PARALLEL TEXT RETRIEVAL ON TEMPORALLY VERSIONED DOCUMENT COLLECTIONS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BİLKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Özlem Gür

September, 2008

ABSTRACT

PARALLEL TEXT RETRIEVAL ON TEMPORALLY VERSIONED DOCUMENT COLLECTIONS

Özlem Gür
M.S. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
September, 2008

In recent years, as the access to the Internet is getting easier and cheaper, the amount and the rate of change of the online data presented to the Internet users are increasing at an astonishing rate. This ever-changing nature of the Internet causes an ever-decaying and replenishing information collection where newly presented data generally replaces old and sometimes valuable data. There are many recent studies aiming to preserve this valuable temporal data and size and number of temporal Web data collections are increasing. We believe that soon, information retrieval systems responding to time-range queries in a reasonable amount of time will emerge as a means of accessing vast temporal Web data collections. Due to tremendous size of temporal data and excessive number of query submissions per unit time, temporal information retrieval systems will have to utilize parallelism as much as possible.

In parallel systems, in order to index collections using inverted indices, a strategy on distribution of the inverted indices has to be followed. In this study, the feasibility of time-based partitioned versus term-based partitioned temporal-web inverted-indices is analyzed and a novel parallel text retrieval system for answering temporal web queries is implemented considering the number of queries processed in unit time. Moreover, we investigate the performance of skip-list based and randomized-select based ranking schemes on time-based and term-based partitioned inverted indexes. Finally, we compare time-balanced and size-balanced time-based partitioning schemes. The experimental results at small to medium number of processors reveal that for medium to long length queries time-based partitioning works better.

Keywords: Temporally versioned document collections, parallel text retrieval, inverted index partitioning, query processing, search engines.

ÖZET

ZAMANSAL SÜRÜMLENDİRİLMİŞ DÖKÜMAN KOLLEKSİYONLARINDA PARALEL METİN ERİŞİMİ

Özlem Gür
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Eylül, 2008

Son yıllarda, İnternet erişimi giderek kolaylaştıkça ve ucuzladıkça, İnternet kullanıcılarına sunulan verinin miktarı ve değişim hızı şaşırtıcı boyutlara ulaşmaktadır. İnternet'in sürekli değişen yapısı, yeni verilerin kimi zaman önemini kaybetmemiş eski verilerin yerini aldığı, sürekli değişen ve güncellenen bir bilgi kolleksiyonunu doğurur. Bu önemli zamansal verileri korumayı amaçlayan çok sayıda yeni çalışma literatürde mevcuttur ve bu çalışmaların sayıları kadar arşiv boyutları da gün geçtikçe artmaktadır. İnanıyoruz ki, yakın gelecekte, geniş kapsamlı zamansal ağ veri kolleksiyonlarına erişebilme hedefi doğrultusunda, makul bir süre zarfında zaman aralığı sorgularına cevap verebilen metin erişimi sistemleri ortaya çıkacaktır. Zamansal verilerin devasa boyutları ve birim zamana düşen aşırı miktardaki sorgu sayısı, zamansal bilgi erişimi sistemlerini mümkün olduğunca paralel uygulamaları kullanmaya itecektir. Paralel sistemlerde, veri kolleksiyonlarını ters dizin endekslerini kullanarak endekslemek için, ters dizin endekslerinin dağıtımı üzerine bir strateji izlenmelidir. Bu çalışmada, zamana göre ve terimlere göre bölümlendirilmiş zamansal ağ ters dizin endekslerinin yapılabilirliği incelenmiş ve birim zamanda cevaplanan sorgu sayısı göz önünde bulundurularak, zamansal ağ sorgularını cevaplayabilen yeni bir paralel metin erişimi sistemi uygulaması sunulmuştur. Ayrıca, atlama listelerini ve rasgele seçim algoritmalarını kullanarak sorgu sonuçlarını sıralayan yöntemlerin zamana göre bölümleme şeması üzerindeki performansları karşılaştırılmıştır. Küçük ve orta sayıdaki işlemciler üzerinde yapılan deneyler, orta ve uzun sorguların zamana göre bölümlenmiş ters dizinlerde daha iyi sonuç verdiğini ortaya koymuştur.

Anahtar sözcükler: Zamansal sürümlendirilmiş döküman kolleksiyonları, paralel metin erişimi, ters dizin bölümleme, sorgu işleme, arama motorları.

Acknowledgement

First, I would like to express my gratitude to Prof. Dr. Cevdet Aykanat for his valuable suggestions, support and guidance throughout my M.S. study.

I would also like to thank Prof. Dr. Enis Çetin and Assoc. Prof. Dr. Uğur Güdükbay for reading and commenting on this thesis.

I would like to acknowledge Berkant Barla Cambazoğlu for providing the source codes of Skynet parallel text retrieval system and A. Aylin Tokuç for extending his work and sharing it with me. She has been a great support as a respected colleague and a dear friend who used to accompany me during long working hours till morning.

I also thank Ata Türk for introducing this research topic to me, to Tayfun Küçükyılmaz for proofreading my thesis and to İ. Sengör Altıngövde and Enver Kayaaslan for their valuable comments and ideas on improving my thesis. I would also like to voice my gratitude to İzzet Çağrı Baykan for his patience as an office mate and to my precious friends M. Cihan Öztürk and Sare Sevil for their moral support.

Last, but not least, I am grateful to my family who have always been my tower of strength. Without them, this thesis would have never been complete.

To my mother

for her everlasting support and love

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In last few decades, the Internet has become an inseparable part of daily life and is slowly taking place of libraries. Today, World Wide Web serves thousands, perhaps millions of users around the world and the number of the Internet users is rapidly increasing. Additionally, new pages are inserted in the web on daily basis and the contents of the existing pages are also changing continuously. According to Brewington and Cybenko [5], the average lifetime of a web page without any modification or deletion is about 138 days. Cho and Garcia-Molina [15] crawled 720,000 pages for four months and observed that 40% of the web pages changed within a week where 23% of these pages changed daily. Therefore, the Internet cannot be seen as a static data repository but an ever-changing highly dynamic infrastructure.

Time-related information has great importance for fast-changing datasets such as the Internet. Even before the Internet, libraries used to archive old newspapers in case of a request from library users. These archives served users as a temporal document collection which can be accessed for finding former material. Researchers, especially historians would like to search in temporal data frequently in order to have extensive information about the roots of their research topics.

With the growing use of the Internet, there is a common tendency of converting textual data into digital media. Thus, temporal information is even more important today for the Internet. Moreover, the Internet also has special properties that make temporal information valuable to its users. Firstly, the age of a web page reflects its freshness. The older the page is, the more likely it is to be outdated. A new page has a higher probability to contain fresh information and therefore is more reliable. Second, the modification history of a page demonstrates that page's reliability. A page which is modified frequently would be more reliable because its mistakes would be corrected by the users. On the other hand, a rarely modified page would be less reliable since it lacks verification. Third, insertion/modification date of a web page might reflect valuable information about that page which is otherwise unavailable. The relation between the content and temporal properties of a page may lead researchers to mining many useful results.

For the past few decades, there is an ongoing research on temporal repositories and its applications. Recent studies on temporally versioned document collections include web browsers that enable elimination of the broken-link problem [16], visualization of earlier versions of web pages and summarization of web pages with their revisions [20]. The broken-link problem is particularly severe for search engines [43], because they might be leading millions of users to a page that no longer exists. Furthermore, summarization functionality of a browser will give hints about the characteristics of a web page by displaying the variations in time.

Another challenge concerning temporal databases is the increasing index size due to the existence of several versions which are generally almost identical. Since disk space is a scarce resource in computer science that should be used carefully, index size reduction research receives even more attention. Moreover, reducing the index size, indirectly leads better temporal query performances. Therefore, several researchers focused on crawling and efficient storage of versioned document collections that make use of data redundancy [6, 12, 1, 37, 22, 43]. There are also several researchers studying on temporal data structures such as the variations of B-Tree or R-Tree based methods, targeting directly query response time. Most of these techniques sacrifice a feasible amount of space in order to obtain better query performance. After all, the Internet users are mainly concerned about the time required for answering a query [39] rather than the space consumption.

Digital documents tend to include temporal information due to increasing popularity on time-related research. Time-related properties of web documents are generally extracted from Last Modified tag of HTML files or approximated based on last crawl date. Moreover, with the increasing use of XML files, time information is being embedded in documents in a standard format. The increasing interest in temporal databases also encouraged several institutions to archive the Web. Internet archive is one of the most well known web repositories among these institutions and provides more that 85 billion web pages. That makes roughly one petabyte of temporal data which is growing at a rate of 20 terrabytes per month [24, 25, 45]. Google, Live Search and Yahoo crawls the web pages with versioning information as well. Other temporal data repositories include accounting documents of companies, email servers, version control systems (e.g. CVS, ClearCase), wikis, Internet forums and blogs [22].

In this work, we propose a method that enables temporal search on temporally versioned document collections to search engines. Our contributions are twofold: First, we implemented a novel parallel text retrieval system on temporal document collections. Second, we partitioned the data among processors in a time-based fashion. Organization of the rest of the thesis is as following: Chapter 2 gives a summary of related work about existing text retrieval techniques on temporally versioned document collections. Chapter 3 explains parallel text retrieval implementation details, indexing and ranking mechanisms. Then, Chapter 4 analyzes time-balanced and size-balanced time-based partitioning schemes. Chapter 5 displays experimental results including properties of real world Wikipedia and Wiktionary versioned document collections as well as synthetically generated temporal queries. Finally, Chapter 6 gives conclusion and future work on how to improve the suggested partitioning models in this thesis.

Chapter 2

Background

Temporal data is defined as the data that can be linked to a certain time or period between two moments in time [36]. Temporal web data is a specific type of temporal data that is an instance of Web resources such as web pages, images or videos. Time related information of temporal web data is estimated from last crawl dates or Last Modified field of HTML files or obtained from time stamp field of XML files.

In this chapter, we will discuss the studies on temporal text retrieval which is a relatively recent application on temporal data. We clustered the research on temporal text retrieval based on space/time efficiency considerations. First, we will mention naive approaches that aims neither space nor time utilization. Second, we will analyze space-improving approaches that aim at index size reduction/compression. Then, we analyze studies which are mainly concerned with optimizing query response time. Finally, we will try to explain studies that exploit both space and time utilization.

2.1 Temporal Text Retrieval

The studies on temporal text retrieval could be classified under four main groups based on space or time utilization: (1) naive, (2) space-improving, (3) query performance improving and (4) both space and query performance improving implementations. We will discuss the studies under each group in the given order.

2.1.1 Naive Implementations

Michael Stack from Internet Archive, published a study on indexing and searching small-to-medium sized temporal document collections on Nutch search engine [41]. Their implementation is extremely naive that suggests neither space nor query response time improvements. Indexing their small document collection that consists of about 1 million documents on one machine using single disc took more than 40 hours although the resulting index file is only 1.1 GB. The slowness of indexing phase is claimed to be due to parsing PDF files in the collection. Indexing medium sized document collection is performed on 2 machines in a distributed manner. Generation of an index file of size 5.2 GB took 99 hours, 86.4 hours of which was due to segmentation. Using these index files, they could only answer 1 query per second on average.

Another naive approach for searching among temporally versioned document collections is a Portuguese Web search engine called tumba! [40]. Tumba! is a temporal search engine specially built for Portuguese users. They implemented a parallel information retrieval system on temporally versioned Portuguese Web documents. However, they do not offer any special indexing scheme or query processing algorithm exploiting temporal information. They used Oracle's SIDRA tool for indexing and distributing data among processors by term-based partitioning. They used Vector-space model for generating scores of resulting documents/versions. They treat each version as a separate document and perform the search among this huge document collection containing highly redundant information.

2.1.2 Space-improving Implementations

One of the earliest suggestions on reducing space consumption in versioned document collections is explained in Anick and Flynn's paper [1]. They used time stamps of versions as version identifier, but this is not applicable to transaction based document collections where several versions could have the same time stamp. They proposed storing current versions of documents and storing the differences of the earlier versions as backward deltas where backward delta is the difference between two versions of the same document. This approach reduces the index size but increases the access cost to older versions. To be able to support temporal querying, they suggested a similar approach for indexes. They based the full-text index on bitmaps for current versions' terms and used delta change records' indexes to go backward in time. As expected, this approach performs well on queries at current time but is costly for historical queries.

Another work on optimizing space consumption on temporally versioned document collections is very recent and aims building a search engine on historical events. In order to reduce the index size, they selected documents with a certain format that includes at least a title and a few sentences. Furthermore, instead of indexing whole document, they only indexed titles and key sentences. As a side-effect, they compromise accuracy for the sake of significant space gain. We believe that search engines could present this approach as an option. If the accuracy of the results is not critical, the user might prefer searching in a partially indexed document collection. In this paper, Huang, Zhu and Li [23] also provided replica detection and clustering the results for display purposes.

Herscovici, Lempel and Yogev [22] also studied on space utilization in temporal document collections. In this work, by solving multiple sequence alignment problem, temporal document collections with a high redundancy rate are indexed. They used a variation of multiple sequence alignment problem to implement a greedy, polynomial time algorithm. The results on two real-life corpora demonstrate 81% improvement compared to the naive approach where all versions are indexed separately. Although the amount of reduction in the index size is significant, time required for indexing increased considerably. Therefore, this schema is only applicable where the amount of data to be indexed per unit time is not excessive. Although they allowed temporal queries, they did not report query response time values.

In the last work on space utilization that will be explained in this thesis, Zhang and Suel [45] deals with redundancy in large textual collections. First, they split versions into a number of fragments using content-dependant string partitioning methods such as winnowing and Karp-Rabin partitioning. Then, they index these fragments instead of indexing whole version. They compared global sharing and local sharing schemes where in the former scheme, a version is allowed to contain a fragment of another document's version and in the latter, a version consists of fragments of the same document's versions. They carried the experiments on search engine query logs and large collections which are generated after multiple crawls. According to the results, their approach causes significant index size reduction. Moreover, the first scheme, global sharing, performs 40%-50% better than local sharing.

2.1.3 Query Performance Improving Implementations

Before explaining query-performance based approaches on temporal text collections, let us define three main temporal query types that are used in these studies.

(1) Given a contiguous time interval T, find all versions in this interval.

(2) Given a set of keywords and a contiguous time interval T, find all versions in T containing these keywords.

(3) Given a set of keywords, find all versions containing these keywords.

Salzberg [39] gave the definition of three query types for specific instances of the three cases above. In time-only queries (1), if a time instant is given instead of a time interval, this type of query is called pure-timeslice query. In the same manner, if a time instant is given in time-key queries (2), it is called range-timeslice queries. In key-only queries (3), if only one keyword is given, then it is called a pure-key query.

In this section, we will explain five mechanisms for improving query performance in the chronological order of publishing dates.

Leung and Muntz [29] worked on temporal data fragmentation, temporal query processing and query optimization in multiprocessor database machines. Although, they did not work on text retrieval, their approach is applicable to Web. Basically, data fragmentation in temporal databases corresponds to data partitioning in parallel architectures. They mentioned three fragmentation schemes: (1) round-robin, (2) hashing and (3) range-partitioning. For round-robin distribution scheme, they split the time interval into k pieces and assigned the i^{th} interval to the processor (i mod k). The hashing scheme maps intervals to processors based on their hash value. In this study, they implemented range-partitioning which is assigning a contiguous group of intervals to the same processor. They chose range-based partitioning because they expected range-based partitioning to work best due to implicitly clustered distribution of the records. Actually, we implemented two variations of range-partitioning scheme in our study which will be explained in detail in Chapter 4. Leung and Muntz [29] tried assigning time intervals to processors based on either start or end times. However, since both choices introduce asymmetry, they both caused imbalance.

The next study on query performance on temporal collections is done by Ramaswamy [37]. They implemented an extended version of the Time List [39]. Instead of complex data structures, they preferred a simple B+ tree. A temporal object is characterized by a key and a time interval. They divided the data into window lists by partitioning the time interval. Since there will be versions spanning more than one time interval, some of these versions will be redundantly stored into different window lists. The temporal objects are inserted into a B+ tree based on the object's key and the starting point of its window list. This structure works well for pure time-slice and pure key-timeslice queries but updating the structure is troublesome. Construction of window lists is very similar to our time-based partitioning scheme.

A relatively recent study on speeding up query processing is V2 temporal text-index proposed by Nørvåg and Nybø [35]. In this model, a version of a document is characterized by its version ID (VID). A document-to-page mapping is stored in a B-tree based index. All versions are indexed in an efficient way such that average storage cost per posting is close to two bytes. Another index called VIDPI is used for version to validity period (start time - end time) mapping. While answering of a temporal query, all versions containing query terms are retrieved from the database. Then, the versions with intersecting time intervals are selected. This approach performs well in general since the version to validity time mapping is assumed to be in memory.

Another research on reducing query response time in temporally versioned document collections proposed interval based temporal text-index (ITTX) [35]. In this work, authors improved V2 temporal text-index. Recall that even if a term t_i exists in all versions of a document, a different posting is stored in the V2 index. Therefore, in ITTX, they decided to index the data in the following format: (w, DID, $DVID_i$, $DVID_j$, t_s, t_e) where w is the score of the version, DID is the document id, $DVID_i$ is the version ID of the document's first version that contains t_i, $DVID_j$ is the version ID of the last version that contains t_i, t_s is the start time of $DVID_i$ and t_e is the end time of the $DVID_j$. They made further improvements by removing $DVID_i$ and $DVID_j$ from the postings. They called this improved index file version as ITTX/ND. As the document collection gets bigger, V2 is no longer usable because version to validity time mapping would not fit into the memory. However, ITTX and ITTX/ND could still be used on tremendously gigantic document collections. Moreover, ITTX and ITTX/ND perform better when updating index files or answering temporal queries. As expected, they also generate smaller indexes compared to V2.

The last study that we will mention is the most recent work that we know of and also the best performing method of the last four methods. DyST, a dynamic and scalable temporal text index, is a two-level indexing scheme where the first level is called ITTX/ND and the second level is called temporal posting subindex (TPI) [32]. When a version is inserted into the database, it is also inserted into the first level, ITTX/ND. When a term's posting list size exceeds a threshold, it is migrated to the second level, TPI. To put it simply, frequent terms are moved to TPI, while rare and moderately-frequent terms are kept in ITTX/ND. Since ITTX/ND is explained in the previous paragraph, we will only explain TPI here. TPI search tree is a variant of monotonic B+ tree. It uses four times as much space as inverted index but it reduces the cost of snapshot search considerably. The trade-off between the cost of a four times bigger index file and query performance is open to question.

2.1.4 Both Space and Query Performance Improving Applications

In this section, we are going to summarize three studies which exploit both index size reduction and query performance optimization on temporal document collections.

First, Tsotras and Kangelaris's snapshot index, which uses $O(n/b)$ space and $O(logb(n) + |s(t)/b|)$ I/O for query time, will be examined [43]. Note that n is the number of changes on temporal objects, b is the I/O transfer size and $s(t)$ denotes the size of the answer set. They presented an access method for timeslice queries that reconstructs the past states of time evolving object collections. They represented time as a set of integers which improved performance while worsening accuracy. They kept temporal objects in blocks. They defined usefulness 1, usefulness 2 for each block and a usefulness parameter α common to all blocks. Usefulness 1 is determined by the temporal object with the minimum start time in that block. Similarly, usefulness 2 is determined by the temporal object with the maximum start time in that block. The usefulness parameter α is used to tune the number of documents in usefulness part 2, in order to trade off between space consumption and query performance. When a block is full, its usefulness period ends and the alive objects which are not yet to be deleted, are copied to the next block. In order to experiment on this method, they created a synthetic temporal dataset. The results also proved that space utilization and query performance could be tuned by playing with the usefulness parameter α.

Second, Chien et al. proposed three indexing schemes for supporting complex temporal XML queries [12]. In their earlier studies they proposed Usefulness Based Copy Control (UBCC) [13, 14]. UBCC is a storage scheme based on usefulness. Usefulness of a page is the percentage of the size of the undeleted objects in that page at a certain time. When the usefulness drops below a usefulness parameter which is set to 0.5 for the experiments, all the versions of this page are copied to another page. In their latest work, they proposed three indexing schemes to improve UBCC: single Multiversion B-tree, UBCC with Multiversion B-Tree and UBCC with Multiversion R-tree. They also proposed SPaR to improve the versioning schemes of their earlier works in order to support complex temporal queries. SPaR keeps start times and end times of versions. They store versions as a combination of objects. During query processing, instead of searching among versions, they search in objects and call this partial version retrieval. This approach also allows querying only a portion of a version such as search in the conclusion or search from chapter 1 to 4. The experiments are performed on a synthetic document collection with 1000 revisions that consist of 10,000 objects where 10% of each revision is modified to generate the next one. For both check-in time and index sizes, third scheme performed best by far.

The last and most recent study we will mention in this section is Berberich et. al.'s time-travel text search [2, 3, 4]. They modify the inverted file to include the valid time interval of each posting in addition the version id and weight. In this study, pure timeslice queries are evaluated. In order to utilize space consumption, they propose temporal coalescing which merges the scores of a posting in different versions of a document if the variation in the score is within an acceptable range. This approach reduces the index size up to 60% but accuracy is compromised [3]. However, they claim that top 10 query results are not noticeably affected. Moreover, since searching in smaller index file will be faster, it indirectly improves query performance.

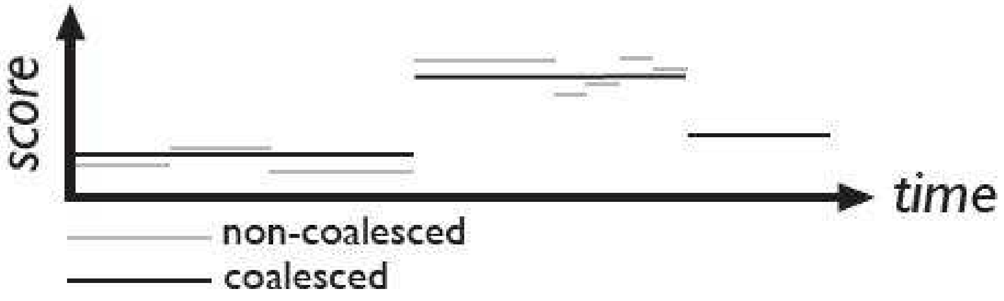

Figure 2.1: Approximate temporal coalescing [2].

The posting list I_i of term t_i is depicted in Figure 2.1 [3].The horizontal line segments represent each posting in I_i. If two postings p_j and p_{j+1} such that $p_j.v$ and $p_{j+1}.v$ are two different versions of the same document and $p_j.e = p_{j+1}.s$, then, the relative error is calculated as $err_{rel}(p_j, p_{j+1}) = |p_j.s - p_{j+1}.s| \ / \ |p_j|$. If $err_{rel}(p_j,\ p_{j+1}) \leq \epsilon$, then p_j and p_{j+1} are coalesced.

They also propose sublist materialization technique which increases query performance by time-based partitioning the inverted index. However, this technique increases the index size considerably. They use a modified version of OKAPI to calculate scores of query results. Below, in Figure 2.2 [3], sublist materialization of three documents are shown. Documents d_1, d_2 and d_3 have four, three and three versions, respectively. Although Berberich, et.al, proposed a time-based partitioning, they did not restrict the granularity of partitioning. If the time axis is divided into largest number of segments, space consumption will be huge but optimal query performance will be gained. If $\forall t_i : i = 1, 2, .., 9$, a partition P_i is created, then $P_1 = \{v_1,\ v_5,\ v_8\}$, $P_2 = \{v_2,\ v_5,\ v_8\}$, ..., $P_9 = \{v_7,\ v_{10}\}$. In the case of a time-based partitioned inverted indexes with maximum granularity, only the required number of postings will be fetched from the disk. Therefore, query response time will be maximized at the expense of enormous space consumption due to replicated versions in different inverted index partitions.

Additionally, they proposed performance and space guarantee approaches. For the experiments, they used Wikipedia and UKGOV datasets. They generated synthetic queries by selecting a set of keywords and a random date from each

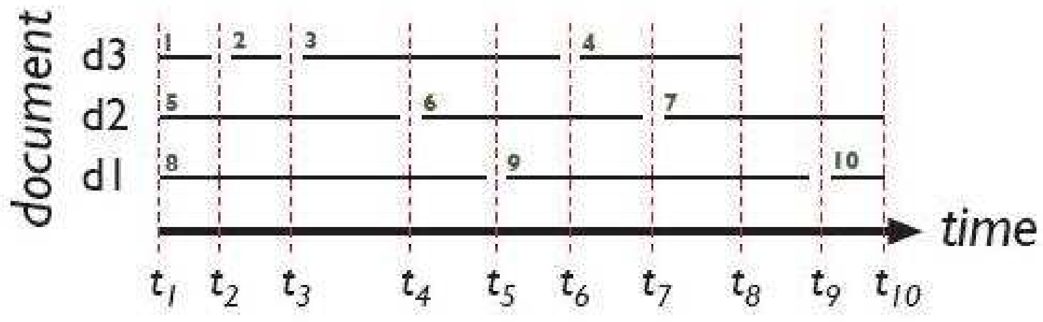
document
d3
d2
d1
1
2
3
4
5
6
7
8
9
10
time
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10}

Figure 2.2: Sublist materialization [2].

week in the dataset. By allowing a 10% reduction in query performance, they improved space consumption by an order of magnitude. Similarly, by allowing three times more space consumption, they improved query performance by an order of magnitude. Although temporal coalescing results drops accuracy of the results, top 10 results are claimed to not be affected. While comparing the accuracy of the resulting versions, they used Kendall's method. When the tolerance is set to 5%, for top-100 results, Kendall's τ is calculated as 0.85. Although, they consider this result to reveal strong agreement in the order of results, it is open to question, particularly if the user is expecting high accuracy.

Chapter 3

Implementation

In shared-nothing parallel text retrieval systems, in order to process user queries, generation of an index file, partitioning it among processors, obtaining document scores and returning the desired number of results to the user in sorted order are required. In this chapter, we will explain generation of inverted indexes for temporal document collections, query processing, ranking the resulting documents and parallel text retrieval implementation details, respectively.

3.1 Inverted Indexes for Temporal Document Collections

The main functionality of a text retrieval system is processing user queries and providing a set of relevant documents to the user. For extremely small document collections, these queries could be answered by simply passing over whole text sequentially [41]. However, as the collection size grows, full text search turns to be infeasible and an intermediary representation of the collection (i.e., indexing mechanism) becomes necessary. In traditional text retrieval systems, inverted indexes are almost always preferred to other index structures such as signature files and suffix arrays [40]. Therefore, we used a modified version of inverted indexes that is applicable to temporal document collections [37].

First, let us describe inverted index file structure. An inverted index consists of a set of inverted lists $\mathcal{L} = \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_T\}$, where $T = |\mathcal{T}|$ is the vocabulary size $\mathcal{T}$ of the indexed document collection $\mathcal{V}$, and an pointer to the heads of the inverted lists. Since our collection contains different versions of a set of documents, we treat each version as a separate document and index them one by one. The pointers to the inverted lists are kept in the index which is usually small enough to fit into the main memory. However, inverted lists are stored on the disk increasing access cost. An inverted list $\mathcal{I}_i \in \mathcal{L}$ is associated with a term $t_i \in \mathcal{T}$ and contains entries (called postings) for the documents containing the term t_i. In a traditional inverted index structure, a posting $p \in \mathcal{I}_i$ has a version id field $p.v = j$ and a weight field $p.w = w(t_i, v_j)$ for a version v_j in which term t_i appears where the value of $w(t_i, v_j)$ shows the degree of relevance between t_i and v_j using some metric.

However, to be able to use inverted indexes on temporally versioned document collections, we extended posting structure such that p also contains start time field $p.s = s_j$ and end time field $p.e = e_j$ of the version v_j [37].

3.2 Query Processing: The Vector-Space Model

While processing a query, picking the related documents and presenting them to the user in the order of document's similarity to the query is important. In order to achieve this goal, many models such as boolean, vector-space, fuzzy set and probabilistic models are proposed [40]. The vector-space model is the most widely accepted model due to its simplicity, robustness, speed and ability to catch partial matches [42].

Below, the cosine similarity $\text{sim}(\mathcal{Q}, v_j)$ between a query $= \{t_{q_1}, t_{q_2}, \ldots, t_{q_Q}\}$ of size Q and a document v_j which is used in the vector-space model is given

$$\text{sim}(\mathcal{Q}, v_j) = \frac{\sum_{i=1}^{Q} w(t_{q_i}, v_j)}{\sqrt{\sum_{i=1}^{Q} w(t_{q_i}, v_j)^2}}, \tag{3.1}$$

assuming all query terms have equal importance. In order to compute the weight $w(t_i, v_j)$ of a term t_i in a document v_j, the *tf-idf* (term frequency- inverse document frequency) weighting scheme [42] is usually used.

$$w(t_i, v_j) = \frac{f(t_i, v_j)}{\sqrt{|v_j|}} \times \ln \frac{V}{f(t_i)}, \tag{3.2}$$

where $f(t_i, v_j)$ is the frequency of term t_i in version v_j, $|v_j|$ is the total number of terms in v_j, $f(t_i)$ is the number of versions containing t_i, and V is the number of versions in the collection. In this study, the *tf-idf* weighting scheme is used together with the vector-space model [40].

In a traditional sequential text retrieval system, a query is processed in several stage. In order to process a a user query $\mathcal{Q} = \{t_{q1}, t_{q2}, \ldots, t_{qQ}\}$, each query term t_{q_i} is processed one by one as follows. First, inverted list of term t_{q_i} is $\mathcal{I}_{q_i}$ is fetched from the disk. All postings in $\mathcal{I}_{q_i}$ are traversed, and the weight $p.w$ in each posting $p \in \mathcal{I}_{q_i}$ is added to the score accumulator for document $p.v$ if the time interval of the query intersects with the validity time interval of the posting being processed. Once all inverted lists are processed, versions are sorted by using a ranking algorithm, and the versions with the highest scores are returned to the user.

3.3 Ranking Algorithms

The primary task of a temporal search engine is after receiving a temporal web query, present relevant versions to the user in sorted order by version to query similarities. As explained in the previous section, similarity calculations are performed using cosine similarity with the vector space model. After calculating the similarity scores of versions, a temporal search engine sorts the scores and displays top s results to the user. In this study, we also addressed efficient sorting of the resultant versions.

There are two different ranking techniques used in this study: randomized-select based ranking and skip-list & min-heap based ranking. Both of these techniques first selects most relevant s versions and then sorts these s versions by quick sort. The difference comes in the selection phase. Compared to the naive case where all version scores are sorted and then top s versions are displayed, this approach increases the query response time by an order of magnitude [9]. This improvement emphasizes the importance of ranking algorithms pointing to the huge cost introduced by sorting. See Figure 3.1 for the pseudocode of TOs algorithms and Figure 3.2 for pseudocode of TOd algorithms [8].

```
TO-s($\mathcal{Q}$, $\mathcal{A}$)
    for each accumulator $a_i \in \mathcal{A}$ do
        INITIALIZE $a_i$ as $a_i.v = i$ and $a_i.s = 0$
    for each query term $t_{q_j} \in \mathcal{Q}$ do
        for each posting $p \in \mathcal{I}_{q_j}$ do
            if $[p.\mathrm{s}, p.\mathrm{e}] \cap [q_j.\mathrm{s}, q_j.\mathrm{e}] \neq \emptyset$ then
                UPDATE $a_{p.v}.s$ as $a_{p.v}.s + p.w$
    $\mathcal{S}_{\mathrm{top}} = \emptyset$
    INSERT the accumulators having the top $s$ scores into $\mathcal{S}_{\mathrm{top}}$
    SORT the accumulators in $\mathcal{S}_{\mathrm{top}}$ in decreasing order of their scores
    RETURN $\mathcal{S}_{\mathrm{top}}$
```

Figure 3.1: The algorithm for TO-s implementations.

Cambazoglu and Aykanat [9] divided the phases of query processing implementation into 5 parts: *creation, update, extraction, selection* and *sorting.* In the creation phase, each version v_i is associated with an accumulator a_i. Based on the application, space required for accumulators might be dynamically allocated. In the update phase, when the posting p is being processed, the score of the corresponding accumulator is updated, i.e., $a_i.\mathrm{s} = a_i.\mathrm{s} + p.w$ where $p.v = i$. The extraction phase selects nonzero accumulators, i.e., $a_i.\mathrm{s} \neq 0$. In the selection phase, the accumulators with top s scores are selected. Finally, in the sorting phase, the selected s scores are sorted in decreasing order.

```
TO-D($\mathcal{Q}$, $\mathcal{V}$)
    for each query term $t_{q_i} \in \mathcal{Q}$ do
        for each posting $p \in \mathcal{I}_{q_i}$ do
            if $[p.\mathrm{s},\ p.\mathrm{e}] \cap [q_i.\mathrm{s},\ q_i.\mathrm{e}] \neq \emptyset$ then
                if $\exists$ an accumulator $a \in \mathcal{V}$ with $a.v = p.v$ then
                    UPDATE $a.s$ as $a.s + p.w$
                else
                    ALLOCATE a new accumulator $a$
                    INITIALIZE $a$ as $a.v = p.v$ and $a.s = p.w$
                    $\mathcal{V} = \mathcal{V} \cup \{a\}$
    $\mathcal{S}_{\mathrm{top}} = \emptyset$
    for each $a \in \mathcal{D}$ do
        SELECT($\mathcal{S}_{\mathrm{top}}$, $a$)
    SORT the accumulators in $\mathcal{S}_{\mathrm{top}}$ in decreasing order of their scores
    RETURN $\mathcal{S}_{\mathrm{top}}$

SELECT($\mathcal{S}$, $a$)
    if $|\mathcal{S}| < s$ then
        $\mathcal{S} = \mathcal{S} \cup \{a\}$
    else
        LOCATE $a_{\mathrm{smin}}$, the accumulator with the minimum score in $\mathcal{S}$
        if $a.s > a_{\mathrm{smin}}.s$ then
            $\mathcal{S} = (\mathcal{S} - \{a_{\mathrm{smin}}\}) \cup \{a\}$
```

Figure 3.2: The algorithm for TO-d implementations.

We selected term-ordered static method 4 (TOs4) which is called randomized-based ranking and term-ordered dynamic method 3 (TOd3) which is called skip-list based ranking throughout this study. Since we took term-based partitioning scheme to compare with our time-based partitioning proposal, we decided to select one static and one dynamic method among the well performing term ordered methods. According to the results and complexity analysis presented by Cambazoglu and Aykanat [9], TOs4 and TOd3 are proper candidates for queries with smaller number of keywords. Please note that our query sets consist of lesser number of keywords compared to their query sets. Details regarding the query sets will be given in Section 5.3.

In term-ordered processing, each term's posting list is processed one by one.

There is an accumulator associated with each version in the collection. In implementations with static accumulator allocation (TOs), an accumulator for each version is allocated statically. However, in implementations where accumulators are dynamically allocated (TOd), at most e accumulators are allocated where e is the length of the posting list being processed. If the number of versions in the collection is huge, TOd saves significant amount of space.

Other than the allocation of accumulators, the main difference between Tos4 and TOd3 is the selection phase. First, let us explain the details of randomized-select based ranking algorithm (TOs4). In the implementation of this method, median-of-medians algorithm could be used for selection due to its linear time worst case complexity. However, although randomized-selection algorithm has expected linear time complexity, Cambazoglu and Aykanat preferred implementing randomized-selection due to its run-time efficiency in practice. After the accumulator with the s^{th} biggest score (a_s) is found by randomized select in expected linear time, we pass over the accumulators array and extract all accumulators having a bigger score than $a_s.s$. Then, these s accumulators are sorted in decreasing order in *O(slgs)* time. As a result, the overall complexity of randomized-select based ranking algorithm is *O(D + slgs)* [9].

Second, let us explain the working principle of skip-list & min-heap based ranking algorithm, TOd3. In this method, Cambazoglu and Aykanat used a skip list to store accumulators. Their skip list implementation could be considered as a variation of linked list with forward and backward pointers for several levels on the top. Note that in order to have a logarithmic search complexity, the number of levels at a time is probabilistically kept proportional to $lg(k)$ where k is the number of elements in the skip-list at that time instant. Although skip lists have bad worst-case complexities, due to good expected-time complexities for search and insertion, they perform well in practice.

During the execution of TOd3, when a posting p such that $p.v = i$ is processed, accumulator a_i needs to be updated. If a_i is already inserted into the skip list, then its score is updated such that a_i.s = a_i.s + $p.w$. Otherwise, a_i is inserted into the skip list with a_i.s = $p.w$. The accumulators in the skip list are ordered by their version ids. Therefore, the proper place for inserting a_i has to be found first. Searching for a_i's place has logarithmic time complexity and insertion takes constant time due to linked-list like data structure. Once all posting lists are processed, the accumulators in the skip list are inserted into a min-heap of size s. Recall that s is the maximum number of desired versions to be returned to the user. Then all accumulators in the skip list are inserted into the min-heap one by one. If an accumulator a_i has a greater score compared to the root of the min-heap, the root is extracted and a_i is inserted into the heap. Finally, the s versions in the min-heap are returned to the user.

Although, selection and sorting has O($s\ lg\ s$) time complexity, because of the high worst case complexity of update process, the overall complexity of TOd3 becomes O($u\ lg\ e$), where u is the total number of postings in all query terms and e is the total number of distinct postings in all query terms.

3.4 Parallel Text Retrieval

Skynet parallel text retrieval system which is created by Cambazoglu [8] and extended by Tokuc [42], is further enhanced in order support search on temporal document collections. For this purpose, a master-client type architecture, which is named as ABC-server, is implemented in C using LAM/MPI [7]. Currently, ABC-server runs on a 48-node Beowulf PC cluster, located in the Computer Engineering Department of Bilkent University.

In parallel architectures, there are two main categories for query processing applications: Inter-query parallelism and Intra-query parallelism [11]. In Inter-query parallelism, after collecting a bunch of queries, CB selects the ISs that will be solely responsible for answering each query. In other words, each IS acts as an independent search engine in this schema. However, in Intra-query parallelism, a single query may be answered by several ISs. In this study, we implemented Intra-query parallelism due to its tendency to exploit parallelism better.

As seen in Figure 3.3, in this architecture, there is a master called Central

Broker (CB), a Query Submitter (QS) to simulate users and K Index Servers (IS) responsible for processing the query. CB collects the incoming user queries represented by QS, and redirects them to the ISs which are nodes of the PC cluster. The ISs generate partial answer sets (PASs) to the received queries, using the their partition of the inverted index stored in their local disk. The generated PASs are then merged into a global answer set by the CB, to be sent to the user.

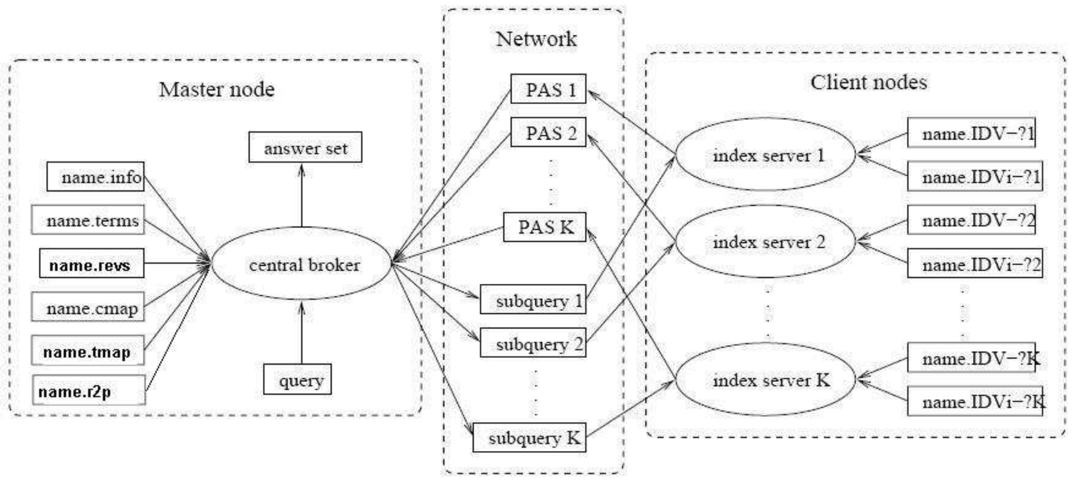

Figure 3.3: The architecture of the Skynet parallel text retrieval system [8].

When the program starts, CB needs to initialize some of its data structures. First, CB reads the .info file to set the number of versions, documents, terms in the dataset. Then, after reading .terms file, it creates a *trie* in which the terms and term ids are kept. When a user, which is represented by QS, submits a query q_s, the id of a query term is accessed in O(w) time where w is the length of that term. After reading .tmap or .cmap files, an array to store the version to document mapping is created. Afterwards, by reading .r2p input file, CB allocates another array to store version to document mapping. Finally, after creating receive buffers and allocating memory to keep memory, time and communication statistics, CB initializes a TCP port over which the queries will be submitted. Index servers also initialize their send buffers, accumulators and other data structures to store statistical information. Both CB and the ISs use first-in first-out (FIFO) queues for processing user queries.

CB inserts two different types of items into its queue: queries and PASs. When CB receives a query from a user, it inserts the query into the queue as a query type item. After CB dequeues a query type item from the queue and processes it, it identifies the responsible ISs for that query and records the number of ISs that a part of this query (subquery) will be sent to. Then, a packet that consists of subquery terms as well as query id and the query time interval is sent to the responsible index servers. Please note that in time-based partitioning scheme, subqueries refer to queries and whole query is sent to all responsible ISs.

An IS periodically checks for incoming subqueries from CB. If a subquery is received, it is enqueued as a subquery item to the queue of that IS. After the subquery is dequeued from the queue, the index server inspects if any subquery term reside in its local term list. If none of the terms appears in its local index, it replies with an empty packet to the central broker. Otherwise, corresponding posting lists are read and related accumulators' scores are updates. Each IS has a static array that will contain an accumulator for each version in the collection. Another highly deployed technique for storing accumulator arrays is the use of a dynamic accumulator array where an accumulator is inserted only if the weight of the corresponding posting is larger than a determined threshold (accumulator limiting).

In the ABC-server implementation, the number of accumulators to be stored is not limited but in the term-based partitioning case, the number of accumulators to be sent through the network are limited since the network becomes the bottleneck. In term-based partitioning scheme, this limit is set to 1% of the total number of versions in the system and in time-based partitioning scheme, it is set to the maximum number of versions requested by the user for display. Be aware that accumulator restriction decreases the accuracy of term-based partitioning scheme but it does not affect the accuracy of time-based partitioning scheme. In other words, as long as accuracy is concerned, time-based partitioning is superior to term-based partitioning.

When processing the subquery is finished, IS selects top s accumulators from the accumulator array by using either expected linear time randomized-select based or skip-list based ranking algorithm. Selected accumulators are sorted by version ids and then, the prepared partial answer set is copied to the sending buffer. Since immediate send is used, IS waits until there exists no ongoing send operation to guarantee that the correct buffer is being sent. The static accumulator array is cleared for future use and contents of the sending buffer is sent to central broker by non-blocking send operation (Isend).

CB periodically checks for incoming packets from ISs. When a PAS is received from IS_i, the content is inserted into CB's queue to be processed later. When a PAS is dequeued from the queue, CB merges it with other PASs received from the rest of the index servers which are responsible for that query too. The merge operation assumes sorted PASs by version ids.

After the merge operation, CB compares the number of sent subqueries for query q_i and the number of PASs received for q_i. If all expected PASs are received, the merge operation is carried to generate the final answer set. Top s accumulators are extracted from the final answer set by randomized-selection method and presented to the user.

Chapter 4

Inverted Index Partitioning

Inverted index partitioning has a crucial effect on the efficiency of a parallel text retrieval system such that the organization of the inverted index determines both disk access and network cost of the system. Therefore, the main focus of this thesis will be efficient inverted index partitioning on a shared-nothing architecture [11]. We proposed a time-based partitioning scheme for efficiently answering time-key queries explained in Section 2.1.3. To our knowledge, this is the first study that addresses answering this query type using a parallel architecture, on temporal web data.

In this chapter, first, we will explain a traditional partitioning scheme adapted to temporal web data: term-based partitioning. Later, we will explain the details of proposed time-based inverted index partitioning schema with two different document to processor mapping strategies. Please refer to Chapter 5 for the detailed performance analysis of these partitioning methods.

4.1 Term-Based Partitioning

In a traditional term-based partitioning scheme, terms are alphabetically sorted and then distributed to the processors in a round-robin fashion. The partitioning info is kept in a mapping file having .cmap extension. Term-based partitioning is also called column-based partitioning because the relationship between versions and the terms they contain can also be displayed as a matrix where rows and columns represent versions and terms, respectively.

In a shared-nothing parallel architecture, the inverted index should be partitioned taking size balance into account. The load imbalance between the IS with the largest inverted index partition and the IS with the smallest inverted index partition should be kept at a minimum possible level. If there are $|\mathcal{P}|$ posting entries in the global inverted index, each index server S_j in the set $\mathcal{S} = \{S_1, S_2, \ldots, S_K\}$ of K index servers should keep an approximately equal amount of posting entries as shown by

$$\mathrm{SLoad}(S_j) \simeq \frac{|\mathcal{P}|}{K}, \qquad \text{for } 1 \leq j \leq K, \tag{4.1}$$

where $\mathrm{SLoad}(S_j)$ is the storage load of index server S_j [10].

Round-robin distribution results in an acceptable load imbalance. Therefore, we used round-robin distribution in our implementation. In term-based distribution, each index server is responsible with a set of terms. Since the query terms will be partitioned into subqueries by the central broker, intra-query parallelism is achieved and query processing task could be divided into smaller units. Moreover, as different queries employ different index servers, the number of idle processors at a time will be smaller leading to a high level of concurrency and maximized processor utilization. Although, term-based distribution minimizes the number of disk accesses, due to the transfer of partial answer sets, redundant communication takes place. If the network is the bottleneck in the parallel system, term-based distribution is expected not to perform well [8]. In Table 4.1, term-based distribution of a toy dataset with 6 versions and 8 terms, on 2 processors is depicted.

Table 4.1: Round robin term-based distribution of the toy dataset on two processors

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
v_0	S_0	S_1			S_0			
v_1		S_1	S_0		S_0	S_1		
v_2	S_0		S_0	S_1				
v_3	S_0	S_1		S_1	S_0		S_0	
v_4			S_0	S_1				S_1
v_5		S_0			S_1	S_0		

4.2 Time-Based Partitioning

In the proposed time-based partitioning scheme, inverted indexes are partitioned by partitioning the time axis. In order to partition an inverted index, first we generate a mapping file that contains the points on the time axis to be used for partitioning. We proposed two generation schemes that will be explained in detail in the following subsections.

Once the mapping file is generated such that T = $\{t_1, t_2, ..., t_{K+1}\}$ such that t_i is the i^{th} chosen instant on the time axis and K is the number of resultant partitions. S_i will have the version v_j if and only if their time intervals intersect, such that: $t_i < v_j.e$ & $t_{i+1} > v_j.s$. This equation is the negation of the case where the two intervals does not intersect. If the start time of the former is greater than the end time of the latter or the end time of the former is smaller than the start time of the latter, their time intervals obviously do not intersect. Since the validity time interval of a version might intersect with the time interval of more than one processors, redundant information is allowed. Actually, this strategy is very similar to Berberich et al.'s [4] sublist materialization. However, the mapping files introduce a difference that significantly affects the performance.

When, central broker receives a query q_i from users, it checks the time interval of the query and sends the whole query to the processors whose time intervals intersect with the validity time interval of q_i.

4.2.1 Time-Balanced Time-Based Partitioning

Time-balanced time-based partitioning scheme, aims dividing the time axis into equal intervals. If there are K processors in the parallel architecture, the time axis starting from the minimum start time of all versions to maximum start time of all versions will be divided into K parts. As the ending point of the time axis, we do not choose the current time because this will lead to a worse load balance since there will be no versions inserted into the system after the last crawl date. In our experiments, we had a static dataset where no modifications are allowed after the creation of the inverted index. For different applications that require continuous insertion, deletion and updates, the upper bound of the time axis should be set to a higher value.

After the lower and upper bounds of the time axis are determined, time axis is simply divided into K equal parts such that:

$$t_{K+1} - t_K = t_K - t_{K-1} = ... = t_2 - t_1 \tag{4.2}$$

If the distribution of query time intervals is even, meaning that the probabilities of intersection with the processors' time intervals were to be equal, this partitioning scheme would perform well. However, this is not the case in practice. The number of versions in a unit time interval increases as the time interval gets close to now. For this reason, the queries are expected to be skewed towards now. Moreover, due to the uneven distribution of versions among processors, a significant load imbalance is observed. During the time-balanced time-based partitioning on 8 processors, the *imbalance* values which are calculated by the Equation 4.3 are observed to be %126 and %263 in Wikipedia and Wiktionary datasets, respectively. Processor idle time is expected to increase with *imbalance* causing a huge decrease in query performance. Therefore, another time-based partitioning scheme is proposed in Section 4.2.2.

$$imbalance = \frac{max(size(\mathcal{L})) - avg(size(\mathcal{L}))}{avg(size(\mathcal{L}))} \times 100 \tag{4.3}$$

4.2.2 Size-Balanced Time-Based Partitioning

Assuming that the amount of queries coming from a time interval is proportional to the total size of the alive versions in that time interval, we proposed a heuristic that will roughly balance the inverted index files in each processor. First, we calculate the size of each index file if we were to create a partition for each week, without actually creating inverted index partitions. Then, by trying to balance the total sizes of the index files assigned to each processor, we assign consecutive weeks to processors.

In order to size-wise balance the inverted index partitions, starting from the most recent index file we start assigning each index file's week to a processor. Once the threshold, which is calculated as the total of inverted index partition sizes divided by K, is exceeded, we start assigning weeks to another processor. If at least one processor remains empty, we gradually decrease the threshold until no processor remains empty. If we do not play with the threshold, empty processors might remain especially when the number of processors increase. This problem occurs because the total size of merged index partitions is not conserved. If we were to merge inverted index partition $\mathcal{L}_i$ with $\mathcal{L}_j$ to create a resultant inverted index $\mathcal{L}_R$, size($\mathcal{L}_R$) $\leq$ size($\mathcal{L}_i$) + size($\mathcal{L}_j$). This inequality might seem unclear at first glance but recall that if a posting p appears in both $\mathcal{L}_i$ and $\mathcal{L}_j$, then it will appear in $\mathcal{L}_R$ only once. Therefore, especially as the number of weeks in the dataset increase, the difference between size($\mathcal{L}_R$) and size($\mathcal{L}_i$) + size($\mathcal{L}_j$) gets bigger.

With the help of this scheme, the processor whose time interval is closest to the current date (now), S_K, will have the narrowest time interval, because the number of versions alive in this interval is expected to be the largest. Moreover, since the majority of the queries are expected to be close to now, S_K will be assigned more queries unless its time interval is kept shorter. As a result, this mapping strategy not only balances the inverted index sizes in each processor, but also probabilistically balances the number of queries assigned per processor. Table 4.2 displays the *imbalance* ratios which is calculated by Equation 4.3.

Table 4.2: Imbalance in size-balanced time-based partitioning

# of processors	Wikipedia	Wiktionary
2	21	1
4	25	13
8	44	26
16	52	34
24	51	32
32	51	40

The imbalance ratios of time-balanced and size-balanced time-based partitioning schemes differ by an order of magnitude particularly on Wiktionary dataset. Therefore, the rest of the experiments are carried using size-balanced partitioning scheme. The *imbalance* in Wiktionary dataset seem to be smaller due to its larger time span (2411 days) compared to Wikipedia's time span of 1619 days. Furthermore, *imbalance* increases with the number of processors since dividing the total number of days into more partitions is more difficult.

Chapter 5

Experiments

The hardware platform used in the experiments is a 48-node PC cluster interconnected by a Gigabit Ethernet switch. Each node has an Intel Pentium IV 3.0 GHz processor, 1 GB of RAM, and runs Mandrake Linux, version 10.1. [8].

In this chapter, we first explain how to create inverted index files step by step including the properties of resultant inverted index partitions. Then, we mention the Wikipedia and Wiktionary temporal datasets used in this study. Next, synthetic temporal web query generation and query properties are explained. Finally, we compare the performances of proposed time-based partitioning schemes with traditional term-based partitioning and further discuss their effects of ranking algorithms.

5.1 Preprocessing

Before the emergence of parallel text retrieval systems, search engines used to implement sequential text retrieval systems. Preprocessing stage of both systems are still quite similar. Below, we explain the creation of inverted indexes which are used for parallel query processing. We also present the properties of inverted index partitions which will have a significant effect on experimental results.

5.1.1 Corpus Creator

The primary task of the corpus creator is transforming given versioned document collection into a common and standard format. It removes whitespace characters and eliminates punctuation from the content. Then, case folding is applied on the remaining alphanumeric character groups and the resultant data is stored in ASCII format in a file with .corpus extension. Corpus creator not only makes corpus parser's task easier but also reduces the collection size. Furthermore, it provides robustness since for different versioned document collections, writing a different corpus creator will be sufficient.

5.1.2 Corpus Parser

After corpus creator converts the data collection into a common and standard format, corpus parser generates the document matrix and other informative files needed for query processing. The file with .corpus extension is given to corpus parser as an input file. Corpus Parser first eliminates the stop words provided by the Brown Corpus [30], as well as the terms that consist of a single letter and extremely long terms. Then, it generates the output files with .info, .terms, .revs, .r2p and .DV file extension.

The .info displays total term count, total distinct term count, total version count and total document count in the collection. The .terms file keeps track of each term t_i's name, id, the number of distinct documents t_i appears in, the number of distinct revisions t_i appears in and the total number of t_i's occurrence in the whole collection. The .revs file is responsible for storing an entry for each version where each entry consists of a version id v_i, total number of terms in v_i, total number of distinct terms in v_i, the exact (not discrete) time stamp of v_i and the title of v_i. Although, both of our real life data collections have one title for each document that is common to all of its versions, in order to not to restrict the application to one title per document, we stored the title of each version separately. The .r2p file is a binary file that stores version-to-document mapping. Since its size is negligibly small, we stored a document id for each version. It could be also stored in a compressed by rows (CBR) format to save from the space, though it is not worthy in our implementation.

Then document matrices with .DV extension are generated. The .DV file keeps a version vector for each version in the collection. A version vector VV_i consists of the *(term id, frequency)* pairs of all distinct terms that appears in v_i. The .DV file is an input file for the following module, that is inverted index creator.

5.1.3 Inverted Index Creator

For fast query processing, the collections are kept in the inverted index format where a posting list for each term is stored. Term t_i's posting list contains a *version id*, *start time*, *end time* and *weight* for each distinct version t_i appears in. The weight of a posting is calculated by the tf-idf weighting. Note that since the granularity of time information is too fine, we only used the year, month and day information to represent the time information. Time is denoted by an integer that counts the total number of days passed since a predefined base date which is guaranteed to be earlier than the insertion date of the very first version in the collection.

Inverted index creator reads .info, .revs, .r2p, .terms and .DV files to generate the IDV files. Due to memory constraints, the program might need to write a portion of the .IDV file and then read the .DV file again. A memory allocation constraint in the header file should be set considering the RAM of the machine that is responsible for inverted index creation. Moreover, by setting the global constants in the header file, the posting lists could be kept in version id, weight or start time order. The default choice is keeping the posting lists in increasing version id order. The version ids are unique among documents and the version ids of a document with a greater document id are guaranteed to be greater than the versions ids of a document with a smaller document id.

Inverted index creator generates two binary output files: .IDV file and .IDVi file. The former is used for storing posting lists while the latter is used for accessing each posting list in one disk access. The file with .IDVi extension keeps a record for each term t_i where each record consists of the distinct number of versions t_i appears in, the file ID of the IDV file that keeps t_i's posting list and a pointer to the location of t_i's posting list. Note that, inverted index creator could create more than one .IDV file since opening large files whose size exceed 2 GB is problematic. In order to be able to open large files, while compiling we set -D_FILE_OFFSET_BITS variable to 64 and used a single IDV file. Another solution is using fopen64() method instead of fopen() method.

5.2 Datasets

In the experiments, we used two real life temporal document collections: English Wikipedia [18] and English Wiktionary [19]. Both Wikipedia and Wiktionary are wikis meaning that their content is under the GNU Free Documentation Licence and the users are allowed to edit the content of their pages.

Table 5.1: Properties of Wikipedia and Wiktionary versioned document collections

	Wikipedia	Wiktionary
Size [GB]	7.88	28.92
Preprocessed size [GB]	7.32	23.62
DV size [GB]	1.60	5.0
IDV size [GB]	2.54	7.53
# of documents	109,690	720,660
# of versions	408,203	3,099,270
# of terms	580,071,671	1,899,914,288
# of distinct terms	1,454,684	3,192,040

Table 5.1 presents the properties of these document collections. Wikipedia and Wiktionary store not only the last modified version but also the history of each page. Therefore, they provide real-life temporal document collections which is exactly what we are looking for. The XML dump of Wikipedia is retrieved on

3rd of November 2007 and the XML dump of Wiktionary is retrieved on 20th of November 2007.

5.3 Synthetic Temporal Web Query Generation

This module uses .DV file to generate temporal queries. Each query consists of a set of keywords and a time interval, which is classified as time-key queries by Salzberg [39]. We define three query lengths such that short queries contain 1-3 keywords, medium queries contain 4-6 keywords and long queries contain 7-10 keywords. We generate three different query files for each query type. Each file contains 1500 queries.

In order to select proper terms from the .DV file, query generator needs to find a version containing sufficient number of candidate terms. Therefore, first, it calculates the proper number of terms that could be included in a query for each version. At this point, if a term frequency threshold is given, query generator will nominate a term t_i only if t_i's frequency is above the threshold. Similarly, if a version threshold is given, query generator will nominate a version v_i only if the percentage of v_i's term count in the collection exceeds the threshold. If no threshold is given, all the terms in each version are considered as proper candidates.

After the number of candidate terms in each version is calculated, the number of query terms for each query is decided using a random number generator. For example, for short queries, an integer between $[1, 3]$ is chosen for each one of 1500 queries. Afterwards, a version v_i that has sufficient number of candidate terms is randomly selected from the collection. Then, required number of terms are randomly selected among the candidate terms of v_i. This process is repeated 1500 times for each query.

Selecting a proper time interval for each query is another challenge. We generated two types of temporal queries based on their time intervals: 1X and 3X queries. For 1X queries, once v_i is selected for generating a query, the validity time interval of v_i is taken with the terms selected among v_i's terms. We believe that this will be a realistic approach since the queries are more likely to come from an interval that has more alive versions. For the 3X queries, we simply multiply v_i's time interval by 3. In other words, let $\delta = v_i.e$ - $v_i.s$, then, the query's time interval would be [v_i.s - δ, v_i.e + δ].

The minimum, maximum and average time interval lengths of each query type is shown in Table 5.2. The total time span of Wikipedia and Wiktionary corpora are 1619 and 2411 days, respectively. Also, recall that each query file consists of 1500 queries.

Table 5.2: Query properties

		1X [days]			3X [days]			
		min	max	avg	min	max	avg	# of terms
Wikipedia	Short	0	1112	108.47	0	1619	554.01	1-3
	Medium	0	1127	114.29	0	1619	531.71	4-6
	Long	0	877	103.65	0	1619	488.61	7-10
Wiktionary	Short	0	1760	104.74	0	2411	684.05	1-3
	Medium	0	873	85.77	0	2411	668.72	4-6
	Long	0	1432	88.98	0	2411	673.07	7-10

5.4 Experimental Results and Discussion

In this section, we present and analyze the results of the experiments on the effects of temporal inverted index partitioning schemes on parallel systems. The experiments are carried on Skynet, on 2, 4, 8, 16, 24 and 32 processors, in addition to a central broker and a query submitter. First, we compare size-balanced and time-balanced time-based partitioning performances. Afterwards, we compare the throughput, which is defined as the number of processed queries per second, for both term-based and time-based partitioning. Finally, we compare the average response time for time-based and term-based partitioning schemes and comment on the results.

5.4.1 Time-balanced vs. Size-Balanced Time-based Partitioning

The experiments comparing time-balanced with size-balanced time-based partitioning schemes are carried on 8 processors. The parallel text retrieval system processed short, medium and long queries of both 1X and 3X query types, on Wikipedia and Wiktionary temporal document collections. Queries are processed once by TOd3 (Skip-list based) ranking algorithm and once by using TOs4 (Randomized-select based algorithm).

As clearly seen from Figures 5.1 and 5.2, size-balanced time-based partitioning works better for all cases. This is and expected result since size-balanced partitioning tends to balance the size of the inverted index partition at each processor. While balancing the inverted index size, it also indirectly and statistically balances the number of queries assigned per processor since the more documents are the more queries are received. The number of queries served at each processor is an indicator of the work load per processor. Since size-balanced partitioning is expected to distribute the data and tasks more evenly, it is expected to perform better. Therefore, for other experiments we only used size-balanced time-based partitioning.

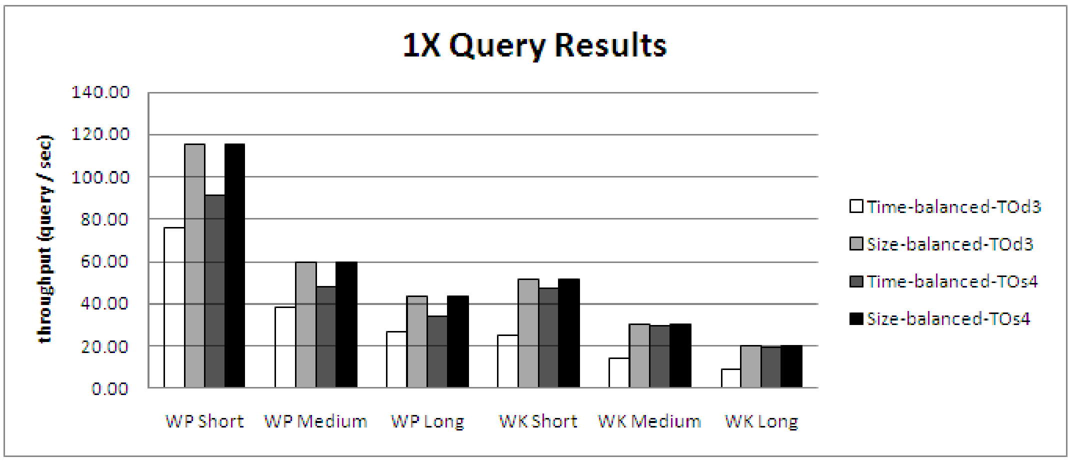

Figure 5.1: Time-balanced vs Size-balanced throughput comparison for 1X queries.

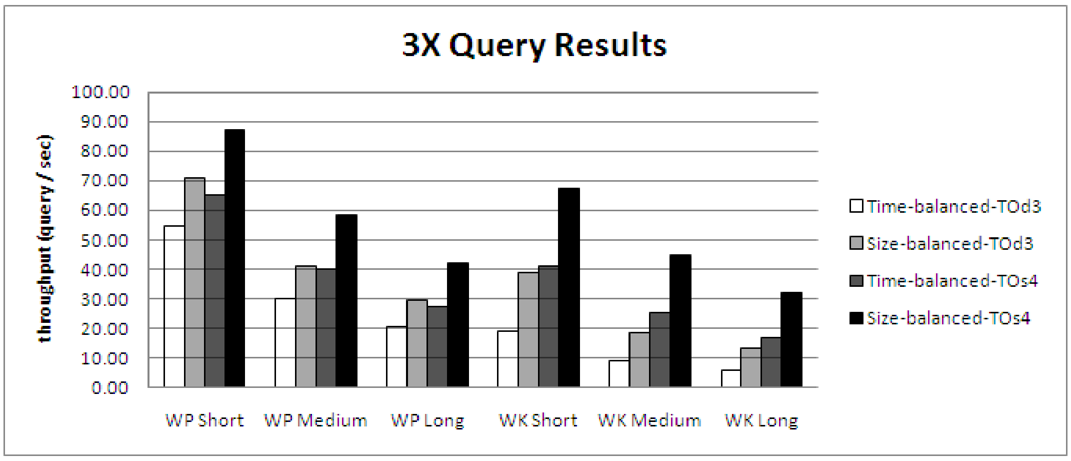

Figure 5.2: Time-balanced vs Size-balanced throughput comparison for 3X queries.

5.4.2 Term-based vs. Time-based comparison considering throughput

Figures 5.3 to 5.5 shown below display the results of processing 1X type queries on Wikipedia temporal document collection. Since we have three query sets with 1-3, 4-6 and 7-10 query terms for each query, this experiment is repeated three times on 2 to 32 processors. All these figures suggest that term-based partitioning performs better than time-based partitioning on Wikipedia corpus. Moreover, TOs4 seems to work better than TOd3 almost always in these three graphs with the exception of short query processing results of 24 and 32 processors.

For queries with a longer time interval, term-based partitioning is expected to perform better. Similarly, for queries with many terms, time-based partitioning scheme is expected to produce higher throughput rates (query/sec). The experiments with short queries meet the expectations and reveal that term-based partitioning works better in this case. However, in medium and long queries, term-based partitioning still performs better unexpectedly. The main reason behind this result is suspected to be related with the query properties and corpus properties in Tables 5.1 and 5.2. Although the average time interval per query barely exceeds 100 days in both Wikipedia and Wiktionary queries, the time spanned by Wiktionary is 50% longer than Wikipedia. Therefore,in time-based

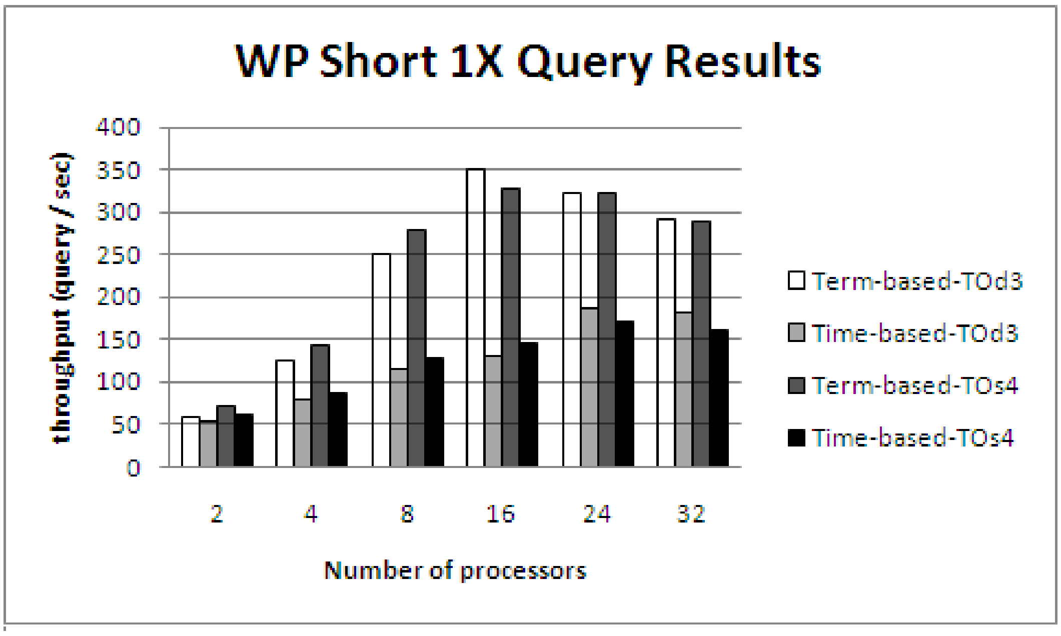

Figure 5.3: Term-based vs. Time-based partitioning on Wikipedia for Short 1X queries.

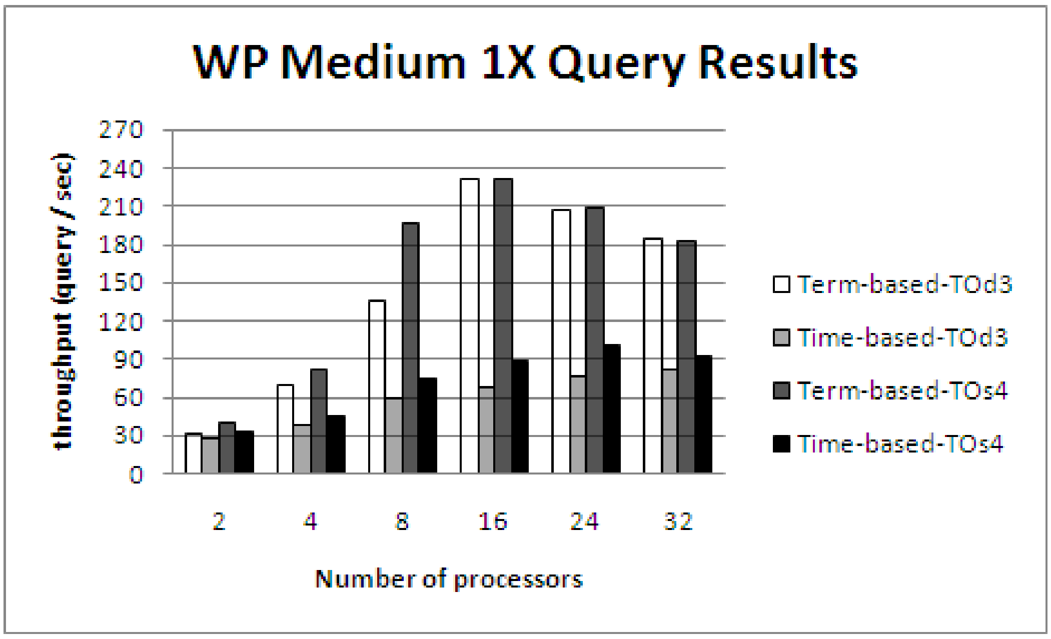

Figure 5.4: Term-based vs. Time-based partitioning on Wikipedia for Medium 1X queries.

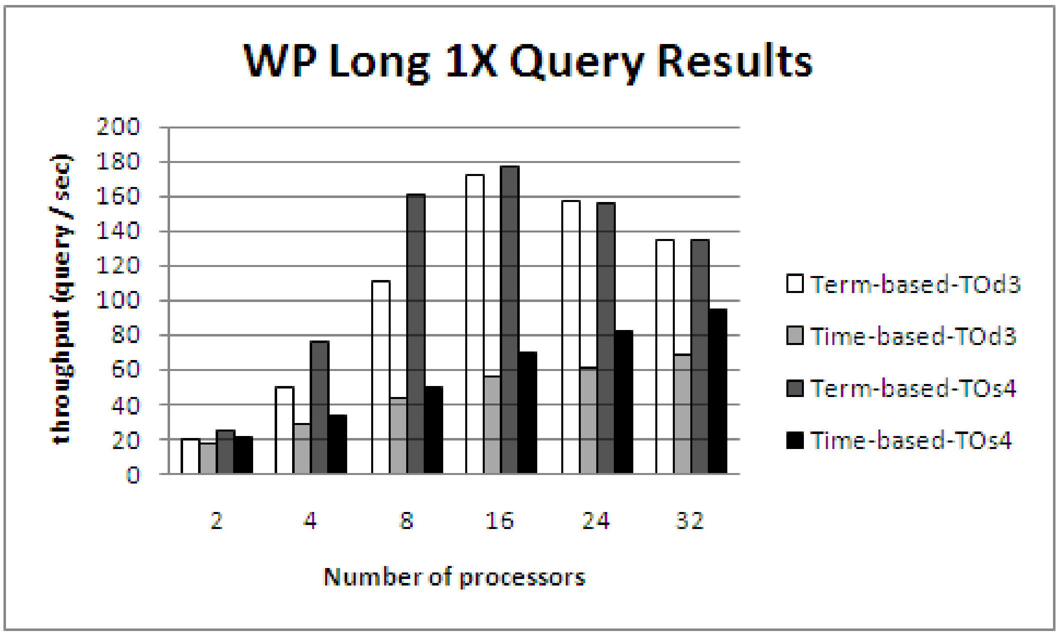

Figure 5.5: Term-based vs. Time-based partitioning on Wikipedia for Long 1X queries.

partitioning scheme, two queries with the same validity time intervals is likely to be sent to more processors on Wikipedia dataset. However, in term-based partitioning case, the number of responsible processors for that query will not differ significantly. Therefore, time-based partitioning is expected to perform worse on Wikipedia dataset due to its short time span.

However, this cannot be the only reason of better term-based partitioning performance in Wikipedia dataset because even if the time interval triples, term-based partitioning still performs better. The size of the inverted index significantly affects query performance on term-based partitioning since the posting length increases with increasing index size. As the posting lists gets longer, the number of postings fetched from the disk even if they do not contribute to the answer set, increase. In other words, with increasing posting list length, the amount of redundant work done in term-based partitioning increase. Therefore, the proposed time-based partitioning scheme scales better than term-based partitioning.

As for the Wiktionary dataset, which is almost three times larger than

Wikipedia dataset, time-based partitioning almost always perform better, even for the short query types. As the number of query terms increase, the number of active index servers in term-based partitioning scheme will increase. However, as long as the query's time interval is kept constant, the number of index servers responsible for answering that query does not change in time-based partitioning. Increasing number of responsible index servers imply increasing communication cost in the network as well as increasing number of disk accesses. Therefore, as queries get longer in terms of number of keywords contained, time-based partitioning results in better throughput values compared to term-based partitioning.

The effect of partitioning on the performance of ranking algorithms is clearly seen from Figures 5.6 to 5.8. Fixing either TOs4 or TOd3 ranking algorithm, compared to term-based partitioning, time based partitioning increases the throughput at all cases seen below. However, regardless of the partitioning scheme, TOd3 performs worse than TOs4, in spite of the fact that in sequential implementation results, TOd3 performs better and has a smaller time complexity.

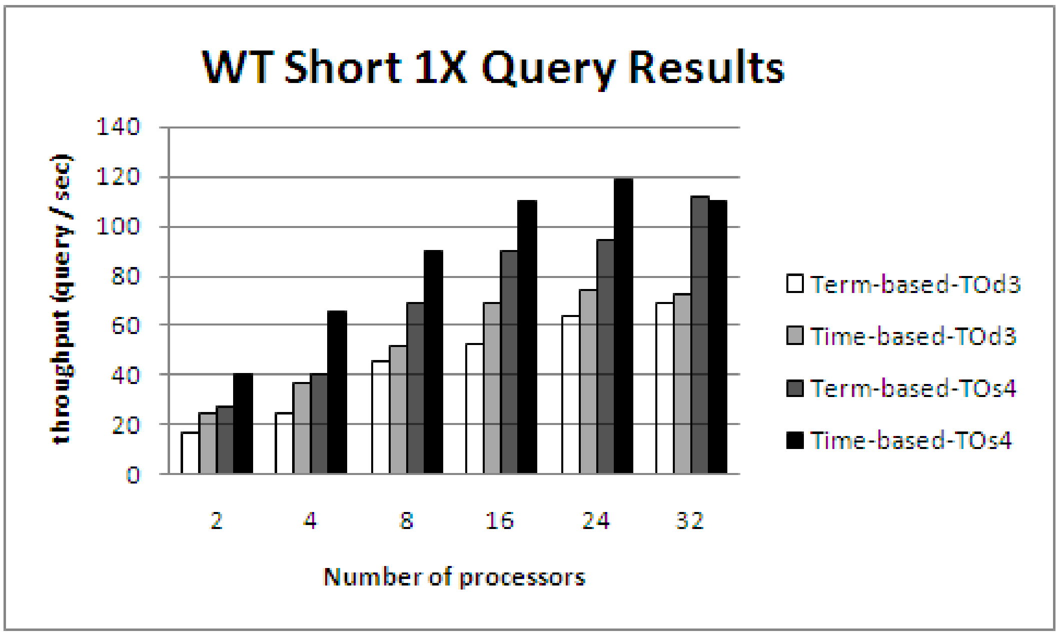

Figure 5.6: Term-based vs. Time-based partitioning on Wiktionary for Short 1X queries.

The decreasing performance of TOd3 in parallel implementation is due to the

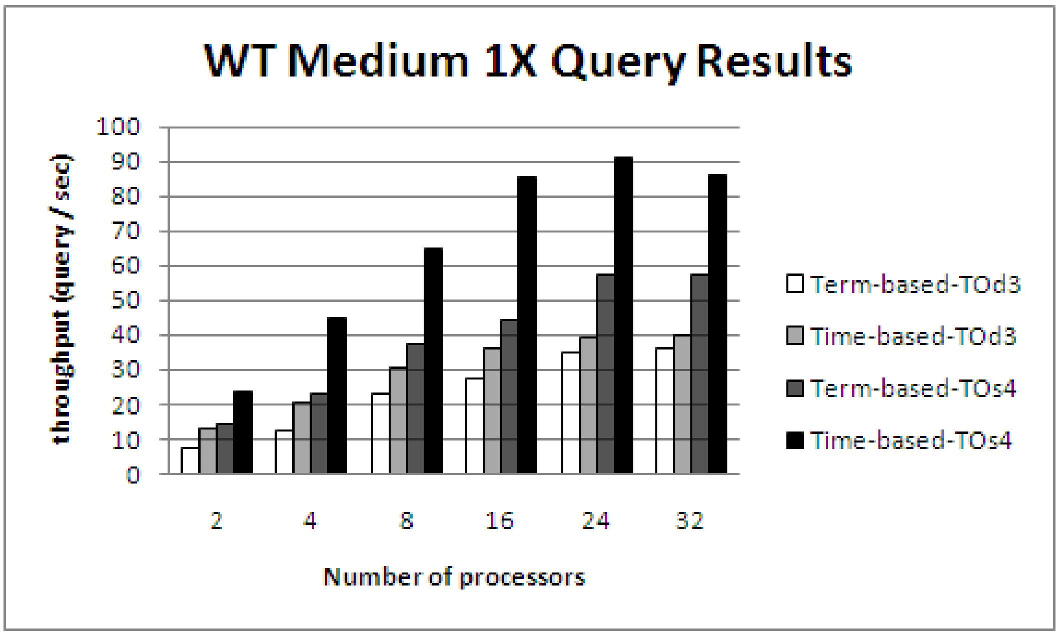

Figure 5.7: Term-based vs. Time-based partitioning on Wiktionary for Medium 1X queries.

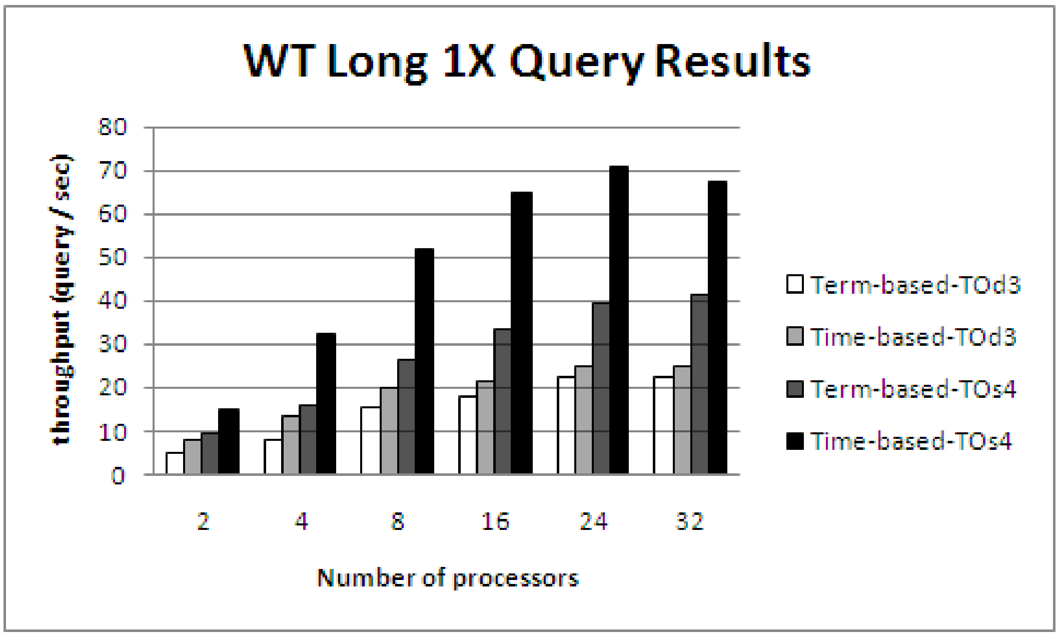

Figure 5.8: Term-based vs. Time-based partitioning on Wiktionary for Long 1X queries.

different query lengths used in sequential and parallel implementations. Cambazoglu and Aykanat [9] assigned 1 to 5, 6 to 25 and 26 to 250 terms for short, medium and long queries, respectively. On the other hand, since we expect significantly fewer number of terms in web queries, 1 to 3, 4 to 6 and 7 to 10 queries are assigned to each query type, in the given order. As the number of query terms increase, the effect of TOd3 on query performance becomes visible due to the data structures used in the implementation of TOd3, which is explained in detail in section 3.3. If there are only a few accumulators in the skip list, the effect of logarithmic search is not reflected to the results. In other words, the overhead introduced due to skip-list data structure cannot be amortized by logarithmic search functionality if there is not enough number of accumulators.

In term-based partitioning, at each processor (or index server) S_i, the number of accumulators inserted into skip list is directly related to the $\sum_{i=1}^{n}(\text{length}(\mathcal{I}_i))$ where n is the number of terms assigned to S_i and $\mathcal{I}_i$ is the posting list of term t_i. If $\sum_{i=1}^{n}(\text{length}(\mathcal{I}_i))$ has a small value, then statistically logarithmic search, insert and update costs does not amortize the overhead cost paid for the skip list data structure. Therefore, since we used shorter queries and generated even shorter subqueries in term-based partitioning or reduced the length of posting lists in time-based partitioning, we significantly reduced $\sum_{i=1}^{n}$. Sending smaller packets for query distribution or decreasing the disk access cost by reducing the posting list size is desirable to increase efficiency. However, as a side-effect, performance of TOd3 ranking algorithm is sacrificed.

Next, let us present experimental results for 3X query type. As seen from the graphs, increasing the validity time interval of temporal web queries has very little effect on the query performance of term-based partitioning scheme. The small decrease in throughput is only due to the increasing number of versions in the answer set as the time span of the query triples. However, it severely effects time-based partitioning results since the number of active processors increases with increasing time interval length. Regarding Wikipedia experiments, term-based partitioning still works better than time-based partitioning but the growing gap between the throughput values becomes more clear particularly as the number of processors increases.

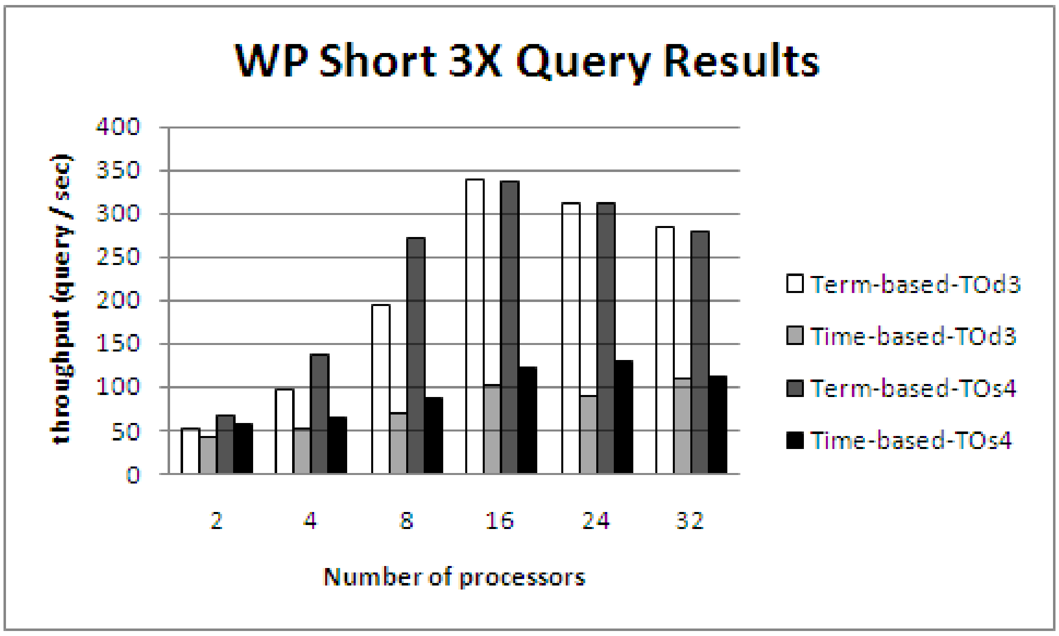

Figure 5.9: Term-based vs. Time-based partitioning on Wikipedia for Short 3X queries.

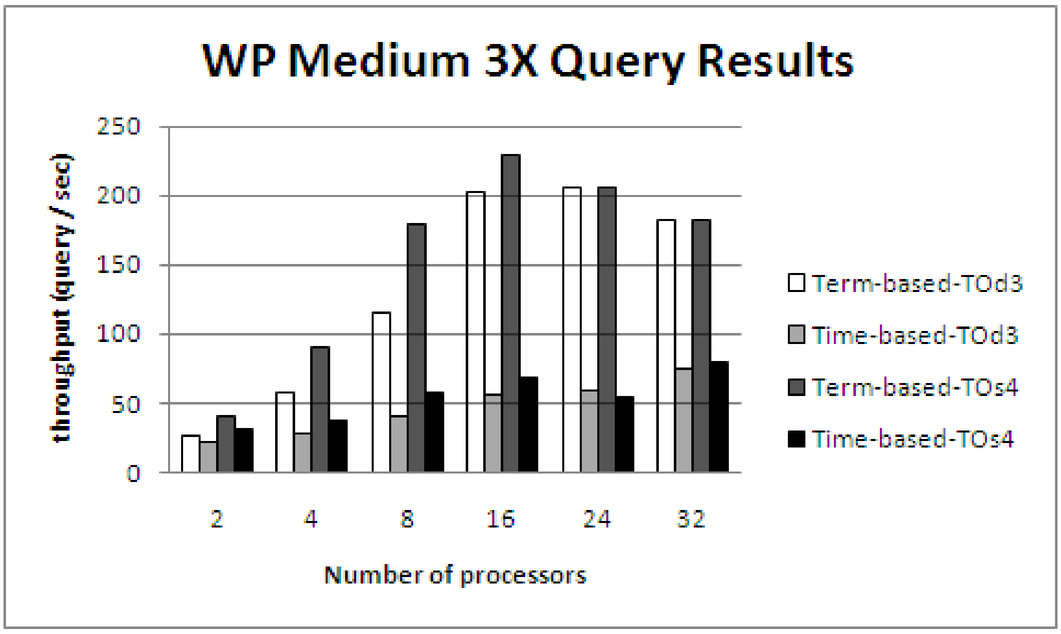

Figure 5.10: Term-based vs. Time-based partitioning on Wikipedia for Medium 3X queries.

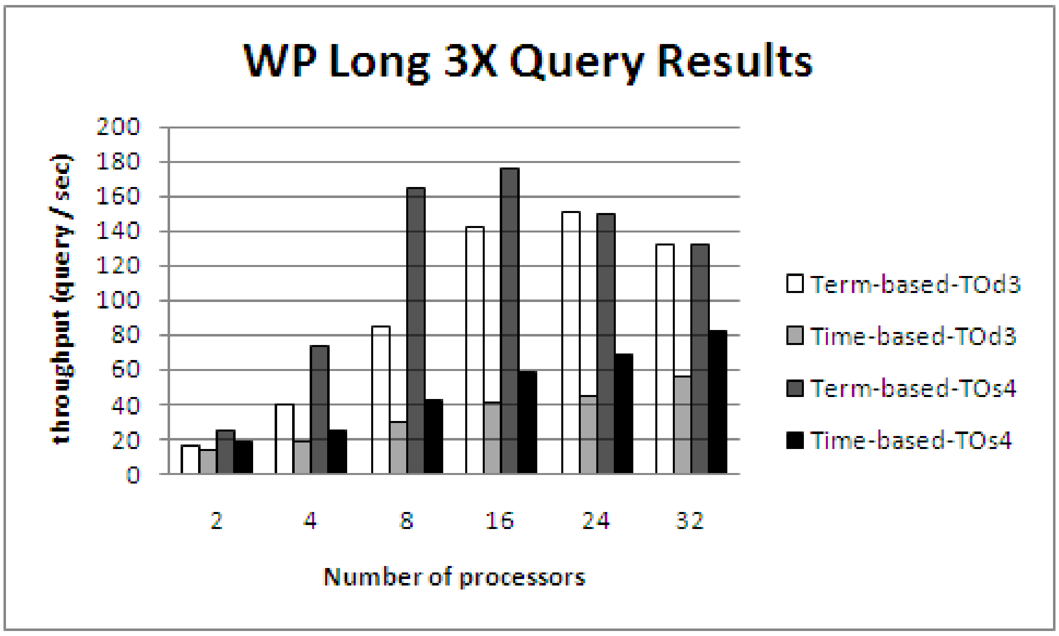

Figure 5.11: Term-based vs. Time-based partitioning on Wikipedia for Long 3X queries.

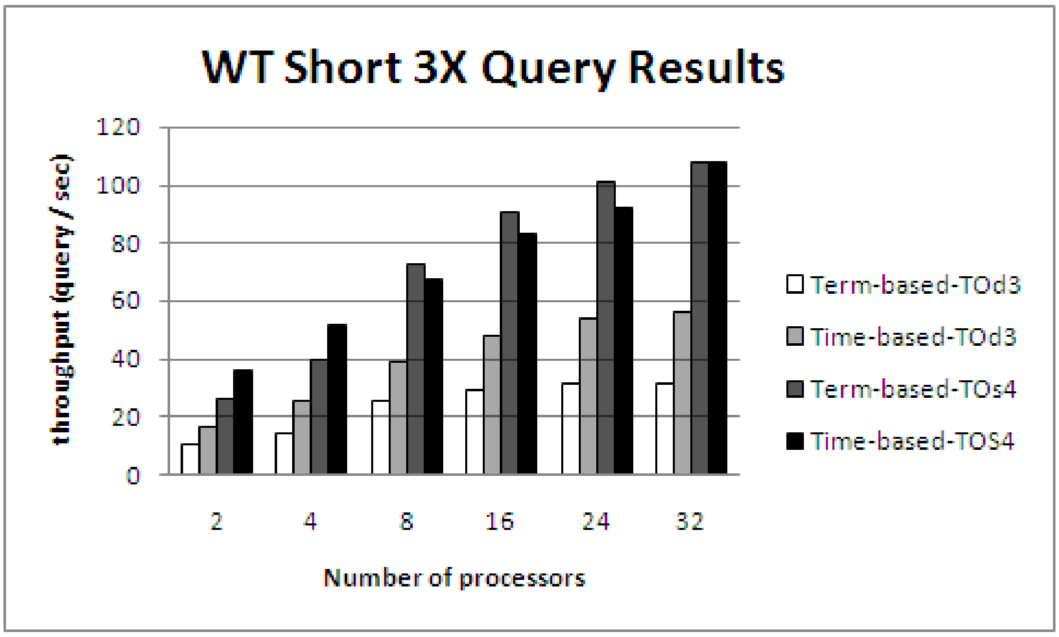

Figure 5.12: Term-based vs. Time-based partitioning on Wiktionary for Short 3X queries.

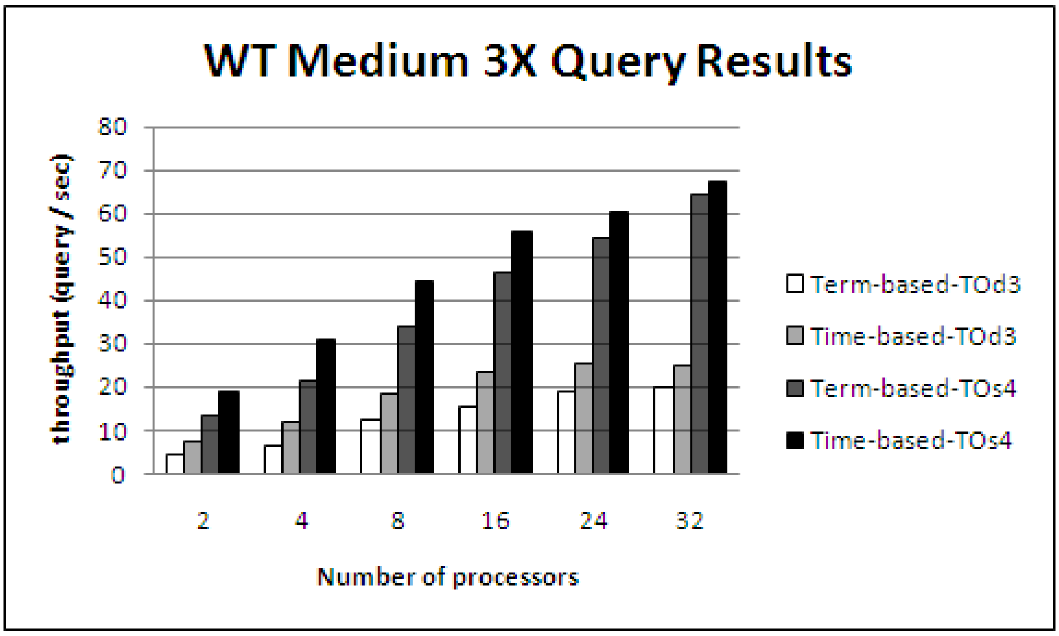

Figure 5.13: Term-based vs. Time-based partitioning on Wiktionary for Medium 3X queries.

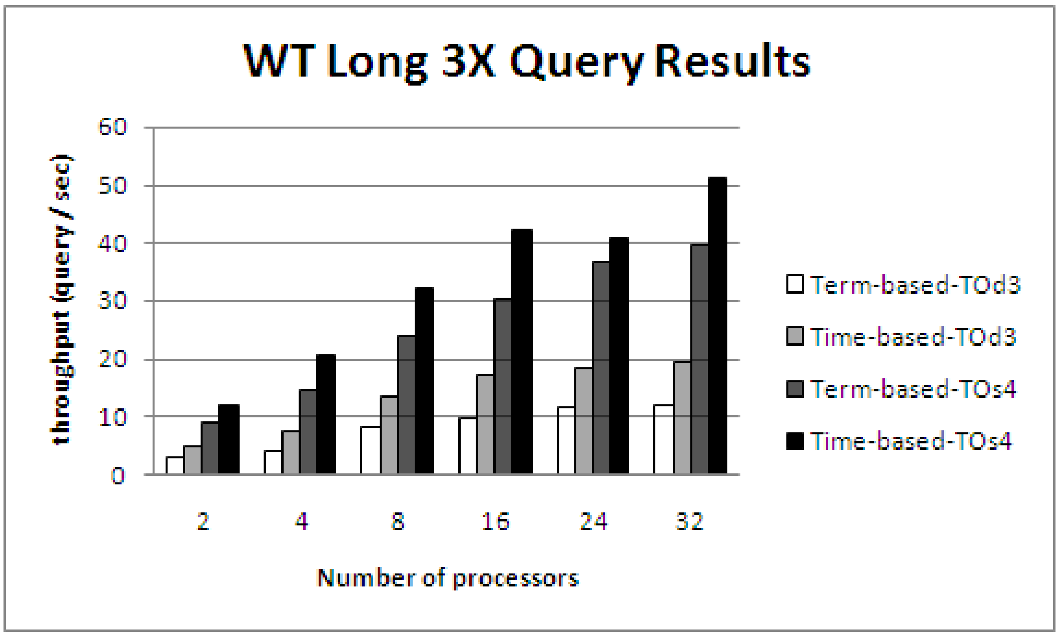

Figure 5.14: Term-based vs. Time-based partitioning on Wiktionary for Long 3X queries.

The 3X experiments on Wiktionary dataset also reveal that time-based throughput rates surpasses term-based throughput rates. However, compared to 1X query results, 3X throughput rates are lower, as expected. In this experimental setup, TOs4 performs better again.

5.4.3 Term-based vs. Time-based comparison considering average response time

In section 5.4.2, we base our comparison on throughput, the number of queries per second. In this section, we analyze the same set of experiments based on average query response time values. Although, an exact equation that relates throughput with average response time cannot be expressed, throughput and average response time could be considered as inversely related in some sense. Throughput is calculated as $throughput = N_q/(T_e - T_s)$ where N_q is the number of queries processed, which is set to 1500 in this experiment, T_s is the time instant when the first query is sent to a processor and T_e is the time instant when the last query is answered by the latest processor. Average response time *art* is calculated as $art = \sum_{i=0}^{N_q} R_i/N_q$ where R_i is the response time of query q_i. Therefore, unless processor idle time is high, average response time is expected to decrease with increasing throughput rates.

Comparing Figures 5.15 to 5.26 with the figures of section 5.4.2, average response time is almost always inversely proportional with throughput except for 2 processor experiments using 1X Short queries on Wikipedia data. Although time-based partitioning has a smaller average response time, its throughput is lower compared to term-based partitioning. This implies that time-based partitioning causes idling time of processors to increase particularly when there are fewer processors.

Actually, this is an expected result because term-based partitioning in a round-robin fashion distributes the terms among processors such that the likeliness of any term appearing in a query is close to one another. Furthermore, by changing the mapping parameter of term-based distribution, terms could be sorted by

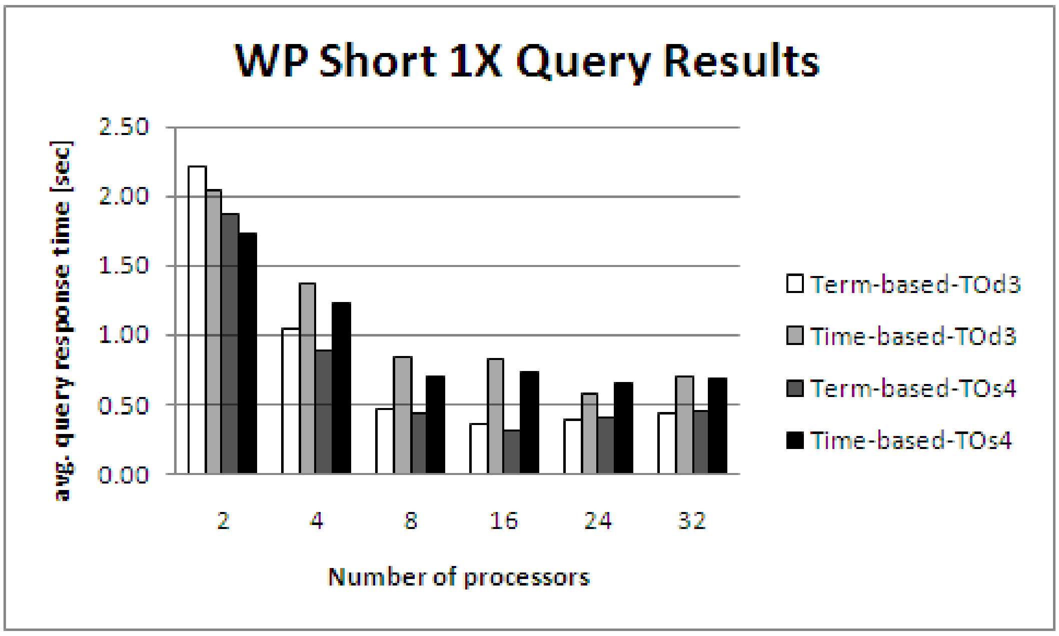

Figure 5.15: Term-based vs. Time-based partitioning on Wikipedia for Short 1X queries.

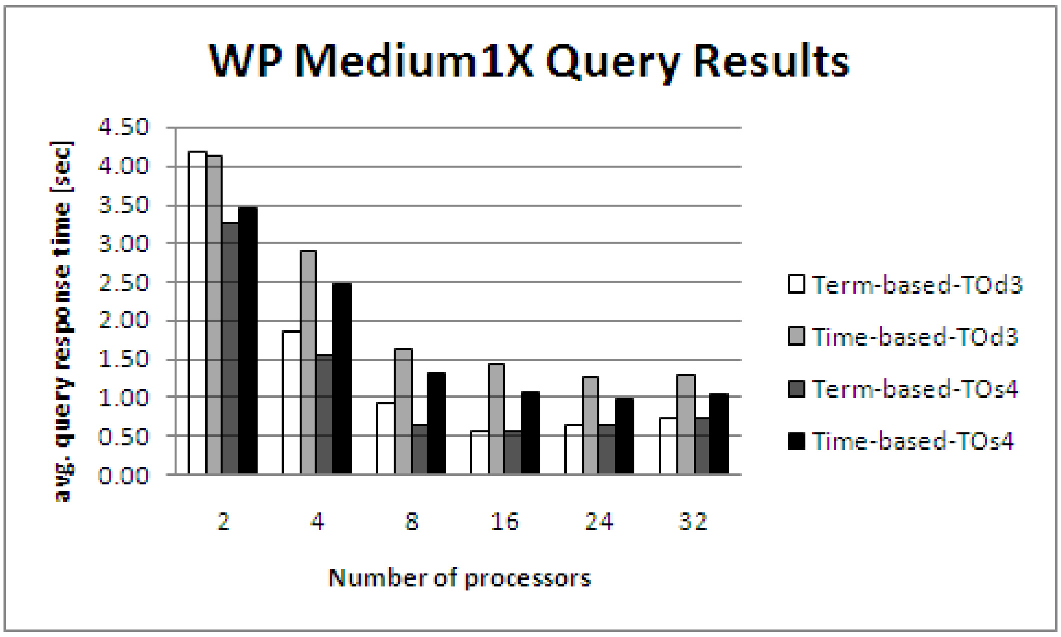

Figure 5.16: Term-based vs. Time-based partitioning on Wikipedia for Medium 1X queries.

their frequencies instead of alphabetical sorting and then round-robin distribution could be carried on. However, in time-based distribution, the queries are statistically more like to come from a recent time period. Since our time-based distribution scheme assigns contiguous time intervals to each processor, round-robin distribution of time intervals is not possible. This challenge leads to an increase in idling time of processors which are responsible from older time intervals.

Considering the average response times on Wiktionary dataset, the same processor idling problem is seen in short queries at 24 and 32 processors. However, in this case, we see the problem at 24 and 32 processors instead of 2 processors. Recall that in the earlier occurrence of decreasing throughput with decreasing average response time, term-based partitioning was giving better throughput. However, in this case, although time-based partitioning gives worse average response time, the throughput is better than term-based partitioning. This means, at high number of processors, time-based partitioning reduces the idling time of processors.

The experiments on 3X queries displayed below show that, as the validity time interval of temporal web queries increase, the performance of time-based partitioning diminishes significantly. As far as term-based partitioning is concerned, increasing the validity time interval of queries has very little effect on its performance. That little effect is actually due to increasing answer set as the query's validity time interval increases.

Apart from comparing experiments carried on different datasets, query types, query lengths and partitioning schemes; the effect of the number of processors should also be analyzed. If the number of processors are increased gradually while keeping the amount of data and tasks constant (in this experiment data is the inverted index and tasks are queries to be processed), after a certain point parallel performance will start decreasing [26]. The reason of decreasing performance is due to the increasing volume of communication and/or increasing amount of total work done by the processors. In these experiments, in the term-based partitioning scheme, as the number of processors increase the volume of communication

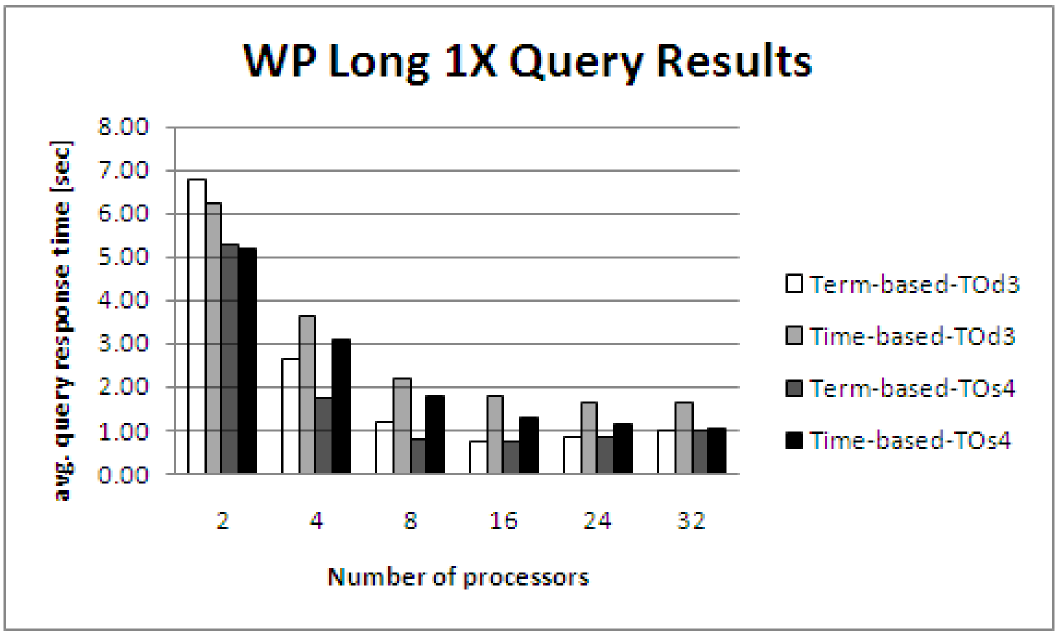

Figure 5.17: Term-based vs. Time-based partitioning on Wikipedia for Long 1X queries.

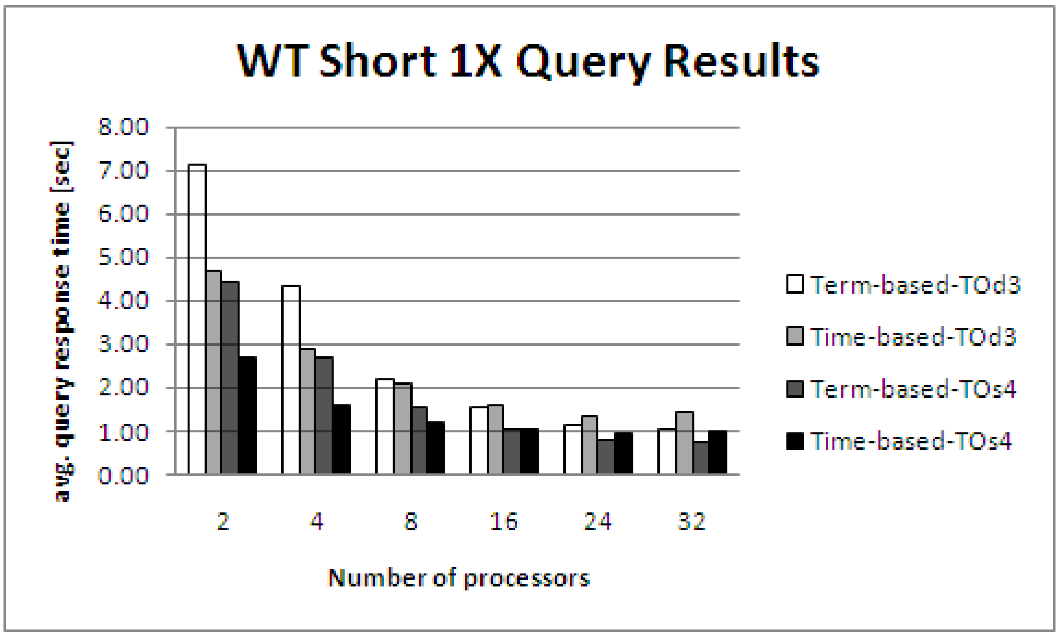

Figure 5.18: Term-based vs. Time-based partitioning on Wiktionary for Short 1X queries.

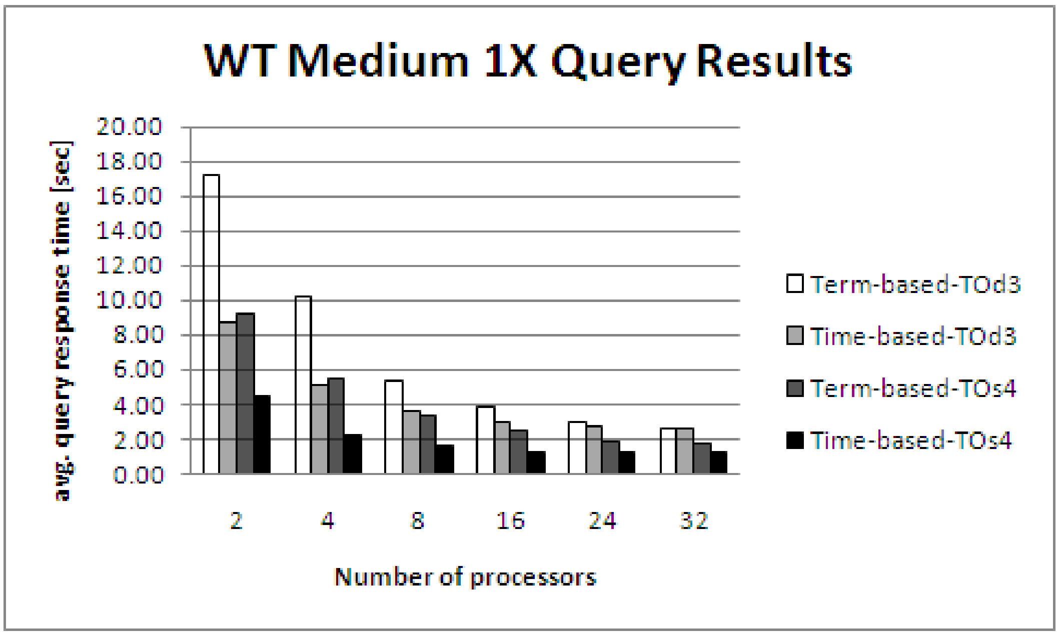

Figure 5.19: Term-based vs. Time-based partitioning on Wiktionary for Medium 1X queries.

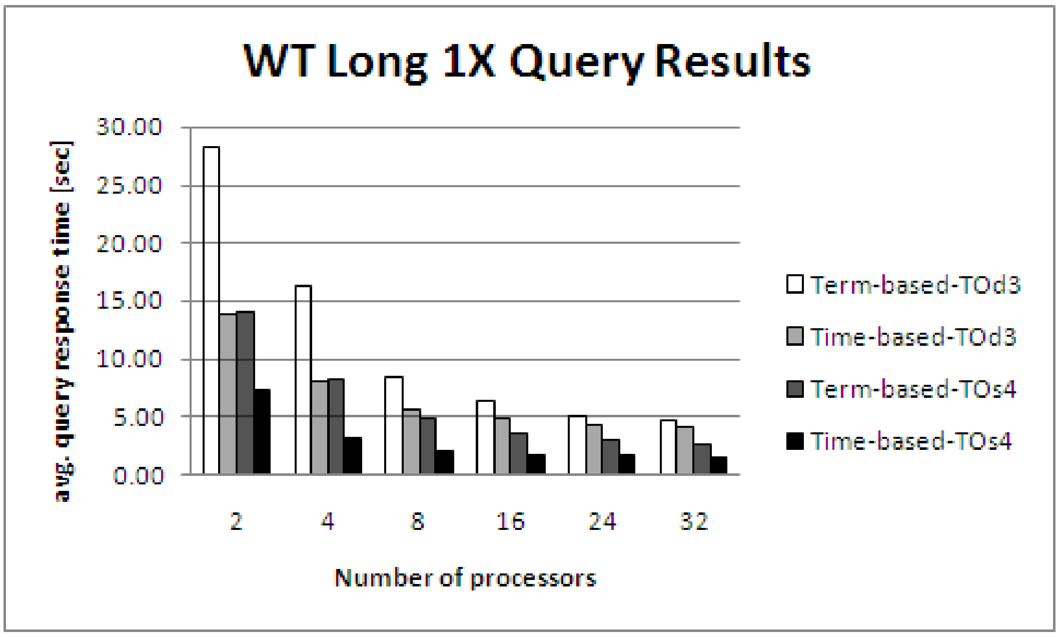

Figure 5.20: Term-based vs. Time-based partitioning on Wiktionary for Long 1X queries.

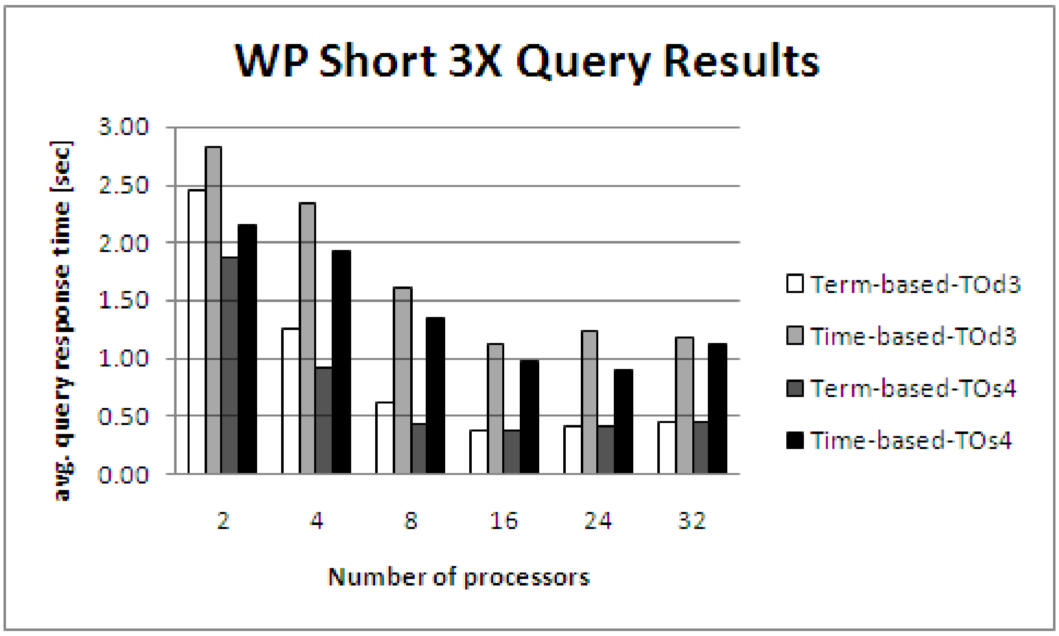

Figure 5.21: Term-based vs. Time-based partitioning on Wikipedia for Short 3X queries.

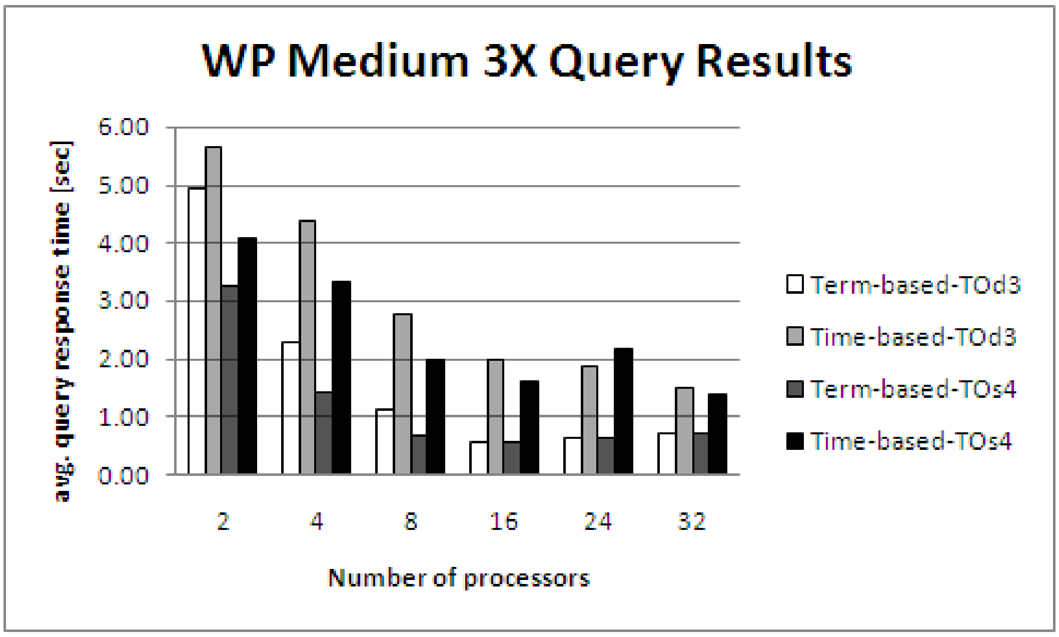

Figure 5.22: Term-based vs. Time-based partitioning on Wikipedia for Medium 3X queries.

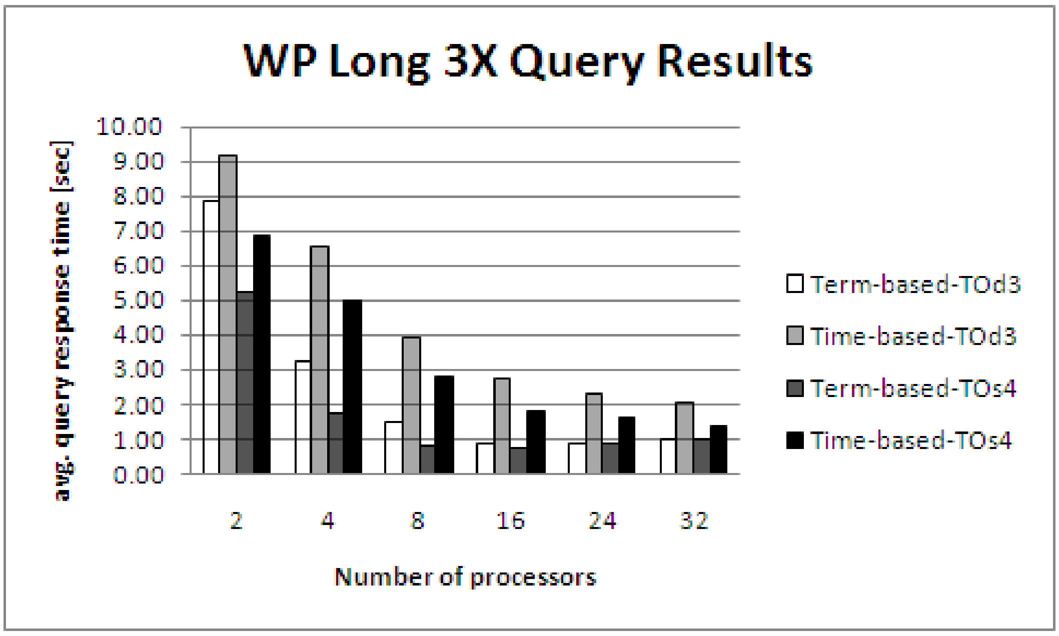

Figure 5.23: Term-based vs. Time-based partitioning on Wikipedia for Long 3X queries.

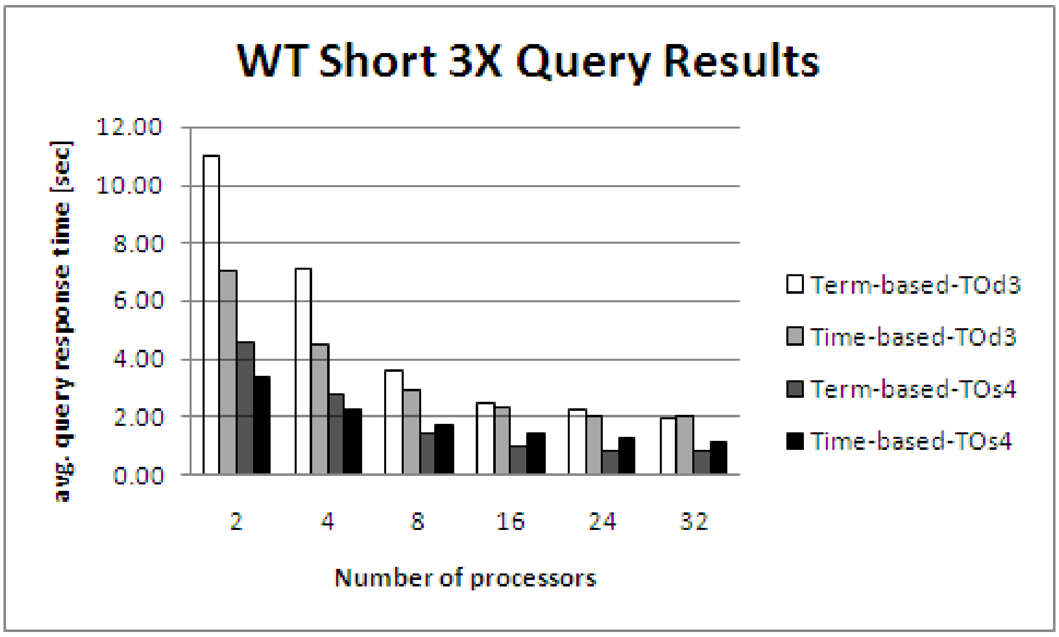

Figure 5.24: Term-based vs. Time-based partitioning on Wiktionary for Short 3X queries.

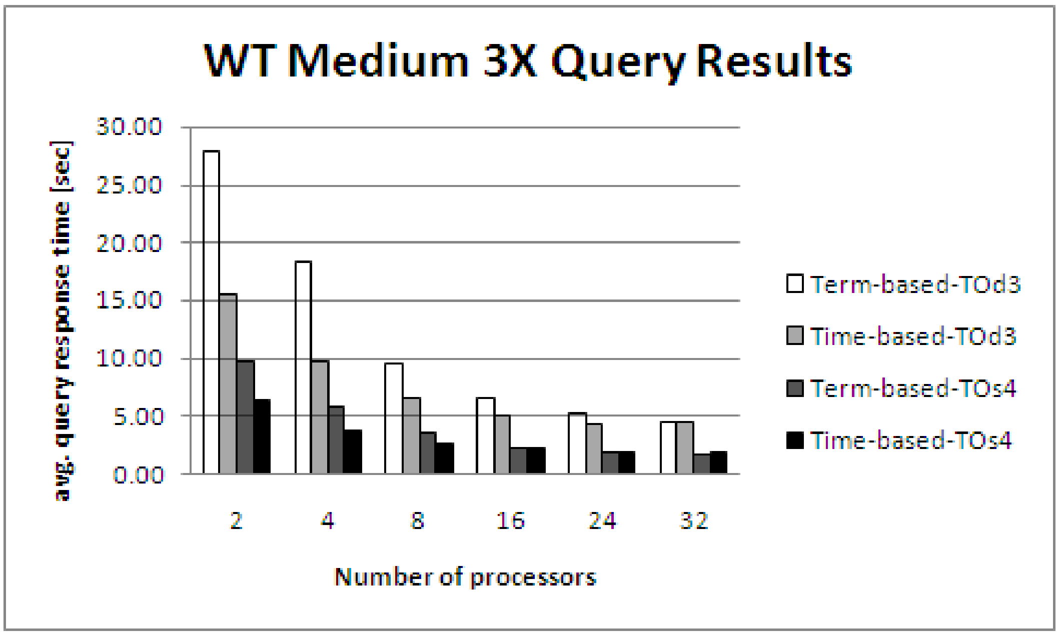

Figure 5.25: Term-based vs. Time-based partitioning on Wiktionary for Medium 3X queries.

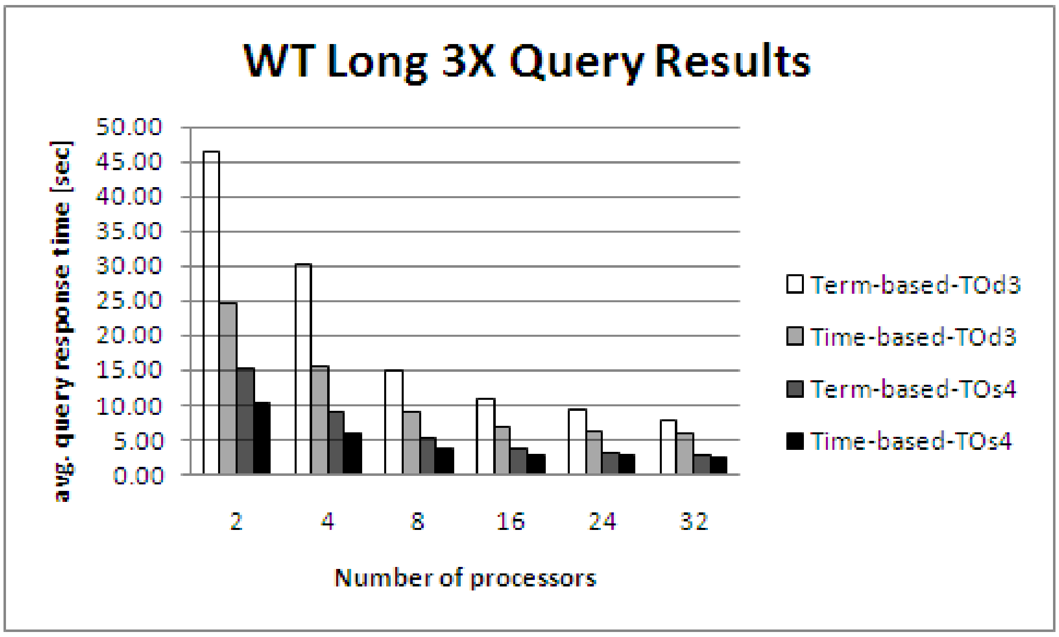

Figure 5.26: Term-based vs. Time-based partitioning on Wiktionary for Long 3X queries.

increase. Furthermore, although the total number of disk accesses remains constant, total number of accumulators to be selected and sorted in each processor increases as well.

On the other hand, in time-based partitioning scheme, as the number of processors increase total volume of communication increases since the number of processors with intersecting time intervals increase. Similar to term-based partitioning total number of accumulators to be selected and sorted in each processor increases and additionally the number of disk accesses in time-based partitioning scheme increases as well. Therefore, as depicted in the figures, after a certain number of processors, throughput starts decreasing.

However, the turning point changes based on the total size of inverted indexes, the total number of query terms and the length of the query time interval. Since the size of Wiktionary dataset is bigger, the turning point on this collection is observed to be later than Wikipedia. The total number of query terms affects the turning point on only term-based partitioning scheme because as the number of terms increase, the number of disk accesses which is an indication of the total number of tasks increases. Due to the same reasoning, the turning point of time-based partitioning is affected as well. As the query time interval increases, the turning point of time-based partitioning is pulled back significantly due to the increasing number of processors which will be responsible for answering that query. However, the time interval has almost no effect on term-based partitioning scheme. To sum up, Wiktionary corpus, time-based partitioning scheme, short and 3X queries are more likely to have earlier turning points.

Chapter 6

Conclusion and Future Work

There have been several studies on temporal text retrieval. However, as pointed in temporal text-retrieval researches, sequential systems are far from being sufficient on today's vast amount of temporal data. This makes employment of parallel and distributed systems a necessity on temporal text retrieval. In this work, we tried to contribute to temporal text retrieval's scientific progress in a small way. To the best of our knowledge, this is the first study that exploits the problem of answering time-interval web queries on temporal document collections on a parallel architecture. In particular, we proposed a parallel text retrieval system which is able to answer time-interval web queries on temporally versioned document collections.

In order to implement a parallel text retrieval system, we should change the classical structure of the inverted index data structure so that the time-related information can also be represented within the index. As a second contribution, we propose two time-based partitioning schemes: time-balanced and size-balanced partitioning. The former method partitions the dataset equally according to timeslice, while the latter aims to balance the storage cost at each index server. According to our experiments on two real life temporal document collections, we observe that size-balanced partitioning leads better results since the queries have a tendency to be related with more recent time intervals.

We also compare the performances of size-balanced time-based partitioning scheme with traditional term-based round-robin partitioning scheme. The experimental results revealed that for long time-interval queries having fewer terms, term-based partitioning yields better results. On the other hand, if the queries contain a short time-interval and many terms, time-based partitioning performs better. As the size of the document collection increase, the performance of time-based partitioning exceeds term-based partitioning. Furthermore, time-based partitioning calculates exact scores while in term-based partitioning, there might be small variations in the order of top s versions. We also investigated parallel performances of two term ordered ranking methods TOs4 and TOd3 and found out that although TOd3 performs better in sequential implementations, TOs4 utilizes parallelism better than TOd3.

Our future plans include, storing versions as a set of fragments to reduce total inverted index size, implementing document ordered ranking algorithms on a parallel architecture and distributing versions among processors by their start/end times. We believe that index size reduction will implicitly improve query performance by considerably reducing the amount of data to be searched in. Furthermore, document ordered ranking algorithms are expected to perform better in time-based partitioning algorithms due to their resemblance with document-based partitioning.

Bibliography

[1] Anick, P. G., and Flynn, R. A. (1992). “Versioning a Full-Text Information Retrieval System,” in *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR ’92)*, pp. 98-111.

[2] Berberich, K., Bedathur, S., Neumann, T., and Weikum, G. (2007). “A Time Machine for Text Search,” in *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR ’07)*, pp. 519-526.

[3] Berberich, K., Bedathur, S., Neumann, T., and Weikum, G. (2007). “FluxCapacitor: Efficient Time-travel Text Search,” in *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB ’07)*, pp. 1414-1417.

[4] Berberich, K., Bedathur, S., Neumann, T., and Weikum, G. (2007), “A Time Machine for Text Search,” *Technical Report*, No. MPI-I-2007-5-002, Max-Planck Institute for Informatics.

[5] Brewington, B. E., and Cybenko, G. (2000). “How Dynamic Is the Web?, *Computer Networks*, 33:257-276.

[6] Broder, A. Z., Iron, N., Fontoura, M., Herscovici, M., Lempel, R., McPherson, J., Qi, R., and Shekita, J. (2006). “Indexing Shared Content in Information Retrieval Systems,” in *Lecture Notes in Computer Science, Vol. 3896, (Proceedings of Advances in Database Technology (EDBT))*, pp. 313-330.

[7] Burns, G., Daoud, R., and Vaigl, J. (1994). "LAM: an Open Cluster Environment for MPI," in *Proceedings of the Supercomputing Symposium*, pp. 379-386.

[8] Cambazoglu, B. B. (2006). "Models and Algorithms for Parallel Text Retrieval," *Ph.D. Thesis*, Department of Computer Engineering, Bilkent University.

[9] Cambazoglu, B. B., and Aykanat, C. (2006). "Performance of Query Processing Implementations in Ranking-based Text Retrieval Systems Using Inverted Indices," *Information Processing and Management: an International Journal*, 42:875-898.

[10] Cambazoglu, B. B., and Aykanat, C. (2006). "Effect of Inverted Partitioning Schemes on Performance of Query Processing in Parallel Text Retrieval Systems," in *Lecture Notes in Computer Science, Vol. 4263*, pp. 717-725.

[11] Catal, A. (2003). "Parallel Text Retrieval on PC Clusters," *M.S. Thesis*, Department of Computer Engineering, Bilkent University.

[12] Chien, S.-Y., Tsotras, V. J., Zaniolo, C., and Zhang, D. (2002). "Efficient Complex Query Support for Multiversion XML Documents," in *Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology (EDBT)*, pp. 161-178.

[13] Chien, S.-Y., Tsotras, V. J., and Zaniolo, C. (2001). "Efficient Management of Multiversion Documents by Object Referencing," in *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*, pp. 291-300.

[14] Chien, S.-Y., Tsotras, V. J., and Zaniolo, C. (2000). "Version Management of XML Documents," in *Lecture Notes In Computer Science, Vol. 1997, (Selected papers from the 3rd International Workshop WebDB 2000 on The World Wide Web and Databases)*,
pp. 184-200.

[15] Cho, J., and Garcia-Molina, H. (Sep. 2000). "The Evolution of the Web and Implications for an Incremental Crawler," in *Proceedings of the 26th International Conference on Very Large Databases (VLDB '00)*, pp. 200-209.

[16] Dalling, T. (1998). "Versioning of Web Resources," *B.S. Thesis*. Department of Computer Science, James Cook University of North Queensland.

[17] Elmasri, R., Wuu, G. T. J., and Kim, Y.-J. (1990). "The Time Index: An Access Structure for Temporal Data," in *Proceedings of the 16th International Conference on Very Large Databases (VLDB '00)*, pp. 296-303.

[18] English Wikipedia. http://en.wikipedia.org/.

[19] English Wiktionary. http://en.wiktionary.org/.

[20] Jatowt, A., Kawai, Y. and Tanaka, K. (2008). "Visualizing Historical Content of Web Pages," in *Proceeding of the 17th International Conference on World Wide Web*, pp. 1221-1222.

[21] Gunadhi, H., and Segev, A. (1993). "Efficient Indexing Methods for Temporal Relations," *IEEE Transactions on Knowledge and Data Engineering*, 5:496-509.

[22] Hersovici, M., Lempel, R., and Yogev, S. (2007). "Efficient Indexing of Versioned Document Sequences," *Lecture Notes in Computer Science, Vol. 4425*, pp.76-87.

[23] Huang, L., Zhu, J.J.H. and Li, X. (2008). "HisTrace: Building a Search Engine of Historical Events," in *Proceeding of the 17th International Conference on World Wide Web*, pp. 1155-1156.

[24] Internet Archive. http://www.archive.org/.

[25] Jatowt, A., Kawai, Y., Nakamura, S., Kidawara, Y. and Tanaka, K. (2006). "Journey to the Past: Proposal for a Past Web Browser," in *Proceedings of the 17th Conference on Hypertext and Hypermedia,* pp. 134-144.

[26] Grama, A., Gupta, A., Karypis, G. and Kumar, V. (2003). *Introduction to Parallel Computing*, 2^{nd} ed. Addison-Wesley.

[27] Kouramajian, V., Kamel, I., Elmasri, R. and Waheed, S. (1994). "The Time Index+: an Incremental Access Structure for Temporal Databases," in *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM),* pp. 1-12.

[28] Lawrence, S., and Giles, C. L. (1999). "Accessibility of information on the Web," *Intelligence, Vol. 11*, pp. 32-39.

[29] Leung, T. Y., and Muntz, R. R. (1992). "Temporal Query Processing and Optimization in Multiprocessor Database Machines," in *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB '92),* pp. 383-394.

[30] Francis, W. N., and Kucera, H. (Jan., 1983). "Frequency Analysis of English Usage: Lexicon and Grammar," *PHoughton Mifflin* . ISBN 0-395-32250-2.

[31] Nørvåg, K., and Nybø, A. O. (2004). "Creating Synthetic Temporal Document Collections", *Technical Report* IDI 6/2004, Norwegian University of Science and Technology, Available at http://www.idi.ntnu.no/grupper/DB-grp/.

[32] Nørvåg, K., and Nybø, A. O. (2004). "DyST: Dynamic and Scalable Temporal Text Indexing", *Technical Report* IDI 10/2004, Norwegian University of Science and Technology, Available at http://www.idi.ntnu.no/grupper/DB-grp/.

[33] Nørvåg, K., and Nybø, A. O. (2004). "Improving Space-efficiency in Temporal Text-indexing", *Technical Report* IDI 7/2004, Norwegian University of Science and Technology, Available at http://www.idi.ntnu.no/grupper/DB-grp/.

[34] Nørvåg, K., and Nybø, A. O. (2003). "Space-efficient Support for Temporal Text-indexing in a Document Archive Context", in *Proceedings of the 7th European Conference on Digital Libraries (ECDL)*, pp. 511-522.

[35] Nørvåg, K., and Nybø, A. O. (2004). "Supporting Temporal Text-containment Queries in Temporal Document Databases", *Journal of Data and Knowledge Engineering*, 49:105-125.

[36] Pearsoned Education. http://wps.pearsoned.co.uk/.

[37] Ramaswamy, S. (1997). "Efficient Indexing for Constraint and Temporal Databases", in *Proceedings of the 6th International Conference on Database Theory (ICDT 97),* pp. 419-431.

[38] Salton, G., and McGill, M. J. (1983). *Introduction to Modern Information Retrieval*, New York: McGraw-Hill.

[39] Salzberg, B., and Tsotras V. J. (1999). "Comparison of Access Methods for Time-Evolving Data", *ACM Computing Surveys,* 31(2): 158-221.

[40] Silva, M. J. (2003). "The Case for a Portuguese Web Search Engine", in *Proceedings of ICWI-2003, the IADIS International Conference WWW/Internet,* pp. 411-418.

[41] Stack, M. (2005). "Full Text Search of Web Archive Collections", in *in Proceedings of the 5th International Web Archiving Workshop (IWAW).*

[42] Tokuc, A. A. (2008). "Performance Comparison of Query Evaluation Techniques in Parallel Text Retrieval Systems," *M.S. Thesis*, Department of Computer Engineering, Bilkent University.

[43] Tsotras, V. J., and Kangelaris, N. (1995). "The Snapshot Index: An I/O-Optimal Access Method for Timeslice Queries", *Information Systems Vol. 20, Issue 3,* pp. 237-260.

[44] Witten, I. H., Moffat, A. and Bell, T. C. 1999. Managing Gigabytes: Compressing and Indexing Documents and Images (2nd ed.). San Francisco, CA: Morgan Kaufmann.

[45] Zhang, J., and Suel, T. (2007). "Efficient Search in Large Textual Collections with Redundancy", in *Proceeding of the 16th International Conference on World Wide Web,* pp. 411-420.