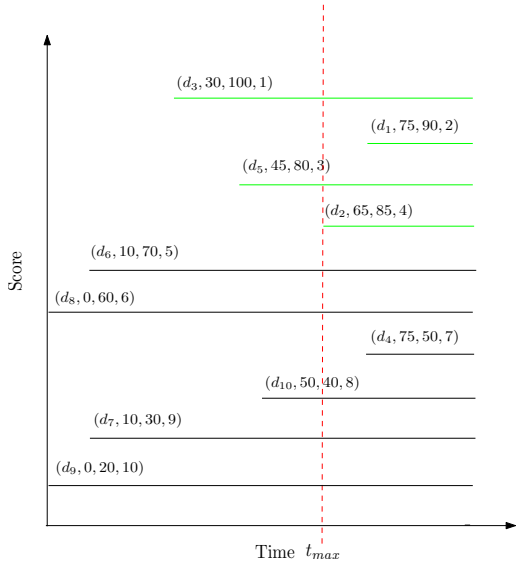


Figure 4.5: Phase 2 of Algorithm 4.5

Notice that for simplicity we use $w(d) = \frac{1}{d.r_q}$ as weight function but in principle we can have the following weight functions.

1. Logarithmic in rank: $w(d) = \frac{1}{\log d.r_q}$.
2. Square root of rank: $w(d) = \frac{1}{\sqrt{d.r_q}}$.
3. Linear in score: $w(d) = d.s_q$.

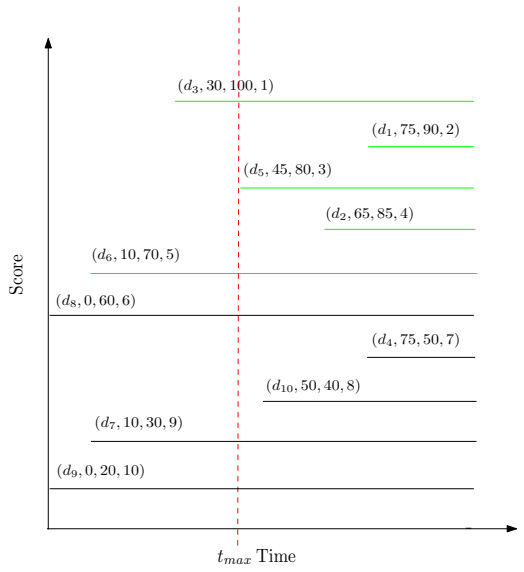


Figure 4.6: Phase 3 of Algorithm 4.5

Table 4.2: Different phases of Algorithm 4.5

Phase	Top-2 time points	Candidates	Best Unseen Interestingness	Worst Interestingness Seen
1	$\{(30, 1.0), (75, 0.5)\}$	$\{(45, 0.33)\}$	0.58	0.5
2	$\{(30, 1.0), (75, 0.5)\}$	$\{(45, 0.33)\}$	0.53	0.5
3	$\{(30, 1.0), (75, 0.5)\}$	$\{(10, 0.2)\}$	**0.5**	**0.5**

Consider an example of result list for a given query q in a Special Data Model $\mathcal{D}'$ as shown in Figure 4.4. Let us assume that we want top-2 interesting time points with k = 3 as the parameter. For simplicity let us assume that $f(d.r_q)$ is linear. i.e. $f(d.r_q) = d.r_q$.

According to the algorithm in phase 1 we read the first 3 documents. Each tuple in top-m and candidates contain <timepoint, interestingness, size>. Now, time points (30, 1.0, 1) and (75, 0.5, 3) make the top-2 interesting time points we have seen so far. And (45, 0.33, 2) becomes the candidate. We can estimate the best interestingness this time point can get i.e, 0.33 (current interestingness of 45) + 0.25 (based on the rank of the next document in result list) = 0.58. After phase 1 we remove document d_1 from the buffer. In phase 2 we read document d_2 and t_{max} in this phase is 65. And since we already have 3 documents alive at time point 65, we know the exact interestingness value, which is 0.25. So it does not make it to the top-2 time points. But since our rank of the unseen documents has gone down to 5, the best interestingness also drops to 0.53. In phase 3 we read document d_6 and now, t_{max} is 45, so time point 45 will not be in candidates anymore. And time point 10 becomes a candidate. We estimate the best interestingness of the only candidate 10 and it turns out to be 0.5 which is equal to the

worst interestingness in top-2. So we are guaranteed that we cannot expect any time point with interestingness more than 0.5 and hence we can stop reading more documents here. All the phases are depicted in Table 4.2.

4.7.2 Algorithmic Analysis

The analysis of the algorithm is straightforward. Assuming that we have n documents in the corpus, in the worst case, the algorithm reads all n documents making its run time complexity $O(n)$. But the algorithm could terminate early in the following two cases:

1. When the best estimated interestingness drops below the worst interestingness of the m^{th} most interesting time point we have so far. In this case we have a guarantee that any of the unseen time points will not be more interesting than the time points we have seen so far.
2. When there are at least k results at every time point of the entire duration of the archive. Once we have k results at every time point we can stop reading the result list. We are guaranteed that the top-m interesting time points we have so far are the best.

The algorithm has extra complexity because of the accumulators used to facilitate early termination using NRA technique. This involves maintaining the priority queues which involves inserting, updating, and removing from the priority queue. At each iteration of the algorithm, we use a constant number of priority queue operations on data structure candidates. The priority queue can grow at the most up to constant size k. So the overall asymptotic runtime complexity of Algorithm 4.5 still remains O(n).

4.8 Prototype Implementation

In this section, we present the end-to-end frame work to identify interesting time points.

4.8.1 Architecture

The system involves the following three layers

1. Text retrieval layer.
2. Interesting time points identification module.

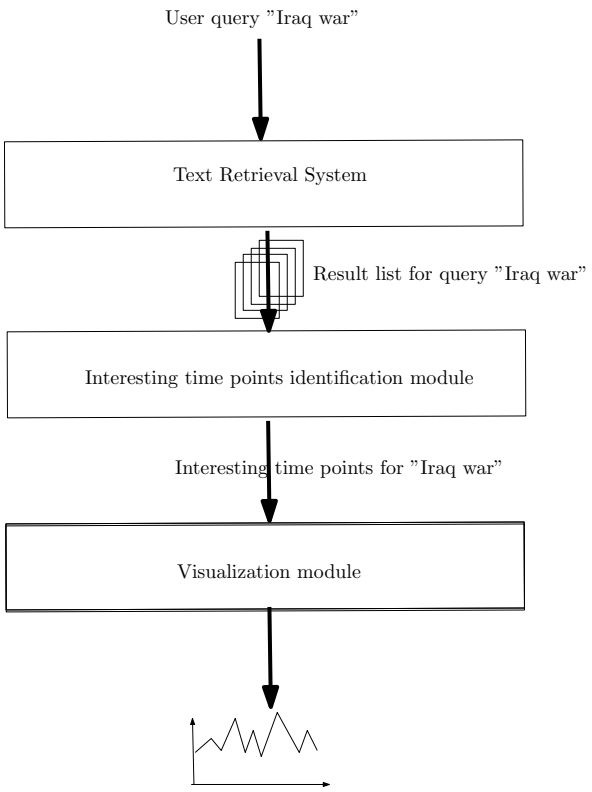


Figure 4.7: Data Flow Diagram of System Prototype

3. Visualization.

Before we discuss each layer in detail, we will describe the interaction between these layers. The user query is received by the "text retrieval layer". Then the text retrieval layer processes the query and returns the relevant results in the score order. The retrieved results are passed on to "interesting time points identification module". In this module depending on the type document collection, the approaches described for various data models in this Chapter are applied on the result set. The output of this module is the interesting time points for the given user query. Finally, the results are presented with graphical visualization of the interesting time points, making it easy for user to browse the time points. The end-to-end data flow is shown in the Figure 4.7

4.8.2 Text Retrieval System

The text retrieval system consists of an inverted index built on the documents. In our prototype the inverted index was static and the index is stored on Oracle 11g database system. There are four main tables in the index, they are described below.

1. **Lexicon:** $< tid, term >$ where tid is the unique identifier for the keyword term.

2. **TF:** $< tid, did, ts, te, tf >$ where tid is term id, did is the unique identifier of document which contains the term tid, ts and te and are the begin time and end time of the document did.

3. **DF:** $< tid, df >$ where df is the document frequency of the term tid.

4. **Document:** $< did, length >$ where $length$ is the length of the document did in words.

Given a user query, we query the database to retrieve the relevant documents in their score order. We strictly follow conjunctive query semantics, i.e., all query keywords are mandatory for a document to be reported as a result. And to compute the relevance score we tried both simple TF-IDF and Okapi BM25[3] and we zeroed in on Okapi BM25 because of its high quality results. The score of a document d for a given query q with keywords $t_1, t_2, ..., t_l$ is given as below.

$$score(d, q) = \sum_{i=1}^{l} IDF(t_i) \cdot \frac{f(t_i, d) \cdot (k_1 + 1)}{f(t_i, d) + k_1 \cdot (1 - b + b \cdot \frac{|d|}{avdl})},$$

where $f(t_i, d)$ is term frequency of the keyword t_i in the document d, $|d|$ is the length of the document d in words, and $avdl$ is the average document length in the document collection and the parameters typically k_1 and b are set to 1.2 and 0.75 respectively. And $IDF(t_i)$ is the IDF (inverse document frequency) weight of the query keyword t_i. It is computed as below.

$$IDF(t_i) = \log \frac{|\mathcal{D}| - df(t_i) + 0.5}{df(t_i) + 0.5},$$

where $|\mathcal{D}|$ is the total number of documents in the document collection from which the results are drawn, and $df(t_i)$ is the document frequency of the keyword t_i.

4.8.3 Interesting Time Points Identification Module

The input to this module is the query result list for the given query, data collection type, parameters like k (top-k results at each time point) and m (number of interesting time points required). Depending on the type of data collection, an appropriate approach is used with the input data and parameters. Finally this module outputs the interesting time points it identifies.

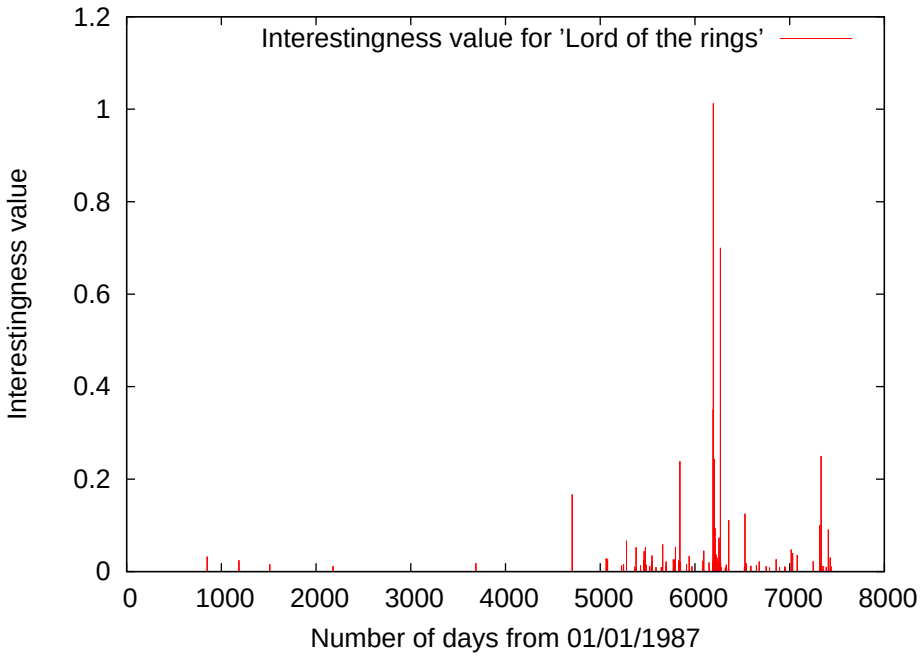


Figure 4.8: Visualization of interesting time points

4.8.4 Visualization Module

This final module is to present the identified time points in a graphical visualization. The graph consists of time in x-axis and interestingness value in y-axis. The visualization consists of impulses representing the interestingness. In Figure 4.8 a snapshot of the visualization of the interesting time points identified for the query "Lord of the rings" is shown.

Chapter 5

Experimental Evaluation

5.1 Experimental Setup and Data Set

We used two data sets in the experiments, the New York Times Annotated Corpus (referred to as NYT in the rest of the chapter) and the English Wikipedia revision history (referred to as WIKI in the rest of the chapter). In this chapter, we conduct an experimental evaluation of the performance gains of our proposed methods. In particular, we study the execution speed, memory usage during processing, and effectiveness of early termination methods in reducing the amount of I/O. We also provide some anecdotal evidence to show that the time-points found by our methods are realistic. We describe the two data sets NYT and WIKI in detail below.

New York Times Annotated Corpus (NYT) The New York Times Annotated Corpus [1] data set contains a total of $1,855,656$ articles published by New York Times between 1987 and 2007. In contrast to the other two data sets, NYT contains only one version per document. We strictly followed our definition of data model ($\mathcal{D}'$) from Section 4.2.2, the validity time-interval of each document would range from its publication time until the current time *now*. We also consider a variation of NYT data collection, by giving a validity time-interval of 90 days. NYT90 corresponds to our special data model II $\mathcal{D}''$ described in Section 4.2.3. NYT90 has a counterpart in the real world, namely, if the New York Times articles were freely accessible only for a limited time after their publication. We will refer to this data set as NYT90 in the rest of the chapter.

Revision History of the English Wikipedia (WIKI) The second data set used in our experiments is wikipedia collection, the revision history of the English Wikipedia [2], which is available for free download as a single XML file. This data set corresponds to the

generic data model $\mathcal{D}$ defined in Section 4.2.1. This large data set, whose uncompressed raw data amounts to 0.7 TBytes, contains the full editing history of the English Wikipedia from January 2001 to December 2005. We indexed all articles excluding versions that were marked as the result of a minor edit (e.g., the correction of spelling errors etc.). This yielded a total of 1,517,524 documents with 15,079,829 versions having a mean (μ) of 9.94 versions per document at standard deviation (σ) of 59.18. Average life time of a version is 8.8 days.

Queries To evaluate our approaches we chose two different types of queries.

1. We used USA all-time box-office movies listed in IMDB [20] as our set of queries both for performance and qualitative analysis. For NYT the movies released between the year 1987 and 2007 were chosen. The list of queries is given in picture Figure 5.1.

2. The second set of queries, we used queries constructed from the list of events of international significance listed in Wikipedia [21]. Each event is encoded by a set of keywords which appear as hyperlinks. A sample of the list of queries is given in picture Figure 5.2.

Implementation Both the data collections were indexed and the corresponding inverted indexes were stored on an Oracle 11g database server. The index was read from Java programs using JDBC driver. All the experiments described below were run on a single machine with Intel(R) Core(TM)2 Duo CPU E6750 @ 2.66GHz, 3GB RAM, Debian 64 bit OS (Linux 2.6.30.5.2.amd64-smp). All experiments were repeated 3 times and their average values are used to evaluate the approaches. For processing keyword queries, we employ conjunctive semantics, i.e., all query keywords are mandatory for a document to be reported as a result. We use Okapi BM25 [3] , as a relevance model to rank documents.

5.2 Results and Comparison

In this section we analyze the performance of our approaches for the two data sets we mentioned above. As a baseline for comparison we use the naïve method mentioned in the section 4.5. Since we have different solutions for different data models ($\mathcal{D}$ and $\mathcal{D}'$),

beverly hills cop, 3 men and a baby, good morning vietnam, fatal attraction, crocodile dundee, big, who framed roger rabbit, rain man, coming to america, parenthood, the little mermaid, ghostbusters, honey i shrunk the kids, lethal weapon 2, indiana jones and the last crusade, driving miss daisy, look whos talking, batman, back to the future part, teenage mutant ninja turtles, total recall, dances with wolves, pretty woman, die hard 2, ghost, dick tracy, home alone, the hunt for red october, the silence of the lambs, the addams family, sleeping with the enemy, beauty and the beast, terminator 2 judgment day, city slickers, robin hood prince of thieves, hook, batman returns, the bodyguard, unforgiven, lethal weapon, a few good men, waynes world, a league of their own, aladdin, sister act, basic instinct, mrs doubtfire, the pelican brief, the firm, the fugitive, in the line of fire, sleepless in seattle, indecent proposal, jurassic park, forrest gump, the lion king, the flintstones, clear and present danger, the mask, maverick, interview with the vampire the vampire chronicles, pulp fiction, true lies, the santa clause, dumb and dumber, die hard with a vengeance, goldeneye, pocahontas, casper, apollo 13, ace ventura when nature calls, batman forever, toy story, se7en, jumanji, the birdcage, twister, ransom, the rock, phenomenon, the nutty professor, 101 dalmatians, mission impossible, the first wives club, eraser, a time to kill, jerry maguire, independence day, the hunchback of notre dame, face off, as good as it gets, good will hunting, titanic, my best friends wedding, batman and robin, contact, con air, the lost world jurassic park, tomorrow never dies, scream, air force one, liar liar, george of the jungle, deep impact, rush hour, enemy of the state, patch adams, the prince of egypt, the truman show, theres something about mary, lethal weapon 4, doctor dolittle, saving private ryan, the rugrats movie, the waterboy, shakespeare in love, a bugs life, mulan, youve got mail, godzilla, american beauty, double jeopardy, the sixth sense, american pie, analyze this, the matrix, wild wild west, the generals daughter, the mummy, stuart little, austin powers the spy who shagged me, toy story 2, tarzan, the blair witch project, the world is not enough, the green mile, runaway bride, sleepy hollow, notting hill, dinosaur, mission impossible , miss congeniality, nutty professor the klumps, the patriot, chicken run, cast away, erin brockovich, meet the parents, xmen, traffic, charlies angels, what lies beneath, big mommas house, the perfect storm, scary movie, wo hu cang long, gladiator, gone in sixty seconds, how the grinch stole christmas, remember the titans, what women want, the mummy returns, vanilla sky, lara croft tomb raider, spy kids, dr dolittle 2, a beautiful mind, the princess diaries, jurassic park i, black hawk down, rush hour 2, planet of the apes, the fast and the furious, american pie 2, pearl harbor, shrek, hannibal, harry potter and the sorcerers stone, oceans eleven, monsters inc, the lord of the rings, sweet home alabama, the santa clause 2, harry potter and the chamber of secrets, men in black, ice age, lilo and stitch, mr deeds, the sum of all fears, catch me if you can, chicago, die another day, 8 mile, road to perdition, scooby doo, signs, spider man, xxx, austin powers in goldmember, the bourne identity, my big fat greek wedding, minority report, scary movie 3, how to lose a guy in 10 days, terminator 3 rise of the machines, swat, bringing down the house, bad boys , somethings gotta give, charlies angels full throttle, the cat in the hat, the italian job, cheaper by the dozen, elf, the matrix reloaded, finding nemo, pirates of the caribbean the curse of the black pearl, the matrix revolutions, daddy day care, seabiscuit, spy kids 3d game over, american wedding, x2, 2 fast 2 furious, anger management, daredevil, the last samurai, bruce almighty, hulk, freaky friday, collateral, fahrenheit 9 11, million dollar baby, the passion of the christ, troy, national treasure, the grudge, oceans twelve, harry potter and the prisoner of azkaban, lemony snickets a series of unfortunate events, dodgeball a true underdog story, the incredibles, the polar express, meet the fockers, spider man 2, the village, 50 first dates, van helsing, i robot, shrek 2, shark tale, the aviator, the day after tomorrow, the bourne supremacy, war of the worlds, robots, the pacifier, the 40 year old virgin, chicken little, mr and mrs smith, the longest yard, fun with dick and jane, the chronicles of narnia the lion the witch and the wardrobe, wedding crashers, star wars, madagascar, harry potter and the goblet of fire, batman begins, king kong, fantastic four, walk the line, charlie and the chocolate factory, casino royale, superman returns, over the hedge, borat cultural learnings of america for make benefit glorious nation of kazakhstan, talladega nights the ballad of ricky bobby, the da vinci code, 300, night at the museum, mission impossible, cars, the pursuit of happyness, happy feet, ice age the meltdown, dreamgirls, the devil wears prada, xmen the last stand, the breakup, pirates of the caribbean dead mans chest, the departed, harry potter and the order of the phoenix, hairspray, pirates of the caribbean at worlds end, spiderman 3, blades of glory, wild hogs, knocked up, paranormal activity, alvin and the chipmunks, ratatouille, rush hour, shrek the third, enchanted, bee movie, oceans thirteen, superbad, i am legend, evan almighty, ghost rider, national treasure book of secrets, the simpsons movie, 4 rise of the silver surfer, i now pronounce you chuck and larry, transformers, juno, live free or die hard, american gangster, the bourne ultimatum

Figure 5.1: USA all-time box-office movies from the year 1987 to 2007 [20]

united states environmental protection agency, congo civil war, defense secretary donald rumsfeld, apple computer mac os x, xbox united states, harry potter us, israel hamas, israel bank west bank terrorism israel palestinian, reformist, space shuttle columbia, iran osama bin laden united states, nuclear program north korea nuclear weapons, north korea nuclear weapons program, capture of saddam hussein, united states congress vice president al gore, colin powell war in iraq, pakistan india kashmir, war on terrorism terrorist pakistan united states, academy awards academy award for best actress, linux kernel, israel suicide bomber, exploration of mars, united states iraq constitution islam, united states department of homeland security justice department, israeli defence force, united states armed forces baghdad, occupation of iraq south korea iraq united states, traffic sign english, president united states public relations muslim arab, honduras brazil, president syria terrorism iraq, michael johnston, guantanamo bay, kofi annan bbc government, oregon oregon state police, republican national convention senator john kerry, stuttgart germany, president iraq united states iraq democratic process security economy, sports fa cup, us congress vice president dick cheney senate, natural disaster french, international monetary fund united states, tamil nadu, sunni muslim iraq, united states guantanamo bay, french president jacques chirac prime minister, palestinian authority israel palestinians, beheaded, nintendo gamecube japan, united states palestinian hamas, solar eclipse, ralph nader president of the united states, kofi annan canada united nations, kofi annan iraq, bbc aids china india, national institute of child health and human development, iraqi republican guard, politics of austria freedom party, grand national party south korea, wesley clark, us congress federal hall new york city, united nations united states iraq kofi annan, war on terrorism al qaeda afghanistan national security council, international court of justice united states mexican, un general assembly israel west bank, federal marriage amendment united states house of representatives, earthquake richter scale united states, saddam hussein possible death of saddam hussein british, bubble boy, malaysia website british, us senate syria, supreme court of ukraine, chechnya russia, itunes, usa al qaeda iraq saddam hussein, israeli defence force palestinian, iraq disarmament crisis, world water council water forum, vancouver international airport, london heathrow, vladimir putin russia, muslim united kingdom terrorism act 2000 terrorism, china and the united nations president of the republic of china united nations independent taiwan, haiti un, heathrow airport, sars, federal marriage amendment united states republican party united states constitution senate, united states senator john mccain, russia ballistic missile, ipod, politics of canada paul martin liberal party prime minister, hamas arab state israel, egypt sinai israel, occupation of iraq iraq insurgency, canada military iraq nato, 2003 california recall, iraq baghdad national assembly, us governing council of iraq iraq president, palestinians israel egypt gaza, israel, electronic voting, british airways air france, 2003 rugby union world cup, mary barnes, japan typhoon, british gloucester, iraq kurds, brussels eu, republican democratic john kerry debate florida, car bomb baghdad iraq police, president of the republic of china taipei, california domestic partner, administrative law judge, canadian alliance progressive conservative party conservative party of canada, united states republic of macedonia greece european union, united states united nations israel gaza strip,mecca muslim hajj pilgrimage, wto world trade organization mexico, bob graham larry king live, transit of mercury, likud party ariel sharon prime minister of israel, pakistan united press international military president pervez musharraf, sadr city baghdad, united nations bangladesh flood, car bomb baghdad insurgency, hans blix mohamed elbaradei united nations security council weapons of mass destruction saddam hussein colin powell, dennis weaver, iraq guerrillas, 2004 haiti rebellion haiti, taiwan affairs office republic of china taiwan independence, occupation of iraq tikrit iraq, national guardsmen, spain madrid european court of human rights canada, roger federer, worm microsoft windows, shanghai one child policy, british government hutton inquiry, united states secretary of education rod paige president white house margaret spellings, treaty federal law torture national security, taliban afghan voter, guatemala usd, immigration and naturalization service united states, hong kong tiananmen square protests of 1989, earthquake gujarat india, israel ramallah west bank, sars world health organization, scott mcclellan ari fleischer white house press secretary, lien chan, swiss air traffic control, gulfstream, nuclear weapon iran iran international atomic energy agency, civil unions in new zealand parliament civil union, quebec jean charest, south korea impeachment, judge coalition provisional authority, eddie guerrero brock lesnar, jurassic park iii, fallujah insurgents, mexico peru cuba fidel castro, great pyramid of giza, iowa democratic john kerry john edwards howard dean, israel rafah gaza strip, tom cruise nicole kidman, shenzhou 2, indonesia bomb attack bali, president south korea impeachment, palestinian legislative council palestinian authority yasser arafat, occupation of iraq red cross baghdad, internet explorer russia, green zone baghdad iraq, british bbc hutton inquiry, extrasolar planet, united states hainan, pakistan nuclear ballistic missile, mars exploration rover mission, aaliyah pop, car bomb mosul iraq civilian, rwanda un congo congo civil war, mexico honduras panama latin america, green party david cobb ohio,

Figure 5.2: Sample queries extracted from wikipedia events [21]

Table 5.1: Notations for Experiments

Notation	Description	Data Model	Data Set	Solution Approach
NYTNAIV	Naïve approach for NYT	$\mathcal{D}'$	NYT	Section 4.5
NYTIMP	Improved approach for NYT	$\mathcal{D}'$	NYT	Section 4.7
NYT90NAIV	Naïve approach for NYT90	$\mathcal{D}''$	NYT90	Section 4.7
NYT90IMP	Improved approach based on rank-aware dynamic segment tree for NYT90	$\mathcal{D}''$	NYT90	Section 4.6
WIKINAIV	Naïve approach for WIKI	$\mathcal{D}$	WIKI	Section 4.5
WIKIIMP	Improved approach based on rank-aware dynamic segment tree for WIKI	$\mathcal{D}$	WIKI	Section 4.6

we do the analysis of data model and solution approach combinations as shown in Table 5.1.

5.2.1 Performance Analysis for NYT and NYT90

We compare the impact of our improved interesting point detection techniques on NYT and NYT90 corpora. We measure (1) execution time, and (2) memory consumption of methods with and without early termination. Further, we characterize the effect of early termination as we increase the values of k and m. Since the performance varies with the size of the result set for a particular query, we group the queries based on the result size (denoted as N in rest of the chapter) they have in the entire corpus.

Execution Time

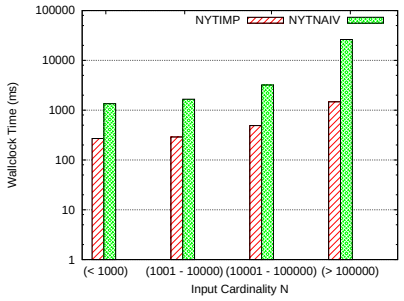
The aim of this set of experiments is to analyze the improvement in execution time of our improved approach when compared to the naïve approach. To quantify the results, we compare the average wall-clock times taken by the approaches earlier mentioned. We compare the CPU running times of the mentioned approaches, both with and without early termination in Figure 5.3. In Figure 5.3(a) we can see that for NYT corpus we can see improvement in our approach even without using the early termination approach. As

the lifetime of a document increases, the naïve approach incurs higher cost of updating the interestingness measure. Thus, our method which computes the interestingness value only at the begin times of the documents, outperforms the naive method by a large margin. Especially when the input is huge like $N > 100,000$ NYTNAIV performs badly because both parameters N and $|\mathcal{T}|$ in its runtime complexity $O(N \cdot |\mathcal{T}|)$ are quite high. But on the other hand, the improved approach NYTIMP, just has to look at N time points in the worst case. For the same reason, NYTIMP performs 17.8 times better than the naïve approach. A similar improvement in performance is observed in Figure 5.3(b). In this case, the improvement is attributed to the fact that, the naïve approach, NYT90NAIV updates interestingness at every time point, while we update the interestingness values withing the segment tree structure, leading to performance improvement.

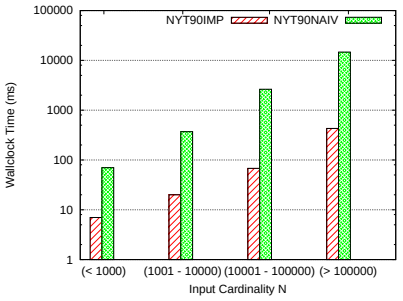
In Figure 5.3(c) and 5.3(d) we can see further improvements in performance because of the early termination effect. Especially when the input is huge, $N > 100,000$. But on the other hand, In Figure 5.3(c) we can see that for NYT data set, there is no difference in performance for $N < 1000$. This behavior is due to the fact that very small input size N. Smaller the time points in input data, lesser the effectiveness of early termination will be. But in Figure 5.3(d), we can see that for NYT90 data set, even with $N < 1000$ we see some improvement in performance because of early termination. This is because, the death of the document after a constant time, contributes to the interestingness of a time point. This effect speeds up the early termination, by accelerating the increase of worst interestingness and decrease of best estimated interestingness. On the contrary, in NYT data set, only creation of a document contributes to the interestingness as documents are assumed to live until current time *now*.

Memory usage analysis

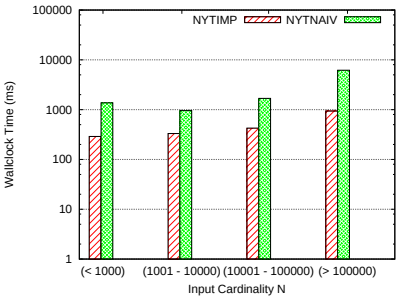
The aim of this set of experiments is to evaluate the memory usage of NYTNAIV and NYTIMP. The space complexity of NYTNAIV is $O(n \cdot |\mathcal{T}|)$ where n is the number of relevant documents to the query and $|\mathcal{T}|$ is the number of time points (refer to Section 4.5.2 for more details). But on the other hand in NYTIMP we store top-k results only at begin time points of the documents. Which means the space complexity of NYTIMP approach is $O(n)$. This is reflected in the Figure 5.4(a). But even though the space complexity of NYTIMP approach is $O(n)$ its memory usage is almost constant because, of the nature of the data. A huge number of documents have same begin time, which reduces the number of time points for which we need to store the top-k results. Also in Figure 5.4(b) for $N > 100000$ we can see that NYT90IMP consumes almost the



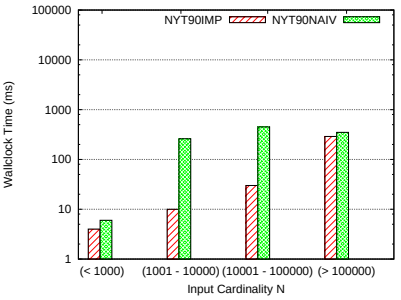
(a) NYT without early termination



(b) NYT90 without early termination

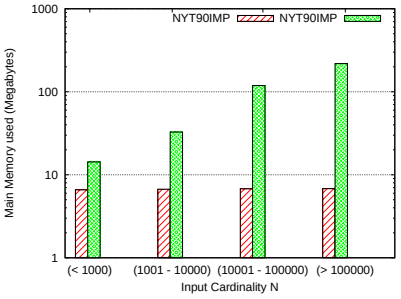


(c) NYT with early termination

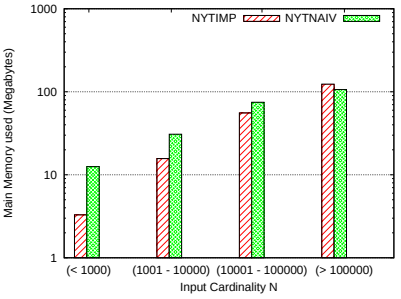


(d) NYT90 with early termination

Figure 5.3: Execution time analysis of NYT and NYT90 with parameters $k = 100$ and $m = 50$ (for early termination) and y-axis is in log-scale.



(a) Memory usage analysis for NYT



(b) Memory usage analysis for NYT90

Figure 5.4: Memory usage analysis with parameter $k = 100$ and no early termination. (y-axis to log-scale)

same main memory as NYT90NAIV. This is because, in general the space complexity of rank-aware dynamic segment tree is $O(n \log n)$ but when we fix the lifetime of a document as a constant, C then space complexity of NYT90IMP becomes $O(n \cdot |\mathcal{C}|)$ (same as NYTNAIV).

Effectiveness of Early Termination

In this experiment we investigate the effectiveness of early termination. To quantify the improvement achieved by the early termination, we look at the percentage of relevant documents read for a given query.

$$\text{Average percentage} = \frac{1}{|Q|} \cdot \left(\frac{\text{Number of tuples read to answer top-m time points}}{\text{total relevant results for the query}} \right) \cdot 100$$

where, $|Q|$ is the number of queries. For NYT, the effect of increasing value of m and k exponentially can be seen in Figure 5.5(a) and Figure 5.5(b). It is clear that as we increase k or m the early termination decreases. In NYT corpus for $m = 1000$ and $k = 1000$ there is no early termination at all. And with $N < 1000$ there is no early termination at all for any values of m and k for the reason mentioned in Section 5.2.1.

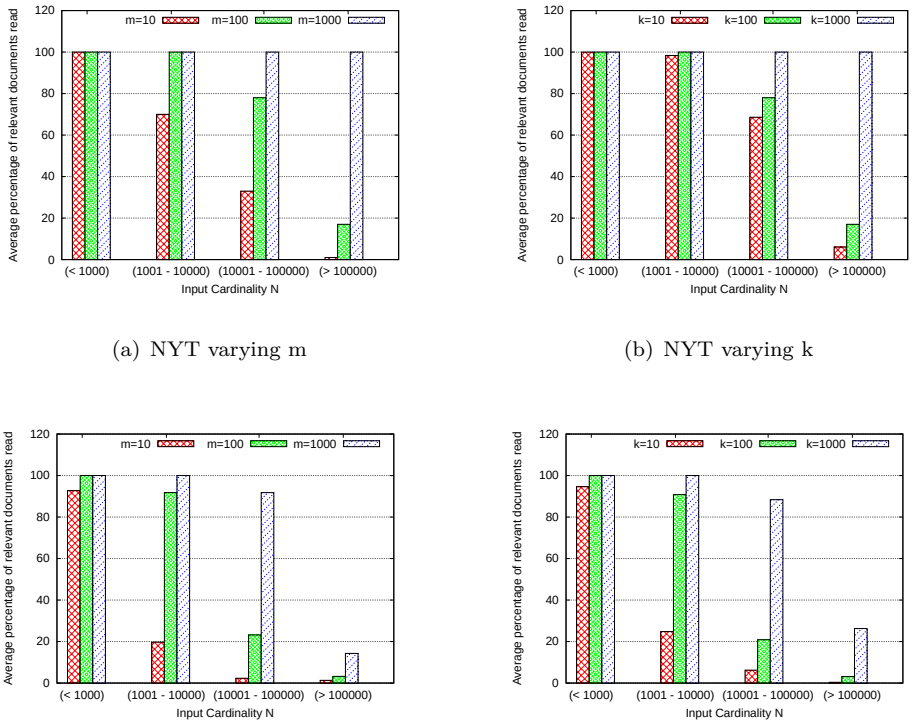
In Figure 5.5(c) and Figure 5.5(d) we can see the effectiveness of early termination for NYT90 with exponentially increasing values of m and k. It is interesting to see that unlike NYT corpus for $m = 1000$ and $k = 1000$ we can see early termination being effective as the value of N increases. Especially for $N > 100000$ the early termination is very effective. This is again because of the reason mentioned in Section 5.2.1.

5.2.2 Performance Analysis of WIKI

In this set of experiments we use the WIKI data set. WIKI is used to test the scalability of our approaches because it is larger than NYT by a factor of 10. Also since WIKI is a versioned collection, the nature of data is different from NYT and NYT90. This constitutes a different type of test case for our approaches. It is important to note that the granularity of timestamps of end time points of documents in WIKI is in milliseconds. Since naïve approach WIKINAIV works by tracking documents at each time points for a given granularity, having a granularity of milliseconds is impractical. So we round off the end times of documents to nearest day. On the other hand, rank-aware dynamic segment tree can support time points of any granularity. For experiments in this section, we use the parameters $k = 100$ and $m = 50$.

Execution Time

In Figure 5.6(a), we can see that even though we see some improvement for the WIKIIMP approach, it is not as significant as seen in NYT. This is because the lifetime of documents



(a) NYT varying m

(b) NYT varying k

(c) NYT90 varying m

(d) NYT90 varying k

Figure 5.5: Effectiveness of early termination in NYT and NYT90

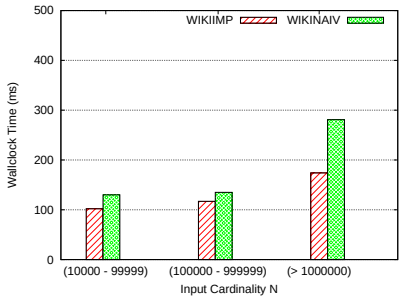
in WIKI are very small (with a mean of 8.8 days) when compared to NYT90, in which lifetime of documents is 90 days. On the contrary the runtime complexity of rank-aware dynamic segment tree remains same. But we can still see that WIKIIMP is almost 100ms faster than WIKINAIV for $N > 1000000$

Early-Termination Effect

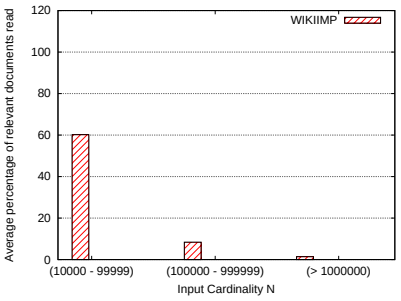
Early termination is very effective in WIKI too. Especially with higher input $N > 1000000$ it is enough to read 1.4% of input data to answer top-50 interesting time points. This can be seen in Figure 5.6(b)

5.2.3 Qualitative Analysis

Having analyzed the efficiency of our approaches, in this section we provide anecdotal evidence to the quality of interesting time points obtained. One interesting observation on NYT is that, the articles published are in synchronization with the real world events. For example when the US-led invasion on Iraq in 2003 begun, there were many news articles tracking the war in Iraq. So all the events with U.S. and international significance



(a) Execution time analysis of WIKI



(b) Effectiveness of early termination in WIKI

Figure 5.6: Performance analysis for WIKI with parameters $k = 100$ and $m = 50$

Table 5.2: Interesting time points for movie "The Lord of the rings (LOTR)"

Rank	Date	Event
1	17/12/2003	LOTR The Return of the King released
2	01/03/2004	LOTR The Return of the King won 11 Oscars
3	16/12/2003	-
4	29/01/2007	LOTR online game release
5	28/12/2003	$224 million since opening two weeks ago
6	29/12/2002	LOTR The Two Towers released
7	18/11/1999	First LOTR game
8	15/11/2004	-
9	27/05/2004	More LOTR games
10	15/01/2007	LOTR musical announced in London

are articulated in NYT news articles almost at the same time as the occurrence of the event. Keeping this observation in mind, we ran the keyword queries shown in Figure 5.1. In this part of the experiments, we annotate the interesting time points we obtained using approach in Algorithm 4.5. We chose 5 movie titles "The Lord of the Rings", "Forrest Gump", "Shakespeare in Love", "Shrek" and "Harry Potter" to annotate the important time points. We annotate the top-10 time points obtained using our approaches. From Table 5.2 to Table 5.6 list the 10 most interesting time points obtained for these queries on the NYT data set. We annotated these time points by checking the facts from IMDB and Wikipedia. The interesting time points obtained often coincide with the movies release dates or popular awards winning date. We observed that the interesting time points obtained from NYT and NYT90 for most of the movie titles are same except their ranks are permuted. Only for the movie "Forrest Gump" there are some new time points were seen. Also it is important to note that, the quality of our results depend on the text retrieval system. Since we follow conjunctive semantics there could be some time points which are falsely identified as interesting time points. This can be avoided by treating movie titles are phrases instead of keywords.

Table 5.3: Interesting time points for movie “Forrest Gump”

Rank	Date	Event
1	11/01/1997	-
2	14/08/1994	-
3	28/03/1995	’Forrest Gump’ Triumphs With 6 Academy Awards
4	24/07/1994	"Forrest Gump" released
5	10/09/1995	-
6	15/02/1995	"Forrest Gump" 13 Oscar nominations
7	18/09/1994	-
8	07/10/1994	The flood of "Forrest Gump" items T-shirts, caps, books like "Gumpisms"
9	30/10/1994	Gump for politician campaign
10	23/10/1994	"The Wit and Wisdom of Forrest Gump" best seller

Table 5.4: Interesting time points for movie “Shrek”

Rank	Date	Event
1	14/05/2007	Shrek the third released
2	19/05/2001	First shrek movie released
3	16/05/2001	First shrek movie released follow up
4	18/05/2007	-
5	12/11/2001	Shrek dvd release
6	18/05/2004	Shrek declared most lucrative film of the year
7	28/11/2004	Shrek 2 released
8	24/05/2004	-
9	15/10/2004	-
10	23/07/2001	-

Table 5.5: Interesting time points for movie “Harry Potter”

Rank	Date	Event
1	18/11/2005	Harry Potter and Goblet of fire released
2	01/07/2003	Harry Potter and the Order of the Phoenix as NYT bestseller
3	16/07/2005	-
4	09/03/2005	"Harry Potter and the Order of the Phoenix," sold about five million copies
5	03/11/1999	-
6	11/07/2000	-
7	13/08/2000	-
8	09/11/2003	Harry Potter and the Order of the Phoenix in German, Hindi and millions of copy sold
9	25/10/1999	Best selling fiction book
10	28/03/2003	Harry Potter and the Goblet of Fire 2.8 million copies sold

Table 5.6: Interesting time points for movie “Shakespeare in love”

Rank	Date	Event
1	22/03/1999	Oscar for Shakespeare in love
2	13/12/1998	13 Oscar nominations for shakespear in love
3	25/03/1999	Oscar for Shakespeare in love
4	12/02/1999	Release of Shakespeare in love
5	13/12/1998	Release of Shakespeare in love followup
6	15/02/2005	-
7	15/02/1999	More Oscar nominations for shakespear in love
8	14/02/1999	-
9	09/03/1997	-
10	28/03/1999	-

Chapter 6

Conclusion and Future Directions

With constantly evolving web archives like the New York Times News archive or Wikipedia version history becoming increasingly available, there is a need for exploring their temporal dimension. The exploration of the temporal dimension along with the text helps in study of evolution of web archives, identifying interesting events and analyzing trends etc. In this direction, we defined interestingness of a time point as change in top-k results at any two consecutive time points. Since, the evolving web archives are huge, we focused on efficiency of interesting time points identification. We did an extensive study of data structures along with early termination technique to guarantee high performance. Depending on the data model, we proposed different efficient techniques to the problem we defined and proved their efficiency by applying these techniques to real world data collections like the New York Times Annotated Corpus [1] and the Wikipedia version history [2].

6.1 Discussion

We proposed an end-to-end system for efficiently identifying interesting time points from a web archive. The system consists of three layers. (i) *The text retrieval system* could be any search engine which can give query results in their relevance-score order along with meta data like publication time of the documents. (ii) *Interesting time points identification module* reads the results returned by text retrieval system incrementally in relevance score order and builds an index of the relevant documents not only to identify interesting time points but also to obtain top-k results at those time points. (iii) *Visualization module* presents the interesting time points identified in time axis which user can browse and obtain the results at those time points.

While we proposed an end-to-end system to identify interesting time points efficiently, we made the following contributions in this work.

1. **Defining Interestingness:** We define interestingness based on change in top-k relevant documents of consecutive time points. To the best of our knowledge we are the first to define interestingness using change in top-k.

2. **Rank-aware dynamic segment tree:** The enormous size of Wikipedia like versioned collection and the variable lifetimes of document versions make it challenging to efficiently identify interesting time points. In this thesis, we did an extensive study of data structures which can be used to address these challenges. In the process, we extended segment tree to make it rank-aware and dynamic, which can be used to efficiently and incrementally update interestingness of time points within the segment tree structure.

3. **Early termination technique:** For further improvement in efficiency of our approaches, we proposed an early termination technique inspired by top-k query processing algorithm NRA [12]. The early termination was facilitated by the fact that we can get an upper bound on the interestingness (based on our definition) values at any time point.

We did an extensive analysis of performance of our approaches. We proposed a naïve method which we use as a baseline to compare with. The experimental results show that our approaches outperform the naïve method both in terms of execution time and main memory usage. We started our solution approach with the key idea to maintain top-k results at every time point, and then use this to measure change in top-k and hence identify interesting time points. But with this approach we faced a challenge that not every time point has top-k results, which prompted us to touch all the documents in the result list, making it very expensive. To address this problem, we defined an early termination technique, based on NRA algorithm. Our efficient data structure along with early termination technique proved to be very effective in efficiently identifying interesting time points. In the experiments, we saw that for both New York Times [1] data set and Wikipedia version history [1] data set, even with millions of documents to process in the result list, and by looking at top-100 docuements at each time point we were able identify top-50 interesting time points in about 100ms on an average. Apart from performance evaluation, we also provided anecdotal evidence that the interesting time points we identified are indeed interesting to some extent.

6.2 Future Work

There are various directions for continuing this work, some of them are listed below.

1. **Richer queries**: In the current setting, we support only keyword queries with strict conjunctive semantics, in future we can extend the proposed system to support richer query types like phrase queries. Currently we use our own retrieval system, but we can replace it with more sophisticated retrieval systems like Lucene.

2. **Temporal References [22, 23]:** In this work we only look at the publication time of the documents or document versions. But it is possible that the documents which contain the keywords from the query may be referring to an event which occurred in the past. This information is ignored in the current setting. Each temporal reference spanning over a period of time can be represented as a line segment with a score associated to it. This model perfectly maps to our setting, so the proposed system in this thesis can be further extended to support temporal references. And since temporal references are often approximated time intervals (for example, the year 2000 refers to time period from 01/01/2000 to 31/12/2000), more research is required to define interestingness for such intervals.

3. **Granularity of time-points:** In this work we assumed a day level granularity of time, but in practice grouping the time points to a hierarchy of granularity helps the user. For example, if any four days of a single week are really interesting time points, we can just mark the whole week as an interesting week. This kind of grouping could be done at any granularity level.

4. **Diversity in interesting time-points:** In the current model of interestingness, it is possible that all the top-m interesting time points identified are in a single month or a year. On the other hand user might be interested in interesting time points from each year or from each month etc if any.

5. **Comparing interesting time-points:** It is possible that we can compare the interesting time points across queries. Given two or more queries, we need to define interestingness of time points in such a way that relative comparison of interestingness among them is possible.

List of Figures

List of Tables

Bibliography

[1] NYT. New York Times Annotated Corpus. http://corpus.nytimes.com/.

[2] WIKI. Wikipedia Version History. http://download.wikimedia.org/enwiki/.

[3] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715.

[4] Klaus Berberich, Srikanta Bedathur, Thomas Neumann, and Gerhard Weikum. A time machine for text search. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 519–526, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-597-7. doi: http://doi.acm.org/10.1145/1277741.1277831.

[5] Jon Kleinberg. Bursty and hierarchical structure in streams. *Data Min. Knowl. Discov.*, 7(4):373–397, 2003. ISSN 1384-5810. doi: http://dx.doi.org/10.1023/A:1024940629314.

[6] Gabriel Pui Cheong Fung, Jeffrey Xu Yu, Philip S. Yu, and Hongjun Lu. Parameter free bursty events detection in text streams. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 181–192. VLDB Endowment, 2005. ISBN 1-59593-154-6.

[7] Yunyue Zhu and Dennis Shasha. Efficient elastic burst detection in data streams. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 336–345, New York, NY, USA, 2003. ACM. ISBN 1-58113-737-0. doi: http://doi.acm.org/10.1145/956750.956789.

[8] Theodoros Lappas, Benjamin Arai, Manolis Platakis, Dimitrios Kotsakos, and Dimitrios Gunopulos. On burstiness-aware search for document sequences. In *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 477–486, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-495-9. doi: http://doi.acm.org/10.1145/1557019.1557075.

[9] Nilesh Bansal and Nick Koudas. Blogscope: a system for online analysis of high volume text streams. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1410–1413. VLDB Endowment, 2007. ISBN 978-1-59593-649-3.

[10] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, third edition, 2008. ISBN 3540779736, 9783540779735.

[11] Zoubin Ghahramani. An introduction to hidden markov models and bayesian networks. pages 9–42, 2002.

[12] Ronald Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99, 1999. ISSN 0022-0000. doi: http://dx.doi.org/10.1006/jcss.1998.1600.

[13] Rosie Jones and Fernando Diaz. Temporal profiles of queries. *ACM Trans. Inf. Syst.*, 25(3):14, 2007. ISSN 1046-8188. doi: http://doi.acm.org/10.1145/1247715.1247720.

[14] Hai Leong Chieu and Yoong Keok Lee. Query based event extraction along a timeline. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 425–432, New York, NY, USA, 2004. ACM. ISBN 1-58113-881-4. doi: http://doi.acm.org/10.1145/1008992.1009065.

[15] Ashwin Machanavajjhala, Erik Vee, Minos Garofalakis, and Jayavel Shanmugasundaram. Scalable ranked publish/subscribe. *Proc. VLDB Endow.*, 1(1):451–462, 2008. ISSN 2150-8097. doi: http://doi.acm.org/10.1145/1453856.1453906.

[16] Thomas H. Carmen, Leiserson E. Charles, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, second edition, 2001. ISBN 0-262-03293-7.

[17] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005. ISBN 0321321367.

[18] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Comparing top k lists. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 28–36, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics. ISBN 0-89871-538-5.

[19] Pablo Gamallo, Caroline Gasperin, Alexandre Agustini, and José Gabriel Pereira Lopes. Syntactic-based methods for measuring word similarity. In *TSD '01: Proceedings of the 4th International Conference on Text, Speech and Dialogue*, pages 116–125, London, UK, 2001. Springer-Verlag. ISBN 3-540-42557-8.

[20] IMDB-USA. All-Time Box Office USA. http://imdb.com/boxoffice/alltimegross.

[21] WIKI-EVENTS. Wikipedia Important Events. http://en.wikimedia.org/wiki/200[1-5].

[22] Omar Alonso, Michael Gertz, and Ricardo Baeza-Yates. On the value of temporal information in information retrieval. *SIGIR Forum*, 41(2):35–41, 2007. ISSN 0163-5840. doi: http://doi.acm.org/10.1145/1328964.1328968.

[23] Irem Arikan, Srikanta J. Bedathur, and Klaus Berberich. Time will tell: Leveraging temporal expressions in ir. In *WSDM (Late Breaking-Results)*, 2009.