An Effective Approach to Temporally Anchored Information Retrieval

Zheng Wei and Joseph JaJa
Department of Electrical and Computer Engineering
Institute for Advanced Computer Studies
University of Maryland, College Park
{zwei, joseph}@umiacs.umd.edu

ABSTRACT

We consider in this paper the information retrieval problem over a collection of time-evolving documents such that the search has to be carried out based on a query text and a temporal specification. A solution to this problem is critical for a number of emerging large scale applications involving archived collections of web contents, social network interactions, blog traffic, and information feeds. Given a collection of time-evolving documents, we develop an effective strategy to create inverted files and indexing structures such that a temporally anchored query can be processed fast using similar strategies as in the non-temporal case. The inverted files generated have exactly the same structure as those generated for the classical (non-temporal) case, and the size of the additional indexing structures is shown to be small. Well-known previous algorithms for constructing inverted files or for computing relevance can be extended to handle the temporal case. Moreover, we present high throughput, scalable parallel algorithms to build the inverted files with the additional indexing structures on multicore processors and clusters of multicore processors. We illustrate the effectiveness of our approach through experimental tests on a number of web archives, and include a comparison of space used by the indexing structures and postings lists and search time between our approach and the traditional approach that ignores the temporal information.

KEYWORDS: INFORMATION RETRIEVAL, TEMPROAL QUERY, TEMPORAL INDEXING, PARALLEL ALGORITHMS, CLUSTER

1. INTRODUCTION

A number of critical emerging applications require the organization and management of an extremely large number of time-evolving objects, such that these objects will need to be interactively explored through content based search within a temporal context. Consider for example the Congressional Web Archive, assembled and managed by the Library of Congress, which includes regular snapshots of selected news and government web sites. A temporally anchored query over such a collection could be used, for instance, to determine a politician's changing views over time regarding a particular issue. Today a large number of time-stamped collections of contents being generated or communicated through the internet are archived for later processing and analysis. Our main goal in this paper is to develop an effective approach for tackling such applications using and extending information retrieval technologies. More specifically, we develop an efficient strategy for building the inverted files, supplemented by new indexing structures, such that a temporal-based search can be carried out as efficiently as in the text-based search for non-temporal collections. The overhead required for building the extra indexing structures is shown to be minimal when compared to the traditional approach.

Information retrieval technologies have not traditionally dealt with the temporal dimension. For example, major web search engines such as Google, Yahoo, and Bing support queries on the web pages that are currently available on the web. Their basic approach is not tailored to handle temporally-anchored searches such as locating the most important news about Barack Obama before year 2005. While the internet is by far the largest library that ever existed, its contents are always changing, and hence old valuable contents can be lost forever if not archived and preserved properly. Major library and archiving organizations are constantly crawling the web to capture and archive particular contents of interest. Such contents constitute a collection of time-evolving documents; each document corresponds to a specific uniform resource identifier (URI) with its corresponding document versions time stamped by the crawl times. Such organizations include the Internet Archive [1], the Library of Congress [2], UK Web Archiving Consortium [3], the National Library of Australia [4] and the California Digital Library [5]. In particular, over 5.8 petabytes of web data have been captured by the Internet Archive Petabox project as of December 2010 [6], which continues to grow at a rate of more than 100 terabytes per month since 2009 [1]. The Internet Archive uses its WayBack machine to search such contents where the user supplies a URL and the engine responds with a chronological list of the dates when the corresponding page was crawled. Other much more dynamic contents produced through social networks, news and various types of information feeds, and blogs are also captured on a regular basis and archived with temporal information. Our work includes the temporal dimension as an integral component of the strategy to build the inverted files so that queries containing text and a time specification can be handled efficiently.

The rest of the paper is organized as follows. In the next section, we provide a brief background and a summary of the previous related work on temporally anchored information retrieval. In Section 3 we formally define the problem and introduce the new indexing structures needed for managing the temporal dimension. In Section 4, the query performance is examined using our new indexing structures. Very high-throughput parallel algorithms for constructing the temporal inverted files are developed in Section 5 and the corresponding experimental results are presenting in Section 6. We conclude in Section 7.

2. BACKGROUND AND PREVIOUS RELATED WORK

We assume that each document, which refers to a time-evolving object in this paper, is uniquely identified by a global identifier such as a URI. In this case, a document may have many versions over time, each of which includes a time stamp that indicates the start time of the document version. Only one version of a document can exist at any time. The start time could be the time when the document version was created or the time when the document was first seen. For example, in web archiving, a time stamp is assigned whenever the corresponding web page is crawled, and the versions correspond to the crawled distinct pages all of which belong to the same

URL We allow a document to cease to exist either for a while (for example, a web page that is unreachable during several consecutive crawls but reachable afterwards) or permanently.

A considerable amount of research has been conducted to deal with various aspects related to the construction of inverted files and related search strategies. Here we focus our review on the work related to the temporal case.

Previous work for the temporal case has been limited. A temporal document database first developed in [7] introduced a solution with a hybrid of document name index, version database, and text index. The document name index maps a document identifier to its list of global Version IDs since a document may have several versions. The version database assigns global Version ID to versions of documents according to the start time, but two contiguous global Version IDs may refer to two versions of different documents. The list of document versions that a term appears in can be found in the text index (or the inverted files in our definition). In [8], the global Version ID is replaced by <Document ID, Document Version ID> tuple, thereby if a term appears in several consecutive versions of a document, its postings could be shrunk into <Document ID, Document Version ID range> tuples for space efficiency. A Time Index+ is further incorporated for frequent terms for large document databases [9]. Since the target is a database for versioned documents, a temporal query has very high cost using the text index since the whole postings list has to be scanned and, moreover, critical elements such as term frequency or term positions in a document are not available for scoring. Berberich et. al. [10] extended the traditional postings to <Document ID, score, time_start, time_end>. Postings with the same Document ID and similar scores are merged to reduce the space requirement thereby employing a lossy compression strategy. To reduce search cost, postings lists are split into sublists that are valid time intervals so that only the sublists valid in the time range in the query will be traversed. Based on this, Anand et. al. [11] further proposed the sharding approach for efficient temporal inverted files in both space and query processing time. However, both [10] and [11] work with stored datasets in batch mode and it is unclear how to extend this approach to incrementally update the inverted files. In [12] an approach is proposed based on temporal partitioning of the collection into time windows. An analytical model is used to determine an optimal size of the time windows based on some assumptions on the input distribution.

3. Problem Definition and Temporal Strategy

3.1 Problem Definition

Following standard information retrieval terminology, we refer to the objects in our collection as documents, each document is uniquely identified by a global identifier such as a URI. Each document evolves over time and hence many versions of the document may exist. A document version $Doc_{i,j}$ denotes the j^{th} version of the document identified by URI_{i} , which was first created or detected at time $t_{i,j}$ and hence can be represented by:

$$\{\boldsymbol{URI_i}, \boldsymbol{t_{i,j}}, < \boldsymbol{T_1^{ij}}, \boldsymbol{T_2^{ij}}, \dots\dots, \boldsymbol{T_{N_{ij}}^{ij}} >\}$$

where T^{ij} is a term that occurs in $Doc_{i,j}$ and $N_{i,j}$ is the total number of terms in $Doc_{i,j}$. By definition, the *start time* of a document version is equal to $t_{i,j}$, but the *end time* is not defined until a new version $Doc_{i,j+1}$ occurs in which case it is equal to $t_{i,j+1,}$ or the document ceases to exist in which case it is equal to the time when this occurs. The lifetime of the document version $Doc_{i,j}$ is then defined as the time interval between the start time and the end time.

A query in the temporal case contains a set of query terms $\{Q_k\}$, possibly connected by Boolean operators, and a query time t_Q. A document version $Doc_{i,j}$ matches the query if it contains some or all the terms in $\{Q_k\}$ (depending on the query Boolean operator) and is alive at t_Q, that is, t_Q lies in the time interval between the start time and the end time of $Doc_{i,j}$ The result of the search is a ranked set of document versions based on a scoring function used to determine similarity scores between the query and the document versions matching the query. Note that the computation of the scoring function typically requires some global statistics at t_Q such as the number and average length of live document versions and the number of live document versions containing Q_k (i.e. document frequency of Q_k). In this paper, we will use the well-known Okapi BM25 function to compute the scores. However, it should be easy to incorporate other scoring functions into our strategy.

As in the traditional case, we assign a monotonically increasing global Version ID (VID) to each document version and incorporate this VID into the postings lists to identify the appropriate document version in our collection. From now on, we use the notation $VID_{i,j}$ to uniquely identify the document version $Doc_{i,j}$. Note that we are assuming that our collection consists of document versions over discrete time steps called *elementary time steps*. An elementary time step captures the time granularity of the document collection as well as the granularity at which queries can be answered. For example, the Wikipedia collections, to be described later, are monthly snapshots of Wikipedia generated at the end of a month. Hence in this case, an elementary time step is a month which constrains t_Q to be within a month such as February 2004. Within an elementary time step, there is at most one document version per URI and the global statistics remain the same within that time step. All time values defining start and end times correspond to elementary time steps. As a result, if document versions are processed in the chronological order of increasing elementary time steps, it is guaranteed by our VID assignment strategy that the start time of a higher VID is later than or equal to the start time of a lower VID.

3.2 Overall Strategy for Temporal Indexing and Searching

Our overall strategy is similar to the traditional strategy of using postings lists: each posting consists of the VID and the term frequency within that document version, sorted according to the VIDs on the list. Other information can be incorporated into the postings as needed. However to carry out temporal searching effectively, we introduce two new indexing structures, in addition to the usual global dictionary structure. The new data structures are two temporal tables defined as follows:

VID table, which maps a VID to its unique URI, start and end times, document length and possibly other document version information (such as location on disk of the document version). Note that a similar data structure is typically built for the non-temporal case, except that it does not include the start and end times.

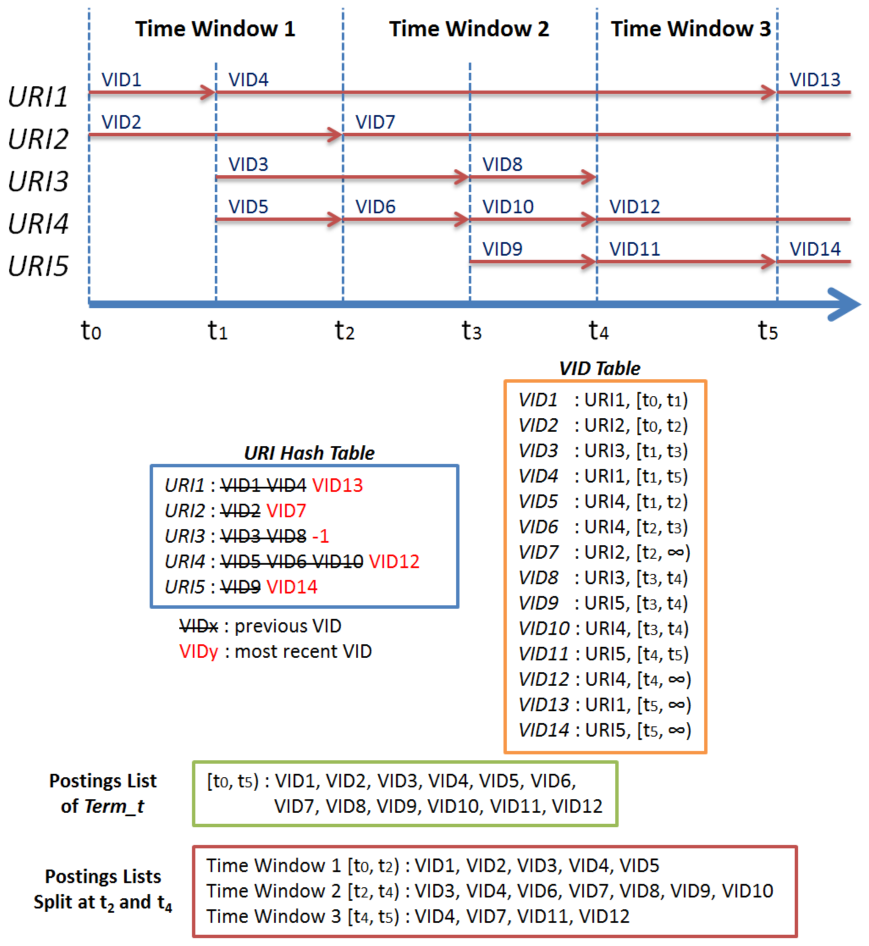
URI hash table, which is used during the construction of the VID table, will map each URI to its most recently seen VID as the document versions are processed. Another possibility is to allow the hash of a URI to include the VIDs of all the corresponding document versions encountered so far, which can later be used to quickly look for all the document versions of a particular document. However in this paper, the hash of a URI contains only a single VID or is equal to -1 indicating that the URI has ceased to exist at this stage during the processing of the document versions to build the VID table and the postings lists.

Accordingly the format of a posting changes from the traditional <*Document ID*, *term_freq*> to <*VID*, *term_freq*>. Note that, in the work reported in [10, 11], each posting includes the life time of a document such as <Document ID, time_start, time_end, term_freq>. Such an approach returns the lifetime of a document version directly from the postings list at a substantial space cost since the same start and end times have to be duplicated in the postings lists for all the terms that appear in this document version. Moreover, using our approach, traditional schemes can be easily extended to generate the new postings lists such that the postings are sorted in ascending order of VID, and hence can be easily compressed.

Let us now consider how a query with a query time t_Q can be handled using our indexing structures and the postings lists. A straightforward approach would be to start by retrieving all the matching VIDs from the postings lists of the query terms, and then determine the temporal validity of each VID by looking up its lifetime using the VID table. This strategy can be easily implemented. However, as mentioned in [10, 12], the postings lists can grow very large, which can introduce a very high overhead when determining the valid VIDs matching the time constraint of the query. To tackle this problem, the authors of [10, 12] propose an approach based on splitting each postings list into many segments after the postings lists are constructed.

Our approach will be very different. As we are building the temporal inverted files, we will construct different segments on the fly such that each segment belongs to a certain time span determined dynamically, and each time span is the same for all the postings lists. We refer to the time span covered by a segment as a *time window*, which is somewhat similar to the time window notion introduced in [12]. A time window [*TW_start*, *TW_end*) consists of one or more elementary time steps such that each elementary time step belongs to a time window and no two time windows overlap. A document version VID_{ij} is considered alive in this time window if there is an overlap between $[t_{i,j}, t_{i,j+1})$ and [*TW_start*, *TW_end*], i.e. VID_{ij} is alive for some time within this time window.

In our approach, each time window has its separate postings lists (segments) and VID table corresponding to all the document versions that are alive within this time window. Query can now be localized to a single time window, which substantially reduces the search time. Moreover, our approach has the added advantage of incorporating additional document versions incrementally under a new time window. However, this comes at the cost of storing some duplicate postings in consecutive time windows due to the fact that some document versions will be alive across a time boundary. This implies that the postings of a document version will be replicated through all the time windows in which this document version stays alive. Therefore a long-lived document version may appear in many time windows and hence will generate multiple copies of the same postings in the output. However as we will see later, this overhead is relatively small overall if the time windows are selected carefully, and the search time is dramatically improved.



URI Hash Table

URI1 : ~~VID1 VID4~~ VID13
URI2 : ~~VID2~~ VID7
URI3 : ~~VID3 VID8~~ -1
URI4 : ~~VID5 VID6 VID10~~ VID12
URI5 : ~~VID9~~ VID14

~~VIDx~~ : previous VID
VIDy : most recent VID

VID Table

VID1	: URI1, [t0, t1)
VID2	: URI2, [t0, t2)
VID3	: URI3, [t1, t3)
VID4	: URI1, [t1, t5)
VID5	: URI4, [t1, t2)
VID6	: URI4, [t2, t3)
VID7	: URI2, [t2, ∞)
VID8	: URI3, [t3, t4)
VID9	: URI5, [t3, t4)
VID10	: URI4, [t3, t4)
VID11	: URI5, [t4, t5)
VID12	: URI4, [t4, ∞)
VID13	: URI1, [t5, ∞)
VID14	: URI5, [t5, ∞)

Postings List of *Term_t*
[t0, t5) : VID1, VID2, VID3, VID4, VID5, VID6, VID7, VID8, VID9, VID10, VID11, VID12

Postings Lists Split at t_2 and t_4
Time Window 1 [t0, t2) : VID1, VID2, VID3, VID4, VID5
Time Window 2 [t2, t4) : VID3, VID4, VID6, VID7, VID8, VID9, VID10
Time Window 3 [t4, t5) : VID4, VID7, VID11, VID12

Figure 3.1. An Example of Temporal Indexing

To fix the ideas, let us consider a small example. A collection with five documents is illustrated in Figure 3.1, where all documents contain term *Term_t* and the time stamp ∞ indicates that the document is still alive. Times t_2 and t_4 are the starting points of time windows. Note that document version *VID4* is alive in the three time windows shown, and hence it has a footprint in the postings lists in all three segments. Note also that the URI hash table shown corresponds to the hash table at the end of the indexing process.

At one extreme, we can set each time window to be the same as an elementary time step, and hence a search will be immediately localized to the corresponding postings lists with no need to check the time validity of the postings, and hence the search can be carried out quite fast (best possible search time). However this will come at an extremely large space cost for the output files. The other extreme is to have a single time window covering all the elementary time steps, which will correspond to the smallest possible output file, but with generally unacceptable search time through very large postings lists. Therefore we need to strike a compromise between these two extremes and try to reach a balance between the query response time and the total size of the postings lists.

We consider three possible strategies for determining the appropriate window size.

***Even-Time**, which splits the document collection into several time windows such that every time window covers the same length of time period. This will make it straightforward to locate the appropriate postings lists and VID tables for any query time during search.*

***Even-Size**, which monitors the total uncompressed size of input files such that a new time window is started when the collection size managed under the current time window exceeds a threshold. Such a strategy may yield to similar size postings lists under the different time windows.*

***Output-Oriented**, which forces an upper bound on the size of the output postings lists. Hence a new time window is started when the total size of the postings lists or average size of the postings lists under the current time window exceeds a preset threshold.*

The first strategy was used in [12]. For example, the 36-month Congressional document collection (to be introduced in Table 4.1) was divided into three 12-month time windows. The second strategy attempts to ensure that the collection size processed during each time window is about the same, and hence the output postings lists from different time windows are expected to be of similar size. Note, however, that the final postings lists for a time window also include those inherited from earlier time windows and hence even with the second strategy, the total output size of the postings lists under a time window is only weakly controlled. The third strategy presents the best option, but its implementation is the most complex.

3.3 Query Strategy

In this section, we take a look at the temporal query strategy details. Given all the necessary temporal inverted files residing on disks, we can handle a temporal query with query terms $\{\boldsymbol{Q_k}\}$ and query time $\boldsymbol{t_Q}$ by executing the following steps:

***Step 1:** Initialization of Search engine*

***Step 2:** For each temporal query* <$\boldsymbol{Q_k}$, $\boldsymbol{t_Q}$>, do:

> ***Step 2.1**: Determine the corresponding time window TW and retrieve the average document length at* $\boldsymbol{t_Q}$;
> ***Step 2.2**: Parse the query string into individual query terms;*
> ***Step 2.3**: Search all the query terms in the dictionary to locate their postings lists;*
> ***Step 2.4**: Fetch data of all query terms from disks*

Step 2.4.1*: Fetch the document frequency of each query term at* $\boldsymbol{t_Q}$*;*
Step 2.4.2*: Fetch and decompress the postings list in TW for each query term;*

Step 2.5*: Merge the postings lists to find all matching documents and compute scores;*

Step 2.5.1*: Merge the postings lists of all query terms to find out common VIDs;*
Step 2.5.2*: Check the validity of VIDs at* $\boldsymbol{t_Q}$*;*
Step 2.5.3*: Fetch the document lengths of the valid VIDs from the VID table;*
Step 2.5.4*: Compute the scores for the valid VIDs;*

Step 2.6*: Find the top K document versions with highest scores;*

In Step 1, we load the dictionary from disk into memory, the time span of each time window, and elementary time steps (ETS) such that, given a query time $\boldsymbol{t_Q}$, we can quickly find which time window and the ETS this $\boldsymbol{t_Q}$ falls into. We also assume that the VID table is pre-loaded into memory in this step.

Steps 2.1, 2.4.1 and 2.5.2 are unique to the temporal query scenario. Step 2.1 is trivial in general because the total number of elementary time steps is typically not so large. For example, 1M elementary time steps correspond to over 20 years of hourly snapshots. Steps 2.5.2 and 2.5.3 involve accessing the VID table but as we explained in the last paragraph that the document length and file location still has to be obtained from some data structure in the non-temporal case and hence we consider these two steps as not introducing any extra cost relative to the non-temporal case. The overhead of Step 2.4.1 will be fully analyzed later.

We use the well-known Okapi BM25 function [13] to compute the scores in Step 2.5.4:

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i,D)\cdot(k_1+1)}{f(q_i,D)+k_1\cdot(1-b+b\cdot\frac{|D|}{\text{avgdl}})},$$

$$\text{IDF}(q_i) = \log\frac{N-n(q_i)+0.5}{n(q_i)+0.5},$$

where N is the total number of documents, $n(q_i)$ is the number of documents with term q_i, $f(q_i, D)$ is the term frequency in D, $|D|$ is the length of D, *avgdl* is the average document length in the collection, k_1 and b are two parameters, which we set to $k_1 = 2.0$ and $b = 0.75$.

No approximation or lossy compression is used in our algorithm, and hence the scores and ranking results of the temporal query reflect the exact scores given the collection of documents that are live at time t_Q.

4. EXPERIMENTAL EVALUATION OF OUR STRATEGY

We use two significant collections that exhibit different characteristics of our strategy. The first collection is the Congressional dataset from the Library of Congress, which includes 36 monthly snapshots of selected news and government websites crawled between January 2004 and December 2006. The second dataset is the Wikipedia01-07 data, which is derived from a publicly available XML dump of Wikipedia articles created on January 3, 2008 with 83 monthly snapshots between February 2001 and December 2007. The overall characteristics of the two benchmarks are given in Table 4.1. The number of terms and tokens may vary with different implementations due to the choice of tokenization and stemming procedures.

The generated output, dictionary, the two temporal tables and postings lists, are written onto local disks.

Table 4.1. Statistics of Document Collections

	Congressional	**Wikipedia 01-07**
Compressed Size	314GB	29GB
Uncompressed Size	1,672GB	79GB
Crawl Time	01/04 to 12/06	02/01 to 12/07
Document Number	28,731,237	2,077,745
Document Version Number	80,092,737	16,618,497
Average Versions per Document	2.788	7.998
Number of Terms	12,229,793	9,404,723
Number of Tokens	41,356,310,380	9,375,229,726

To evaluate our approach, we estimate the space overhead due to the introduction of the new indexing structures, the URI hash table and the VID table, and then the query response time relative to the time windowing strategy used. We note that we will show later in Section 6 that the processing time to create the inverted files with the extra indexing structures is very close to that of the traditional strategy for processing the same collection while ignoring the temporal information.

We start by taking a close look at our implementation of the new indexing structures.

4.1 URI Hash Table and VID Table: Implementation Details

At any time during the initial processing of the document collection, the URI hash table is used to map a URI into the most recent VID of the document version encountered thus far. If multiple URIs hash to the same value, then a singly linked list is created to store these URIs in a sorted order (according to URI strings) for fast insertion of a new URI.

On the other hand, the VID table stores, for each VID, specific information pertaining to that document version such as: start time *t_start*, end time *t_end*, document length, pointer to the corresponding URI string, and the location of this document version within the underlying file system. In the non-temporal case, a similar data structure is typically constructed as well to fetch this type of information pertaining to a document except for the temporal information. A significant difference between the two cases is that during the indexing process an entry of the VID table will be written twice (when the document version is first encountered and later when the end time is determined) and will be read multiple times as we will see later in this section. On the other hand, in the non-temporal case, each such entry is completely built during its creation and will not be read or written any more during the indexing process.

When a document version is processed, given its {$\boldsymbol{URI_i}$, $\boldsymbol{VID_{ij}}$, $\boldsymbol{t_{ij}}$, ***Doc_Length***}, the URI hash table is first visited to fetch the most recent VID corresponding to this URI, that is, $\boldsymbol{VID_{i,j-1}}$. In the VID table, the ***t_end*** of $\boldsymbol{VID_{i,j-1}}$ can now be set to $\boldsymbol{t_{ij}}$. Then, $\boldsymbol{VID_{ij}}$ is inserted in the URI hash table indicating that this is now the most recent version. The VID table will be updated with the rest of the information regarding $\boldsymbol{VID_{ij}}$ except ***t_end,*** which cannot be determined until a newer one $\boldsymbol{VID_{i,j+1}}$ is encountered or the document ceases to exist.

As we process the collection of documents, we will only need the portion of the VID table whose entries are still alive, that is, the entries whose end times have not been yet determined. Hence we will attempt to keep that portion in memory, while most of the rest is stored on disk. To accomplish this goal, we note that most of the lower VIDs are not likely to be alive while the most recently assigned higher VIDs are still alive. Therefore we maintain a small VID table in memory consisting of two parts: **Part I** contains the consecutive VIDs for higher VIDs and **Part II** contains nonconsecutive VIDs for lower VIDs. All the rest of the entries are not alive and are stored on disk. We construct and maintain Part I and Part II as follows.

The Part I (high VIDs) can be indexed directly by the VID so that the lookup time is very small, but it will grow over time and many of the VIDs may no longer be alive. On the other hand, Part II is stored in a more compact fashion in such a way that binary search can be used to look up an entry corresponding to a VID. In our current implementation, if the size of Part I exceeds 8M after 100GB of input data are processed or at the end of a time window), only the highest (also the most recent) 8M VID entries are kept in Part I and the remaining entries are migrated to Part I,I followed by a complete scan on Part II so as to transfer the non-live entries onto disks (and hence the size of Part II will shrink considerably). These parameters (8M and 100GB) have been determined empirically and seem to yield the best performance on our platform.

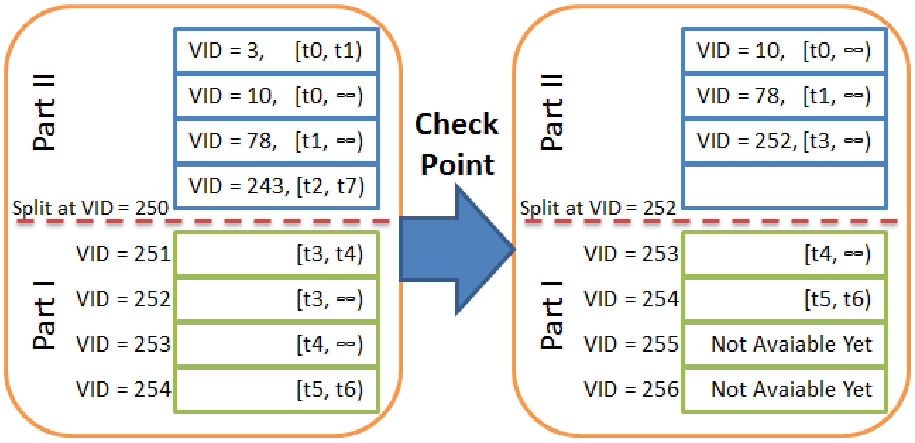


Figure 4.1. Example of the VID Table Data Structure

A trivial example of the VID table is shown in Figure 4.1. In Part I, VIDs are implicit and can be calculated by the sum of an offset and table index; on the other hand, part II stores the VIDs inside the table memory space explicitly in sorted order. After the reorganization step, both Part I and Part II may appear as shown in Figure 4.1.

We will illustrate next the savings achieved by using such a strategy.

4.2 Growth of URI Hash Table and VID Table

We take a closer look at how the dynamic strategy used to manage the VID table contributes to saving space. The VID tables in memory are checked periodically in such a way that invalid VIDs (that is, VIDs that are not live) are written onto disks and some VIDs are migrated from the VID table Part I to Part II. As a result, after each reorganization the size of the VID table Part II increases and the size of the VID table Part I shrinks back to 8M entries as illustrated in the graph shown in Figure 4.2. On the other hand, the number of VIDs stored in the URI hash table grows steadily over time. The top line in Figure 4.2 shows the total number of assigned VIDs, which also reflects the number of entries in the entire VID table in the temporal case (which is also the same as the corresponding data structure for the non-temporal case given the same collection). The difference between this top line and the second top line indicates the memory savings achieved by using our dynamic strategy for managing the VID table.

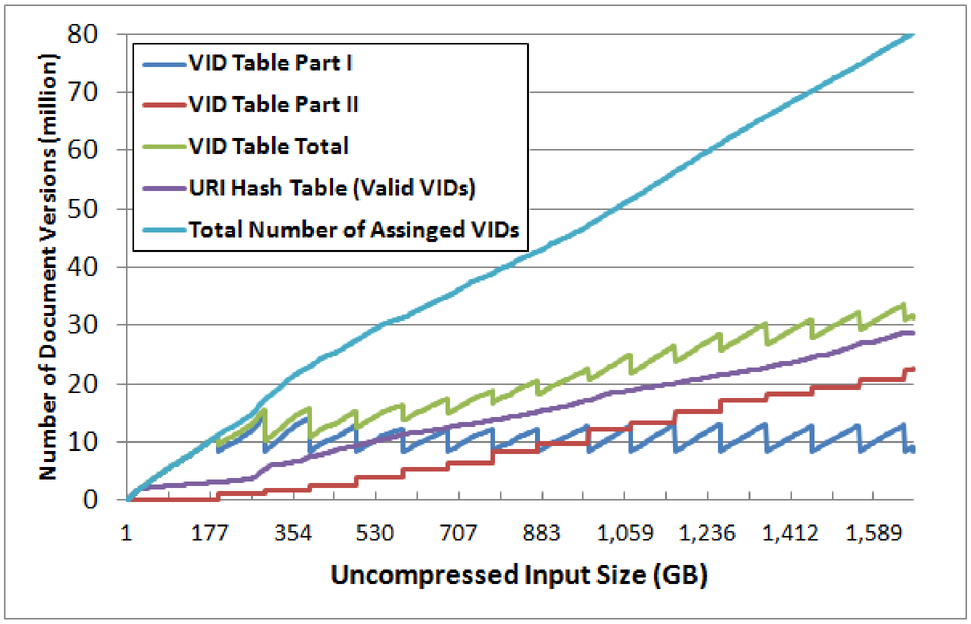


Figure 4.2. Growth of URI Hash Table and VID Table

4.3 Relationship between Query Performance and Time Windows

In this section, we explore the effects of different strategies for selecting time windows on the sizes of the postings lists and query performance. We will also compare these strategies to the traditional strategy when the temporal component of the document collection is ignored. We use the Even-Size strategy to evaluate the impact of the number of time windows on the size of the postings lists.

4.3.1 Size of Postings Lists Relative to the Number of Time Windows

Table 4.2. Output Size over the number of Time Windows

Number of Time Window	Compressed Output PL Size (GB)	Total Number of Postings
1	33.7	27,591,138,345
3	42.0	35,514,010,829
9	88.6	71,112,310,390
18	136.6	110,371,532,702
36	248.1	199,463,515,717

We start by considering the compressed output size as a function of the number of windows for the Library of Congress Congressional dataset. For the extreme case of a single time window, the generated postings lists are exactly the same as those that would be generated had we indexed the document collection without paying attention to the temporal information. The output size is minimal in this case since no duplicate documents have to be considered. As the number of time windows increases, we expect the output size to increase since the number of duplicate document versions across the time windows is expected to increase. This is illustrated in the next table. Note that we only have 36 monthly snapshots, i.e. 36 elementary time steps, for this collection. The compressed output size of postings lists increases about 7 times when the number of time window grows from 1 to 36, which is comparable to the results in [10] where the number of postings increases from the original 8 million to over 54 million in the performance optimal scenario. Note also that the average lifetime of a document version is about 11.2 months in this collection.

4.3.2 Detailed Query Processing Time

We now take a close look at the detailed query processing time. To conduct our tests, a thousand queries are tested, each query having a random query time between 01/2004 and 12/2006, with three terms chosen randomly from the dictionary. The search program runs on a CPU (Intel X5560) with a single thread such that the one thousand queries are processed sequentially. A commodity 2.5 inch 5400RPM hard drive is used to store the inverted files which are not preloaded into the memory by any caching strategy. We filter out the 10% slowest query batches and calculate the average of the remaining 90% data. Such filtering is carried out due to two reasons: (1) queries that consist of very popular terms map onto very long postings lists resulting in millions of matching results, which will significantly slow down the search; (2) the first few queries are generally slower because of the empty caches (including CPU cache, virtual memory and disk cache) but will later be filled up resulting typically in accelerated memory and disk accesses. The same approach is used to represent the query processing time in this paper.

We use the Even-Size strategy to split the entire collection into three time windows. Table 4.3 lists the average processing times from Step 2.1 through Step 2.6 as well as the overall time of Step 2 of the search algorithm presented in Section 3.3. We note that the most time consuming part is Step 2.4, in which the postings lists and the document frequencies of the query terms are read into memory. As we have mentioned in Section 3.3, the extra cost to processing a temporal query lies in the execution of Step 2.4.1, which is about 105ms for three query terms or 35ms per query term, which is close to the 25ms random disk seek time that was observed through our tests. Over 98% of the time is spent in executing Steps 2.4 and 2.5, and hence in the next sections we focus on these two steps.

Table 4.3. Query Process Time in Step 2

Step	**Time (millisecond)**
Step 2.1 Find ETS and TW	0.018
Step 2.2 Query Parse	0.010
Step 2.3 Search Dictionary	0.073
Step 2.4 Fetch Doc_Freq and PL	648.052
Step 2.4.1 Fetch Term Doc_Freq	105.905
Step 2.4.2 Fetch PL	540.517
Step 2.5 Merge and Score	80.539
Step 2.6 Display Top 100 Results	10.036
Total	741.894

4.3.3 Query Performance Relative to the Number of Time Windows

The main reason for introducing multiple time windows is to reduce query response time by localizing the search into a small time frame that generates relatively short postings lists. We expect the number of time windows to primarily influence the execution of Steps 2.4 and 2.5 since these steps process the postings lists corresponding to the query terms. The size of the VID table in each time window as the number of time windows increases, which makes it quite possible for the local VID table to fit into the CPU cache and thereby accelerate the execution of Steps 2.5.2 and 2.5.3.

Table 4.4. Query Performance as a Function of the Number of Time Windows

Number of Time Windows	Step 2.4 Fetch Doc_Freq and PL (millisecond)	Step 2.5 Merge and Score (millisecond)	Average Size of Compressed PLs for One Time Window (GB)	Average Length of PL per Term for One Time Window
1	2383.81	126.91	33.7	2256
3	648.05	80.54	14.0	968
9	372.66	28.07	9.8	646
18	310.21	21.79	7.6	501
36	300.64	21.35	6.9	453

Table 4.4 shows the execution times of Steps 2.4 and 2.5 corresponding to the number of time windows using the Even-Size strategy. When the number of time windows is larger than 9, very limited performance gain is obtained beyond that value because the average size of postings lists for a time window decreases very slowly with more than 9 or more time windows. The size of postings lists from newly arrived document versions is roughly proportional to the length of the current time window. Hence using many short time windows results in relatively few newly generated postings within the time window with many that will be passed from the previous time window. Consequently, the average size of postings lists in a time window decreases slowly with a large number of time windows. On the other hand, the total size of postings lists from all time windows grows quickly as a function of the number of time windows (from 9 to 36) with limited improvement on query performance. Therefore, based on these empirical results, choosing three or nine time windows achieves an optimal balance between space and performance for the Congressional dataset.

4.3.4 Summary

Figure 4.3 shows the relationship between the overall query response time, the sizes of postings lists, and the size of the additional temporal indexes as a function of the number of time windows, assuming the Even-Size strategy. Comparing the non-temporal case (i.e., indexing the collection without taking the temporal information into consideration) and the temporal case with a single time window, we can see that only a small increase in query response time is introduced to handle temporally anchored queries. Therefore, the query performance does not suffer due to the additional temporal indexing structures even though we can now handle temporal queries. In fact it can be improved using a temporal partition of the collection (in our case 3 or 9 time windows). In addition, our indexing can incrementally process additional document versions and easily incorporate the new document versions into our existing indexing structures.

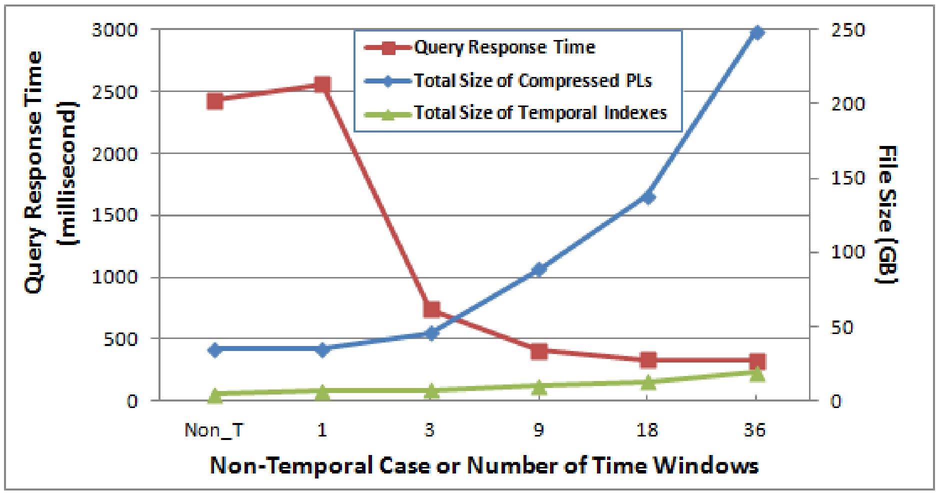


Figure 4.3. Impact of Time Window Numbers on Index Size and Query Performance

Table 4.5. Comparison of Additional Tables between Non-Temporal and Temporal Cases

Number of Time Windows	Size of Doc_ID Table in the Non-Temporal Case	Size of VID Table in the Temporal Case
Non-Temporal	1.19	-
1	-	1.81
3	-	2.59
9	-	5.02
18	-	7.92
36	-	14.52

4.4.4 Relationship Between Query Performance and Different Time Window Strategies

In this section, we compare the query performance under the two distinct strategies introduced earlier, the Even-Size strategy and the Even-Time strategy. By the Even-Size strategy, the resultant three time windows for the Congressional dataset cover respectively 19 months (01/2004 to 07/2005), 10 months (08/2005 to 05/2006) and 7 months (06/2006 to 12/2006), and each time window consists of about 560GB of input. On the other hand, the Even-Time strategy splits the 36 monthly snapshots into three time windows, each of which covers 12 months, consisting of 235GB, 638GB, 796GB of input data respectively. Table 4.6 shows the sizes of the generated postings lists after compression. Under the Even-Size strategy, the size of the postings lists grows slowly from one time window to the next (due to the postings lists inherited from the previous time window). This is not the case for the Even-Time strategy as the sizes of the postings lists increase significantly from one time window to the next.

Table 4.6. Size of Output Size with Different Time Window Splitting Strategies

Time Window	Compressed Size of Output PLs (GB)	
	Even-Size	Even-Time
1	12.6	5.4
2	13.9	13.2
3	15.5	22.3

As for the query performance, the average query response time remains almost the same in either strategy if we randomly choose a query time between 01/2004 and 12/2006. However, if we take a deeper look and capture the average time required to process a query during Steps 2.4 and 2.5 with query time in years 2004, 2005 or 2006 separately, some interesting but expected results emerge as illustrated in Table 4.7. Since the size of output postings lists grows quickly from 2004 to 2006 with the Even-Time strategy, the resultant execution time of Step 2.4 and Step 2.5 also increase at a similar rate. This leads to a query response time that fluctuates drastically. In practice, this trend is likely to get worse if we assume that users will be more interested in recent document versions and more temporal queries fall into year 2006. On the other hand, the Even-Size strategy yields much more stable results for both Steps 2.4 and 2.5. The time cost is still growing but at a much smaller rate.

Table 4.7. Query Performance with Different Time Window Splitting Strategies

Query Time Range	Step 2.4 Fetch Doc_Freq and PL (millisecond)		Step 2.5 Merge and Score (millisecond)	
	Even-Size	Even-Time	Even-Size	Even-Time
2004	580.59	336.12	76.92	37.37
2005	614.21	655.24	81.39	76.02
2006	748.95	1124.66	84.30	99.78

5. Parallel Algorithms for Constructing the Inverted Files in the Temporal Case

In this section, we present high-throughput parallel algorithms for single multicore processors and for clusters of such processors to build the inverted files for the temporal case. In addition to generating the postings for each time window, our algorithms will construct a global dictionary, the URI hash table, and the VID tables, and will incrementally process additional document versions as they are introduced under a new time window. Our starting point will be the pipelined algorithms presented in [14, 15], which will be extended to build the additional indexing structures and to manage postings lists that survive from a time window to the next. The resulting throughput is comparable to the non-temporal case achieving scalability up to the largest cluster we have access to.

5.1 Algorithm On A Single Multicore Node

5.1.1 Overall Approach

The pipelined algorithm presented in [14, 15] consists of a parsing stage during which each of a number of parsers reads a fixed size block (roughly 1GB) from the disk containing the documents, executes the parsing algorithm, and then writes the parsed results onto a buffer. A number of indexers pull the parsed results from the buffer as soon as they are available and jointly construct the postings lists, which are written onto the disk as soon as they are generated. The details can found in [14, 15]. In the temporal case, the parsers also pass the URI and the start date for each parsed document version to the indexers. During the indexing stage, a special thread pulls the parsed results from the buffer as soon as they are available, and builds the URI hash table (one entry per document) and the VID table (one entry per document version), after which a number of indexers jointly update (or construct during the first time window) the dictionary and generate the corresponding postings lists. The extra workload to build the URI hash table and VID table is trivial compared to the work performed by the parallel indexers to build the dictionary and postings lists.

As in the dictionary case, the URI hash table and the VID table remain in main memory until the entire data collection has been processed; however the two temporal tables are checked periodically such that only a small portion of the VID entries are kept in main memory as described in Section 4.1; the postings lists are first kept in memory and then written into a disk as temporary files before the memory gets full. When the current time window is closed, a number of parallel *temporal mergers* read these temporary files and the postings lists inherited from previous time window from the disk, and generate the postings lists for both current time window and those that are sill alive during the next time window. The number of parsers, indexers and temporal mergers are determined depending on the physical resources available as will be illustrated in Section 6.1. The overall scheme is shown in Figure 5.1.

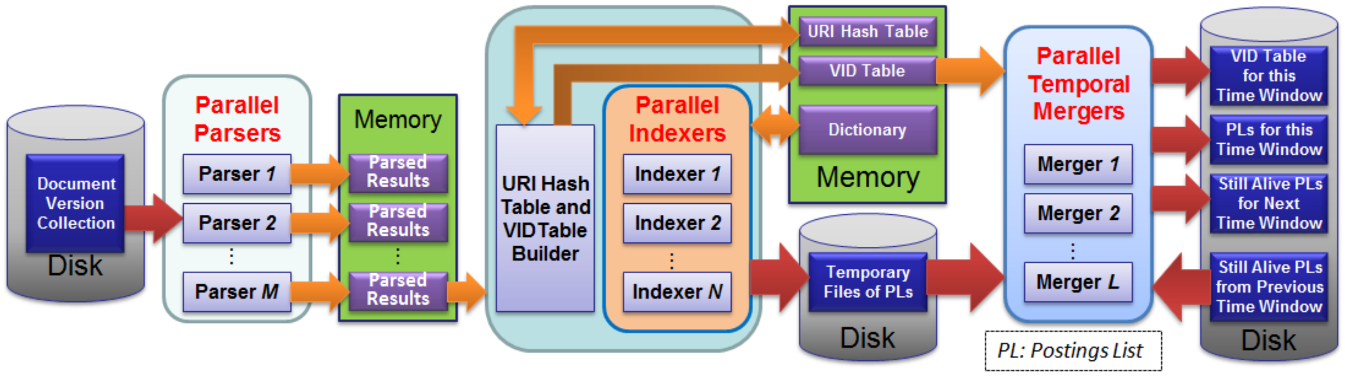


Figure 5.1. Overall Data Flow on a Single Node

5.1.2 Highlights of the Parsing and Indexing Stages

A key element of our pipelined strategy introduced in [14, 15] is a hybrid data structure to represent the dictionary, which consists of a trie at the top level and a B-Tree attached to each of the leaves of the trie as shown in Figure 5.2. Essentially, terms are mapped into different groups, called trie-collections, each of which is then represented by a B-tree. This data structure is critical to our parallel algorithms since each trie-collection can now be processed independently.

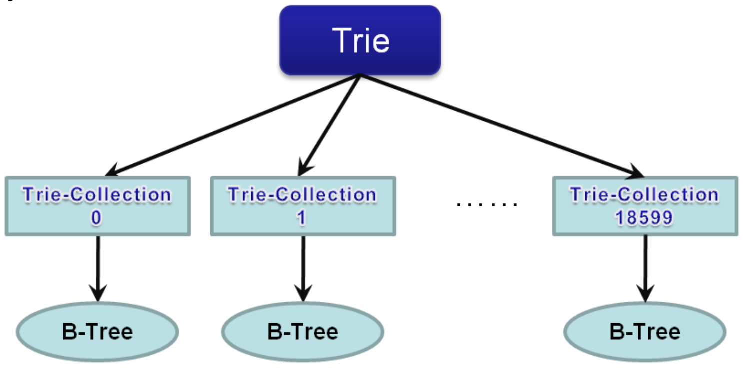


Figure 5.2. A Hybrid Data Structure of Dictionary

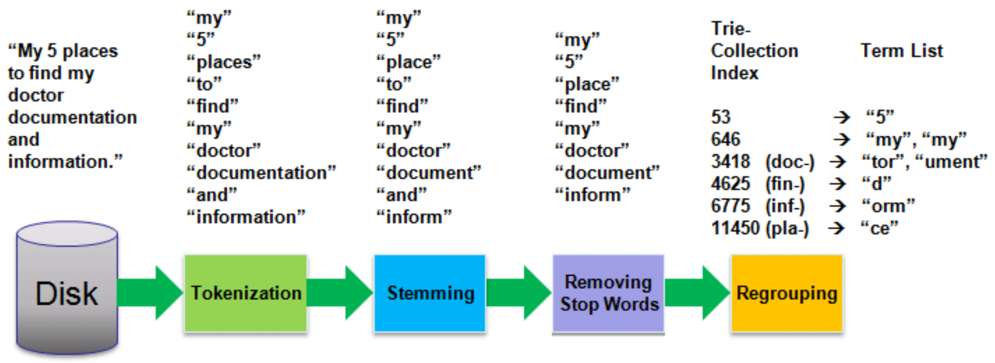

"My 5 places to find my doctor documentation and information."
"my" "5" "places" "to" "find" "my" "doctor" "documentation" "and" "information"
"my" "5" "place" "to" "find" "my" "doctor" "document" "and" "inform"
"my" "5" "place" "find" "my" "doctor" "document" "inform"
Trie-Collection Index
Term List
53 → "5"
646 → "my", "my"
3418 (doc-) → "tor", "ument"
4625 (fin-) → "d"
6775 (inf-) → "orm"
11450 (pla-) → "ce"
Disk
Tokenization
Stemming
Removing Stop Words
Regrouping

Figure 5.3. Data Flow of One Parser Thread

During the parsing stage, we use a traditional parsing algorithm that ends with a regrouping step as illustrated in Figure 5.3. This step rearranges the terms with the same trie index so that they are located contiguously (and the prefix of each term captured by the trie index is removed). The overhead of this step is relatively small and this regrouping is needed to partition the parsed data for the indexing stage. The assigned VIDs are local within each parser. A global VID offset will be calculated by the indexer; and the global VID will be obtained by adding the local VID and the global offset. In the temporal case, parsers also pass the URI, document length and start date of each VID to the indexers.

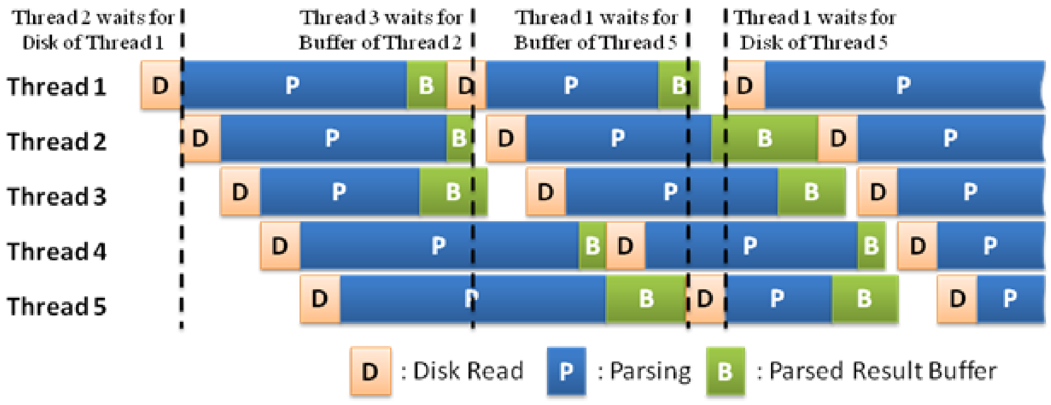

Thread 2 waits for Disk of Thread 1
Thread 3 waits for Buffer of Thread 2
Thread 1 waits for Buffer of Thread 5
Thread 1 waits for Disk of Thread 5
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
D : Disk Read
P : Parsing
B : Parsed Result Buffer

Figure 5.4. Timing Sequence of Parallel Parsers

To prevent parsers from trying to read from the same disk simultaneously, a scheduler is used to organize the reads of the different parsers, one at a time. On the other hand, an output buffer is allocated to each parser to store the corresponding parsed results. The indexers in the next stage will read from these buffers in order, that is, (buffer of *Parser 0*, buffer of *Parser 1*, …, buffer of *Parser M-1*, buffer of *Parser 0*, …). Such read sequence is enforced to ensure that documents will be indexed in the same order they are read, and hence the postings lists are intrinsically in sorted order of assigned VIDs and the start times of the document versions. A parser has to also wait until buffer is cleared to start the parsing of the next block of documents to ensure that it has the space to write the parsed results. When these constraints are applied, the timing sequence of parallel parsers looks like the example shown in Figure 5.4.

Once parsed results are available in a buffer, the corresponding URI hash table and VID table are generated. The work load here is trivial compared to the work to be performed by the indexers since for one document version the above two tables are updated once while each term of each document version has to be handled separately by the indexers.

Once the URI hash table and the VDI table of the corresponding parsed stream have been constructed, an indexer will construct all the B-trees and the postings lists corresponding to each input term in the stream. To ensure load balancing, a CPU thread will take care of the B-trees of several trie collections assigned such that each indexer processes more or less the same number of tokens.

5.1.3 Temporal Merging Stage

The temporal merging stage is invoked only at each time window boundary, where the current time window closes and a new one starts.

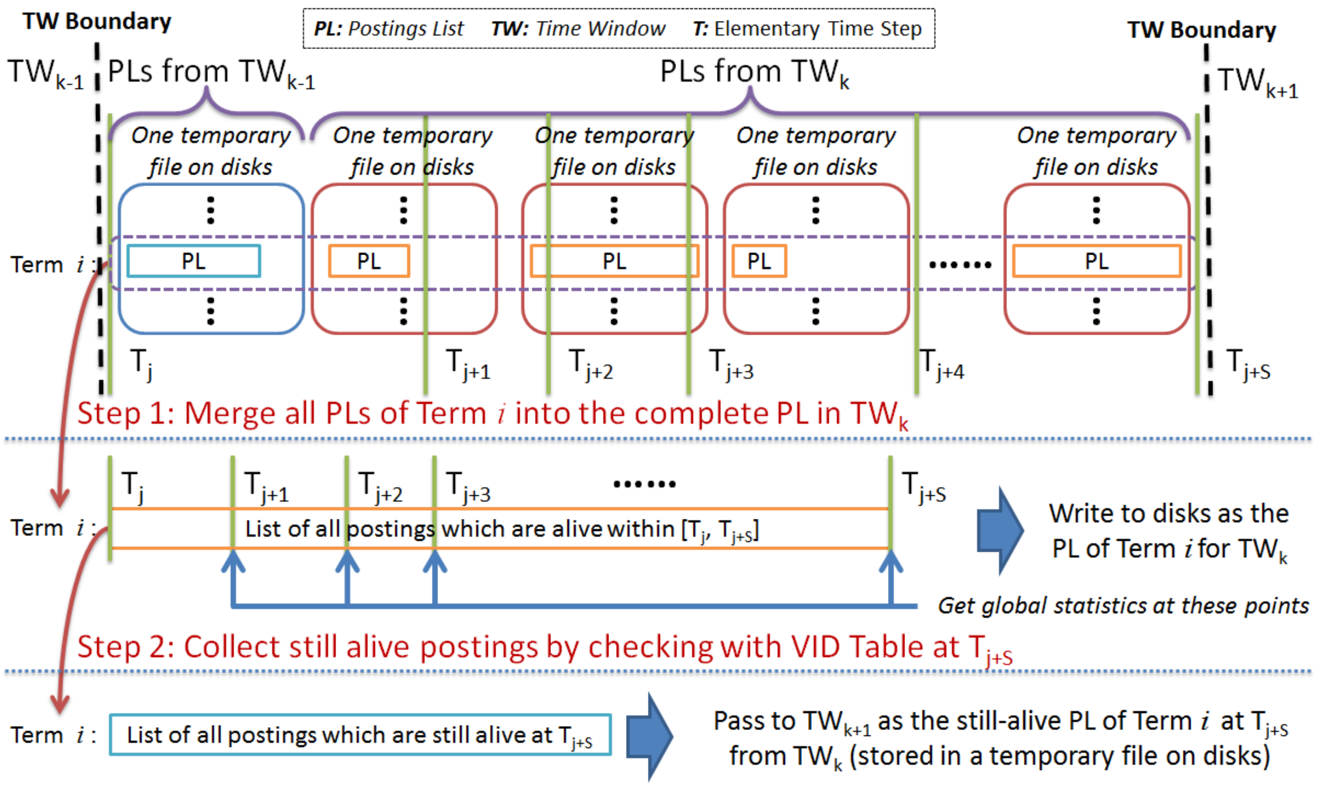

TW Boundary
PL: Postings List TW: Time Window T: Elementary Time Step
TW Boundary
TW$_{k-1}$
PLs from TW$_{k-1}$
PLs from TW$_k$
TW$_{k+1}$
One temporary file on disks
Term i :
PL
T$_j$ T$_{j+1}$ T$_{j+2}$ T$_{j+3}$ T$_{j+4}$ T$_{j+S}$
Step 1: Merge all PLs of Term i into the complete PL in TW$_k$
List of all postings which are alive within [T$_j$, T$_{j+S}$]
Write to disks as the PL of Term i for TW$_k$
Get global statistics at these points
Step 2: Collect still alive postings by checking with VID Table at T$_{j+S}$
List of all postings which are still alive at T$_{j+S}$
Pass to TW$_{k+1}$ as the still-alive PL of Term i at T$_{j+S}$ from TW$_k$ (stored in a temporary file on disks)

Figure 5.5. Data Flow in the Temporal Merging Stage

The input data to the temporal mergers for the time window TW_k consists of a temporary file passed from time window TW_{k-1} and several temporary files generated during this time window. A temporal merger processes one term at a time such that the postings list of this single term can fit in memory. It starts by reading out all the partial postings lists from the temporary files. These postings lists are then combined into the complete postings list of this term in TW_k, which contains all postings that are alive within time $[T_j, T_{j+S})$. From this complete list, we can get the needed global statistics at each elementary time step for this term. For example, the document frequency of this term at T_{j+2} can be obtained by counting the number of postings that are alive at T_{j+2}. The postings lists of all terms are written onto the disk. In our current implementation, we generate one value of average document length for each elementary time step and the document frequency for each term and each elementary time step. The cost of accessing the document frequency of each term was already reported in Section 4.3. However in the case when we have hundreds of thousands of elementary time steps, we can approximate the document frequency of each term at a time step by the average document frequency of the term over the time window. As reported in [12], this approximation does not significantly change the relative rankings of document versions for a random sequence of queries, and results substantial space savings in the case of a large number of elementary time steps.

In the next step, we go through the postings of this term to find out all postings which are still alive at the end of TW_k, that is, at the elementary time step T_{j+S}. The temporary file containing such postings is therefore passed to the next time window TW_{k+1}.

Even though temporal mergers are reading and writing multiple files at the same time, each file is accessed contiguously. The reason for this is that when the postings list of Term *i* is finished, the temporal merger moves onto Term *i+1* whose file locations are in contiguous locations of Term *i* for both the input and the output files. To exploit this property, we use buffered file read and write (a typical buffer size is 10MB) to alleviate disk read/write conflicts.

When the entire data collection is ingested and the last time window is completed using an appropriate number of temporal mergers, we end up with two types of postings lists:

- **postings lists for each time window:** *postings lists capturing all the document versions that are alive in this time window, which are enough to serve any query that falls within this time window.*
- **postings lists of live document versions:** *the temporary file of postings lists containing live postings, as the byproduct from the last time window, provides a useful way to search for the most up to date version of a document if it still exists. Typically, this is the type of postings lists generated by today's search engines – postings lists referring to current live web pages.*

To carry out incremental updating, we can easily use the same algorithm incorporating the postings lists of still alive VIDs at the end of the new time window.

5.2 Parallel Algorithm on a Cluster

We now extend our single node strategy to a cluster of multicore processors in a similar vein as in [15]. There are several possible strategies for such an extension, all of which assume that document versions are distributed on the disks of the various nodes of the cluster and hence the parsing stage can be carried out by all the nodes independently.

- **Partition-by-Time-Window.** *Each node fetches the input files corresponding to one time window, builds the local dictionaries and postings lists for this time window. The merging phase is not needed for temporal queries because we only search within one time window for a query.*
- **Partition-by-URI.** *At the end of each parsing stage, we partition documents among indexers based on their URIs following the single node algorithm such that each node only processes a portion of the document collection, after which the local dictionaries, local URI hash table, local VID table, and local postings lists from all the nodes are merged. This method follows the standard divide-and-conquer strategy and hence its effectiveness depends on the merging phase.*
- **Partition-by-Trie-Collection.** *This strategy includes a sampling preprocessing step that creates a persistent mapping between the trie collection indexes and the IDs of the indexers, which is used to distribute the parsed streams to the nodes. On the other hand, temporal information for building the URI hash table and the VID table is distributed to indexers by hashing on the URIs and each indexer builds local URI hash table and local VID table on its assigned portion of URIs. A synchronization of VID tables is needed to obtain the global VID table when a time window is closed.*

It is clear that the first strategy will achieve excellent performance during the processing of each batch of a time window. However the temporal mergers can't finalize the postings lists of the current time window until they receive the still alive postings from the previous time window. As a result, the lower bound of the running time is $\sum$(*time in the merging stage in each time window*) which makes this strategy ineffective.

The bottleneck of the second strategy lies in the global merging stage. Such merging is required since for a given query term we have to search the same term in all the local dictionaries and the local postings lists to find all matching results. Moreover, if newer documents are ingested afterwards, we have to carry out the complete merging process again.

The last approach is extended from the partition-and-index approach described in [15] with the advantage of a coherent, global dictionary and the global VIDs stored in the postings lists on all nodes. The global synchronization of VID tables will cause some overhead but it is well hidden by our pipeline design. As a result, we choose this strategy to implement.

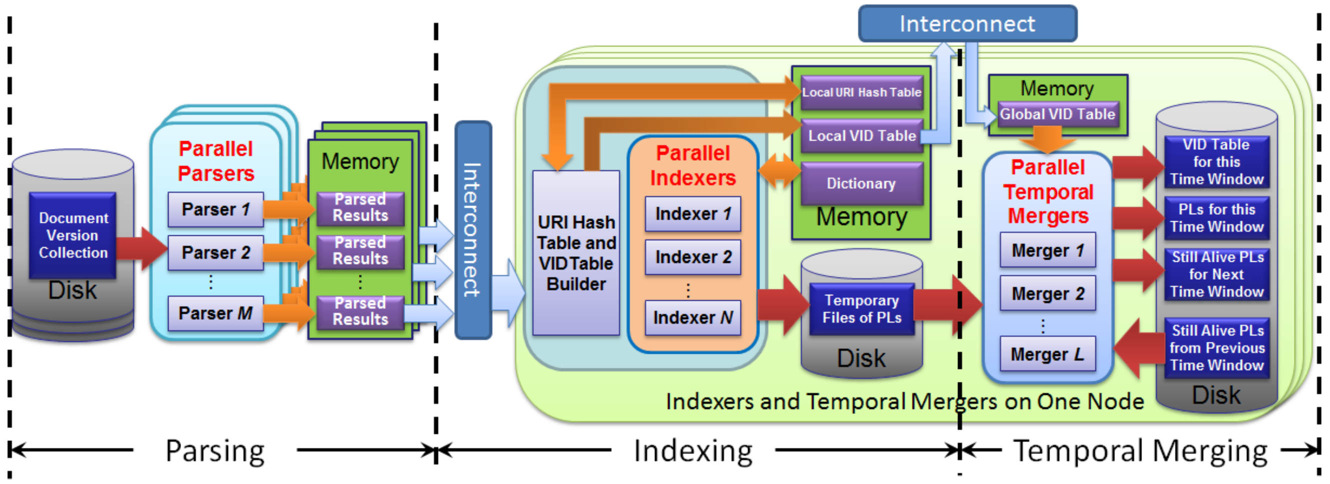


Figure 5.6. Data Flow of Partition-by-Trie-Collection-and-URI Strategy

The data flow of the third approach is illustrated in Figure 5.6. The indexers and temporal mergers still share the same disks and hence the temporary files in Figure 5.5 will be written and read locally; however, at time window boundaries, the temporal mergers on different nodes have to exchange their local VID tables to generate the global VID table.

Our algorithm on the cluster distributes the task of building the URI hash table and the VID table evenly among the nodes by hashing on the URIs. This is applicable because only VIDs crawled from the same URI will have temporal relations. However, when a time window closes, the distributed VID tables on all nodes are synchronized to reflect the complete and global view of all VIDs. This is a required step because the postings lists generated by the indexers on different nodes are not partitioned by URIs. Consequently, the postings handled by a temporal merger on one node may contain any VIDs residing on an arbitrarily different node, not necessarily restricted to the VIDs assigned to this node by the URI hashing function. On the other hand, the local URI hash tables are not synchronized since they are not used in the temporal merging stage due to the fact that they contain neither the start time nor the end time of VIDs which are required by temporal mergers to check if a posting is still alive.

An alternative implementation is to let all nodes build their own copies of the global URI hash table and VID table from the parsed results. Such an approach eliminates synchronization by duplicating computation at every node. If the average time to build these two temporal tables is T_1 and the average indexing time is T_2 for each batch, then the processing time of the indexers with P nodes becomes $T_1 + T_2/P$. When P is over eight in our experiments, T_1 will be dominating and hence the indexing speed does not really improve with larger P. In our algorithm, the processing time is $(T_1 + T_2)/P$, which is clearly scalable. The temporal mergers are responsible for this synchronization and hence parsers and indexers will not be stalled due to this communication operation.

Putting all the pieces together, we get the overall data flow shown in Figure 5.6 for a cluster of multicore processors. Similar to the single node case, we use synchronous communication to enforce the sequence of messages processed by indexers, that is, each node sequentially receives messages from node 1 through node *P* to guarantee the order of VIDs in the postings lists. Indexers on different nodes start indexing once the parsed results are received via the interconnect and they don't necessarily communicate with other indexers; on the other hand, all the temporal mergers will begin at the same time once the VID table synchronization is done.

6. EXPERIMENTAL EVALUATION OF OUR PARALLEL ALGORITHMS

The performance of our single multicore node is evaluated using the Congressional dataset, where a node consists of two Intel Xeon X5560 Quad-core CPUs with 24 GB of memory and 210 GB disk. The cluster algorithm is tested on a cluster of 32 of such nodes that are interconnected by a 10Gb/s InfiniBand.

In what follows, we start by exploring the optimal empirical values of the numbers of parsers, indexers and temporal mergers for the single node algorithm (described in Section 5.1). We then show that our cluster algorithm is scalable relative to the optimized single node algorithm, up to the largest number of available nodes, as well as the performance of our single node algorithm on the two document collections in Table 4.1.

6.1 Number of Parallel Parsers, Indexers and Temporal Mergers on a Single Node

For simplicity we use the triplet <M, N, L> to represent the combination of *M* parsers, *N* indexers and *L* temporal mergers in the temporal case and the tuple <M, N> for the combination of *M* parsers and *N* indexers in the non-temporal case. We conduct tests based on the following rules:

1) the total number of parsers and indexers is at most eight beyond which the parsers and indexers will compete for the available cores which will hurt the overall performance;

2) the tasks of temporal mergers are mainly disk I/O bound, and hence we let indexers and temporal mergers share the same cores should the total number of running threads exceeds eight;

3) As shown in [14, 15], two indexers are fast enough to keep up with up to six parsers but using a single indexer will constitute a major bottleneck, and hence we fix the number of indexers to two;

4) During the indexing stage, each indexer generates its own portion of postings lists and hence it seems to be impossible to effectively assign multiple temporal mergers to process a single indexer's output; therefore, the postings lists from a single indexer will only be processed by a single temporal merger, which means L must be less than or equal to N.

Under these rules, we consider the following combinations of the numbers of parsers, indexers and temporal mergers: <6, 2, 2>, <6, 2, 1>, <5, 2, 2>, <5, 2, 1>, <4, 2, 2>. The Congressional dataset is used and three time windows are created under the Even-Size strategy. To show the extra cost from temporal processing, the throughput of the non-temporal case using <6, 2> is also included, where we use the algorithms in [15] and treat every document version in the collection as a new and independent document.

Figure 6.1 illustrates the resulting throughputs of the various combinations. It is clear that the combination <6, 2, 2> of our temporal algorithm achieves the best throughput reaching to over 60% of the throughput achieved by the best non-temporal algorithm that uses six parsers and two indexers. Therefore our single multicore node algorithm to generate the inverted files in the temporal case incur a very reasonable overhead in spite of the fact that it has to deal with the extra temporal dimension.

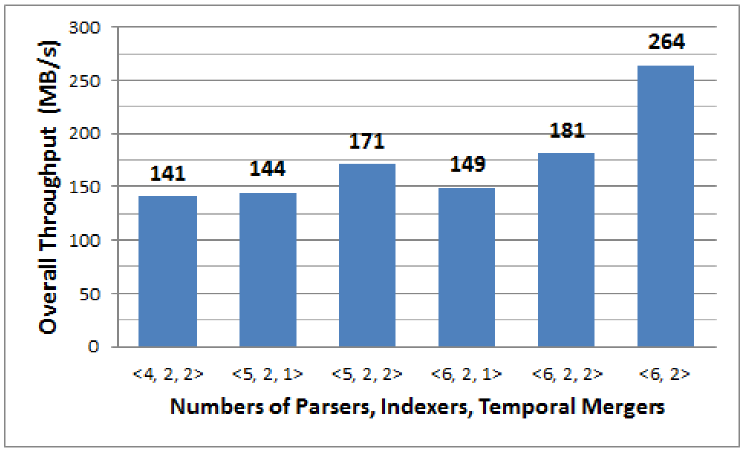


Figure 6.1. Throughput on a Single Node

We now take a closer look at in the performance of each stage of the pipeline as illustrated in Figure 6.2. In the non-temporal case and as expected, the execution times of the parsing and indexing stages are the best in all the three time windows, followed by the temporal case using the combination <6, 2, 1>, which is due to the extra work of building two temporal tables and the impact of the new temporal merging thread. The parsing and indexing speed is slowest when using the combination <6, 2, 2>; however, the overall throughput is higher than using the combination <6, 2, 1> because of the much slower temporal merging stage.

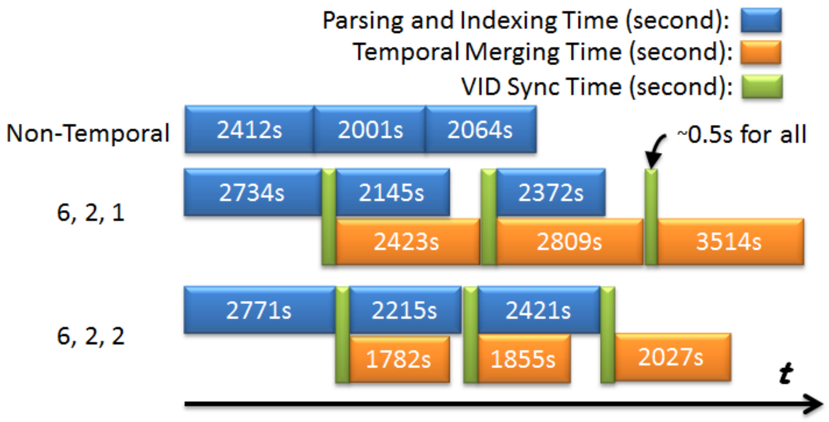


Figure 6.2. Detailed Timeline in the Pipeline with Three Time Windows

6.2 Throughput on the Cluster

We now evaluate the performance and the scalability of our cluster algorithm relative to the number of nodes. The speedup is calculated relative to our best algorithm on a single node, and hence the speedup of our cluster algorithm on a single node is less than 1 due to the cluster algorithm overhead such as the overhead incurred by the MPI communication operations. We note that the optimal combination of the numbers of parsers, indexers and temporal mergers is not necessarily <6, 2, 2> in all scenarios. For example the combination of <5, 2, 2> yields a throughput that is 6% better than the combination of <6, 2, 2> on 32 nodes. The optimal choice also changes with different time window numbers and splitting methods, but in general the result from <6, 2, 2> is very close to the best possible (within 10%). Our main focus is to study the extra cost of temporal processing and to carry out a direct comparison with the non-temporal case (where <6, 2> is applied). Again, we use the Congressional dataset with three time windows under the Even-Size strategy.

6.2.1 Scalability over the Number of Nodes

From Table 4.7 and Figure 6.3, we consider the throughput of our temporal and non-temporal algorithms as a function of the number of nodes. Both algorithms scale almost linearly with the number of nodes. The speedup values in the temporal and non-temporal cases are relatively close and the ratio is around 0.70. As we have shown in [15], our non-temporal algorithm achieves a very good load balancing, and given the constant ratio with our temporal algorithm, we can conclude that the latter also achieves good load balancing.

Table 4.7. Throughput Over the Number of Nodes

Number of Nodes	**Temporal (6, 2, 2)**		**Non-Temporal (6, 2)**		**Throughput Ratio**
	Throughput (MB/s)	**Speedup**	**Throughput (MB/s)**	**Speedup**	
1	172	0.95	249	0.94	0.69
2	312	1.72	480	1.81	0.65
4	619	3.41	935	3.54	0.66
8	1226	6.76	1717	6.50	0.71
16	2382	13.14	3176	12.02	0.75
32	3796	20.93	5401	20.44	0.70

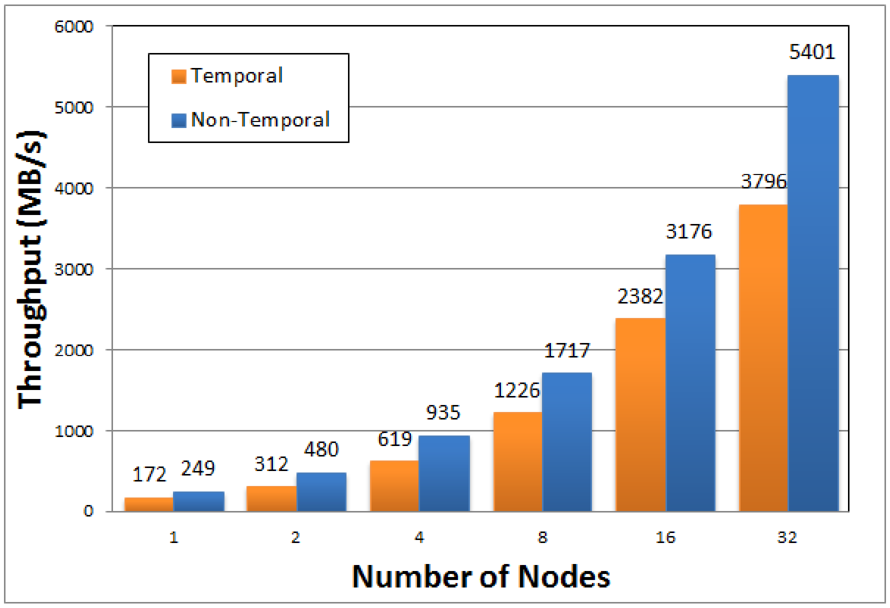


Figure 6.3. Scalability over the Number of Nodes

6.2.2 Detailed Running Time of the Pipeline

Figure 6.4 illustrates the running time for each time window. Similar to Figure 6.2, it takes longer to parse and index the document versions whenever indexers and temporal mergers share the same physical CPU cores. Compared with the running time on a single node, the VID table synchronization time is also significantly larger because of the all-to-all communication between the 32 nodes. Another interesting fact is that the temporal merging time, which is over 40 times faster than the one on a single node. This is due to the fact that, with *P* nodes, the sizes of input and output files in Figure 5.5 are only 1/*P* of the corresponding files on a single node, in which case the file access caches can significantly impact the file reading and writing throughput. In Table 4.8, we list the speedup over the number of nodes by summing up the total running time spent in the temporal merging stage. It is obvious that the super-scalability doesn't show up until the number of nodes reaches eight in which case the size of a temporary file in Figure 5.5 is as low as about tens of MBs, which is close to the 10MB memory buffer size we mentioned in Section 5.1.2 for each temporary file.

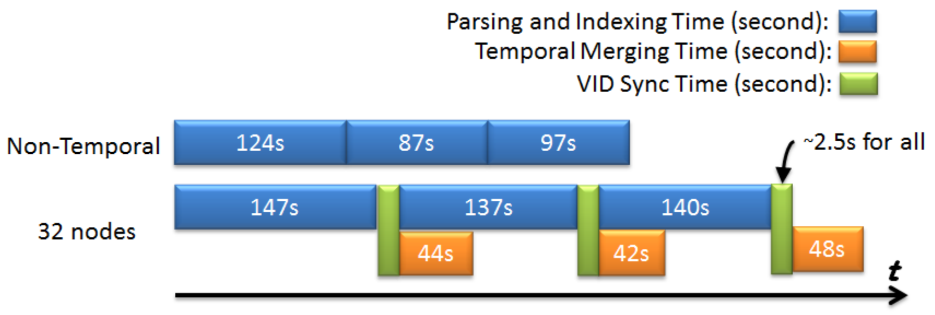


Figure 6.4. Detailed Timeline on the Cluster

Table 4.8. Speedup in the Merging Stage

Number of Nodes	Speedup of Temporal Merging Stage
1	1.00
2	1.63
4	3.75
8	9.41
16	20.29
32	43.76

6.2.3 Latency of the Pipeline

Besides the throughput, another important factor for incremental updating is the latency between the time a document version is inserted into the pipeline and the time that this document version is included in the postings lists on disks and hence is available for searching. We tested the maximum latency relative to different number of time windows under the Even-Size strategy. Since in our current strategies, document versions are included into the final and permanent postings lists at the end of each time window, it is expected that the latency decreases as the number of time windows increases (and their sizes decrease) as shown in Figure 6.5. However, we also need to process the postings lists passed from previous window, which costs the majority of the running time with a large number of time windows and prevents us from further improving the latency. Overall, with 32 nodes the latency is well controlled in less than 4 minutes.

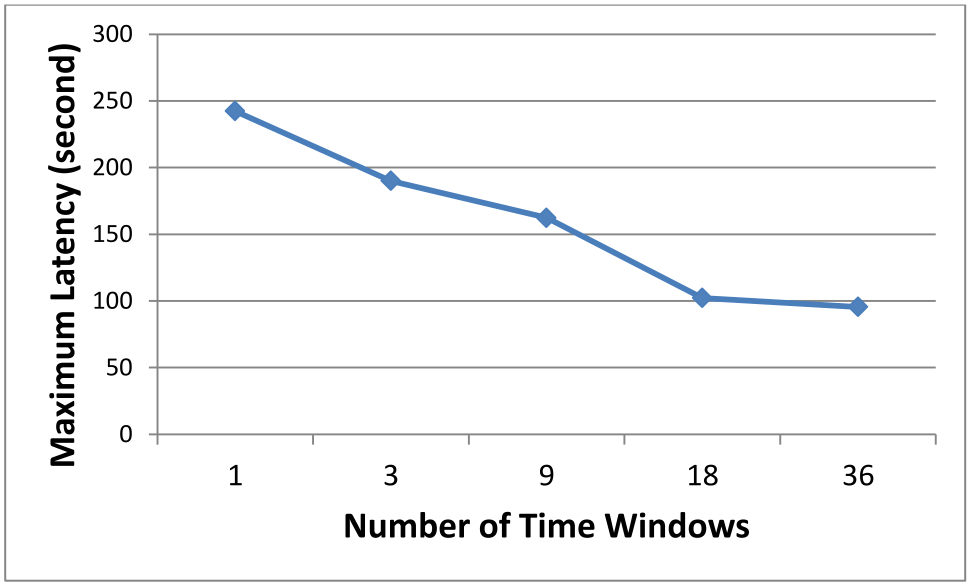


Figure 6.5. Maximum Latency Relative to the Number of Time Windows

6.2.4 Throughput over the Number of Time Windows

In this section we test the overall running time and throughput with different numbers of time windows on our cluster with 32 nodes. With only a single time window, the process of temporal merging won't start until all parsing and indexing tasks are finished and hence the temporal merging time is not hidden by the parsing and indexing time at all. When the number of time windows increases, the time cost by the last batch of temporal merging (the right most temporal merging time in Figure 6.4) is also reduced, that's why the running time decreases when the number of time window grows from 1 to 9. Beyond that, VIDs that are alive across time window boundaries are more likely to be duplicated in the postings lists of several time windows, which can be verified by the drastically inflated size of output postings lists. Writing such duplicates onto disks consumes a significant amount of CPU cycles and as a result the overall running time climbs back to even higher values when there are 18 or 36 time windows.

Table 4.9. Throughput over the Number of Time Windows

Number of Time Window	Running Time (second)	Throughput (MB/s)	Compressed Output PL Size (GB)
1	538	3185	33.7
3	451	3796	42.0
9	408	4196	88.6
18	550	3110	136.6
36	612	2797	248.1

6.2.5 Performance of our Algorithm on Different Document Collections

We show in Table 4.10 the overall throughput of our algorithm on our two document collections with the parameter combination of <6, 2, 2>. For the Wikipedia01-07 collection, the HTML tags were removed, and the remainder was just pure text. As we can see from Table 4.1, the uncompressed size is only 1/21th of the Congressional collection, yet the number of tokens is about a fourth compared to the one of the Congressional collection. Hence the 50MB/s throughput achieved on Wikipedia01-07 actually amounts to a very high processing speed given the large numbers of tokens. Since the size of Wikipedia01-07 is too small, we could not get any meaningful speedup beyond four nodes and hence those results are not included in Table 4.10. The scalability from single to four nodes is as good as the one achieved on the Congressional collection. We believe that, should we have much larger Wikipedia document version collection, similar speedups as shown in Table 4.7 can be achieved for 8 to 32 nodes.

Table 4.10. Throughput on Different Document Version Collections

	Throughput (MB/s)			
	Congressional		**Wikipedia 01-07**	
	Temporal	**Non-Temporal**	**Temporal**	**Non-Temporal**
Single Node	181.35	264.26	51.33	78.29
Four Nodes	618.54	935.08	192.12	242.05

7. CONCLUSION

In this work, we introduced a new approach to build the inverted files and auxiliary indexing structures to enable temporally anchored text-based search of a collection of time-evolving documents. We presented effective strategies to build the inverted files and the indexing structures, both on a single multicore processor and a cluster of such processors, and illustrated the performance of our algorithms on two significant collections using several parameters such as search time, space, and construction time. In particular, we showed that our overall throughputs on the two collections reach around 70% of the fastest known algorithms when the collections are processed without taking into consideration the temporal dimension. Moreover, our overall strategy can naturally handle additional documents incrementally without having to completely rebuild the inverted files from scratch.

ACKNOWLEDGEMENT

We would like to thank the Library of Congress and the Internet Archive for making the congressional dataset available to us. We would also like to thank Sangchul Song who developed the version of Wikipedia04-09 dataset used in our experimental tests. This research was partially supported by the NVIDIA Research Excellence Center at the University of Maryland and the NSF Research Infrastructure award, grant number CNS-0403313.

REFERENCES

[1] L. Mearian, “The Internet Archive's Wayback Machine gets a new data center,” Computer-world.com, Mar. 2009, http://www.computerworld.com/s/article/9130499/The_Internet_Archive_s_Wayback_Machine_gets_a_new_data_center, Accessed: August 15, 2012.

[2] "Minerva: Library of Congress Web Archives" Available: http://lcweb2.loc.gov/diglib/lcwa/html/lcwa-home.html, Accessed: Octorber. 3, 2011.

[3] "UK Web Archiving Consortium" Available: http://www.webarchive.org.uk, Accessed: Octorber. 3, 2011.

[4] "Pandora: Australia's Web Archive" Available: http://pandora.nla.gov.au, Accessed: Octorber. 3, 2011.

[5] "Web-at-Risk," Dec. 2008 Available: https://wiki.cdlib.org/WebAtRisk/tiki-index.php, Accessed: Octorber. 3, 2011.

[6] http://www.archive.org/web/petabox.php, Accessed: April. 30, 2012.

[7] K. Nørvåg, "V2: a database approach to temporal document management," Proceedings of the seventh Database Engineering and Applications Symposium (IDEAS 2003), Hong Kong, China: IEEE Computer Society, 2003, pp. 212-221.

[8] K. Nørvåg, "Space-Efficient Support for Temporal Text Indexing in a Document Archive Context," Proceedings of the seventh European Conference on Digital Libraries, Trondheim, Norway: Springer Verlag, 2003, pp. 511-522.

[9] K. Nørvåg and A.O. Nybø, "DyST: Dynamic and Scalable Temporal Text Indexing," Proceedings of the Thirteenth International Symposium on Temporal Representation and Reasoning, IEEE Computer Society, 2006, pp. 204-211.

[10] K. Berberich, S. Bedathur, T. Neumann, and G. Weikum, "A time machine for text search," Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval, Amsterdam, The Netherlands: 2007.

[11] A. Anand, S. Bedathur, K. Berberich, and R. Schenkel, "Temporal Index Sharding for Space-Time Efficiency in Archive Search", Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval, New York, USA, 2011, pp. 545-554.

[12] S. Song and J. JaJa, "Effective Strategies for Temporally Anchored Information Retrieval", UMIACS Technical Report.

[13] S. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford, "Okapi at TREC-3," Proceedings of the Third Text REtrieval Conference (TREC 1994), Gaithersburg, USA, November 1994.

[14] Z. Wei and J. JaJa, "An Optimized High-Throughput Strategy for Constructing Inverted Files", to be published on IEEE Transactions on Parallel and Distributed Systems.

[15] Z. Wei and J. JaJa, "Constructing Inverted Files on a Cluster of Multicore Processors Near Peak I/O ThroughputA Fast Algorithm for Constructing Inverted Files on Heterogeneous Platforms", Journal of Parallel and Distributed Computing 72 (2012), pp. 728-738.