Inverted Index Partitioning Strategies for a Distributed Search Engine

by

Hiren Patel

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2010

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

One of the greatest challenges in information retrieval is to develop an intelligent system for user and machine interaction that supports users in their quest for relevant information. The dramatic increase in the amount of Web content gives rise to the need for a large-scale distributed information retrieval system, targeted to support millions of users and terabytes of data. To retrieve information from such a large amount of data in an efficient manner, the index is split among the servers in a distributed information retrieval system. Thus, partitioning the index among these collaborating nodes plays an important role in enhancing the performance of a distributed search engine. The two widely known inverted index partitioning schemes for a distributed information retrieval system are document partitioning and term partitioning.

In this thesis, we introduce the *Document over Term* inverted index distribution scheme, which splits a set of nodes into several groups (sub-clusters) and then performs document partitioning between the groups and term partitioning within the group. As this approach is based on the term and document index partitioning approaches, we also refer it as a *Hybrid Inverted Index*. This approach retains the disk access benefits of term partitioning and the benefits of sharing computational load, scalability, maintainability, and availability of the document partitioning. We also introduce the *Document over Document* index partitioning scheme, based on the document partitioning approach. In this approach, a set of nodes is split into groups and documents in the collection are partitioned between groups and also within each group. This strategy retains all the benefits of the document partitioning approach, but reduces the computational load more effectively and uses resources more efficiently.

We compare distributed index approaches experimentally and show that in terms of efficiency and scalability, document partition based approaches perform significantly better than the others. The Document over Term partitioning offers efficient utilization of search-servers and lowers disk access, but suffers from the problem of load imbalance. The Document over Document partitioning emerged to be the preferred method during high workload.

Acknowledgements

I would like to express my sincere gratitude and appreciation to my supervisor, Dr. Ihab Illyas, for his encouragement, guidance and support all throughout my graduate studies as well as during the preparation of this thesis. I would like to thank my committee members: Dr. Frank Wm. Tompa and Dr. Daudjee Khuzaima for their constructive comments and for serving on my committee.

I have been fortunate to have had as my senior colleague and mentor Andrew Kane. His advice and support through the period of my candidacy has made him nothing less than a second supervisor to me. I am thankful to Gunes Aluc and Anup Challamalla for their suggestions. I am deeply thankful to Jalaj Upadhya for his comments on my thesis. I am thankful to Rakesh Patel for providing support and encouragement during my stay at Waterloo.

I am deeply grateful to my family for their love and support. Without them, this work could not have been completed.

I am highly indebted to my friends, Pradeep, Somit, Garurav, Tejas, and all the other colleagues in the Database Research Group, whose help, stimulating suggestions and encouragement helped me at all times in this research. I would like to thank all the little people who made this possible.

Dedication

This is dedicated to my family.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

"Information retrieval (IR) is a field concerned with the structure, analysis, organization, storage, searching, and retrieval of information" [40]. Further, IR is the science of searching for documents, for information within documents and for metadata about documents, as well as that of searching relational databases and the World Wide Web (WWW). IR systems involve a range of tasks and application. The usual search scenario (task) involves user typing in a query and receiving answers in the form of a list of documents. Searching information is a crucial part of application in corporations, government and many other domains like vertical search, enterprise search, desktop search, and peer-to-peer [36]. However, searching the WWW is by far the most common application involving IR systems. The users rely on a IR application like Web search engine to navigate through ever evolving ocean of Web data. In this thesis, we focus on a Web search engine application of IR.

In the context of the Web, data continues to grow at an ever increasing rate. Millions of Web pages are created every year. In 1992, there were just 1000 pages on the Web. In 1999, the estimated indexable Web size increased to 800 million pages [24]. As of June 2000, over two billion Web pages were posted on the internet. In August 2005, Yahoo! disclosed the number of indexable Web pages as 19.2 billion documents [28]. As amount of data is in the order of petabytes and growing exponentially, sophisticated techniques are required to implement an efficient IR system. A typical search engine is hosted by a centralized machine and the index is replicated across several machines. Such large centralized systems are not capable of handling increasing number of users and volume of the data. This gives rise to the need for a parallel and distributed search engine. The main challenge here is to design a large scale distributed search engine that satisfies the user's expectations and at the same time uses resources efficiently, thereby reducing the cost per query. This is possible only by efficient usage of the network, caching and high concurrency.

In this chapter, we first present the architecture of a typical search engine in Section 1.1. In Section 1.2, we discuss about architecture of a traditional monolithic search engine along

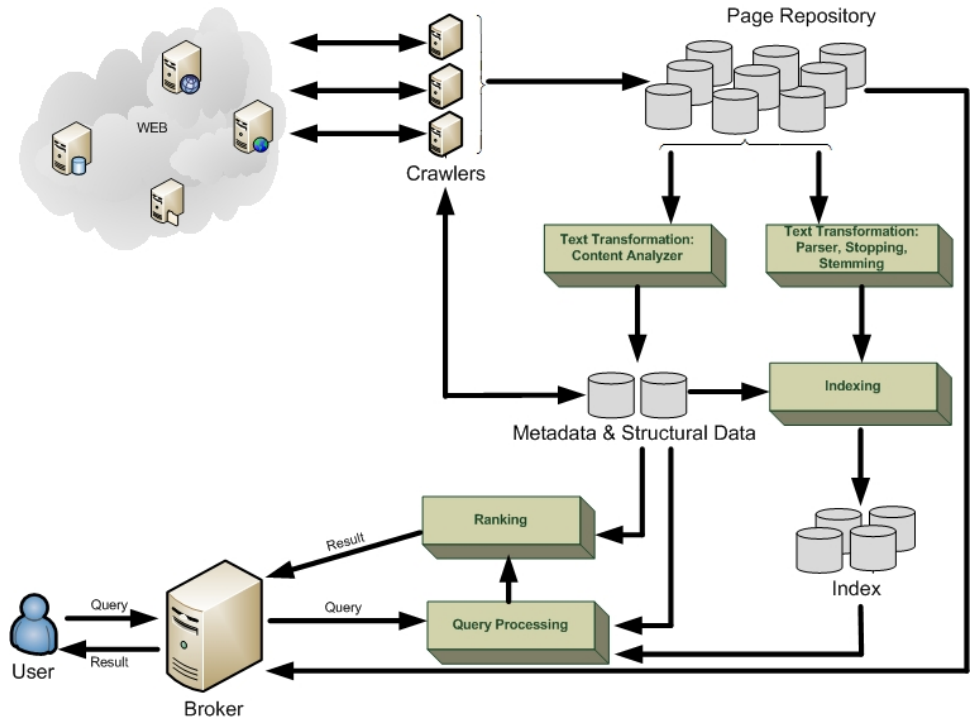

Figure 1.1: Architecture of Search Engine.

with its limitations followed by distributed search engine. In Section 1.3, we discuss various index partitioning approaches for a distributed search engine and their shortcomings. In Section 1.4, we give the formal description of the problem. Finally we state the main contributions of this thesis.

1.1 Architecture of Search Engine

In this section, we present a high level abstraction of various components of a search engine. A search engine consists of three core components: crawling, indexing, and query processing. A brief description of each component is given below. Explaining all components in detail is beyond the scope of this thesis but more information can be found in [4, 16, 40].

A *crawler* is a program that identifies and acquires documents from the Web by following the hyper-links. It is also known as spider, walker and wanderer. Crawlers begin with an initial set of URL's (Uniform Resource Locator) as input to scan and download their content and store it for further processing in a page repository (data store). Then crawlers proceed iteratively following all the outgoing links extracted from the already

downloaded Web pages in order to acquire more content. This process is repeated until the crawler runs out of *seeds* (URL's), bandwidth, or storage-space. The crawler downloads Web pages of various formats like HTML, XML, PDF, and Microsoft-word, etc. from various sources like database servers, file servers, Web servers, or database-driven content management systems. Most search engines require these documents to be converted into some consistent format, mainly plain text [40].

Before advancing to the indexing phase; text-transformation and content analysis is carried out on each page stored in the page repository as shown in Figure 1.1. Text-transformation mainly deals with parsing, stopping, stemming, and content analysis. The *parser* is responsible for processing the text by first tokenizing and then recognizing structural elements of the page such as titles, figures, links, and headings. As the query is compared against same set of documents, both the query and the documents must be tokenised using the same parser. The parser must also ignore *stopwords* while tokenizing[1]. A stopword is a common word that helps to form a sentence but contributes little meaning on their own to the text. Thus, it is believed that removing stopwords usually has no impact on the search engine's effectiveness [36]. Some of the examples of stopwords are "of", "to", and "the". Further, removing stopwords reduces the size of the index. *Stemming* is another text-level transformation, that groups similar words derived from a common stem and replaces them with one designated word. For instance, "stemmer", "stemming", "stemmed" are based on "stem" and should be replace by a designated word "stem".

The content analysis component deals with the extraction of *metadata* from a document. Metadata is the information about the document and is not a part of the text. Examples of metadata include: document type, length, and author. The content analysis component also extracts links and anchors text from Web pages. Document metadata, links, and anchor text are stored as structured data into a datastore. *Query processing* and *ranking* components of the search engine make extensive use of such structured information. Links identified by the content analysis component are again fed to the crawler(s).

The output of the text transformation component becomes input to the indexing component, which creates index data structures that enable fast searching. Considering the size of the Web, index creation must be efficient, both in the terms of the time taken to create the index and time to search against the index. Further, it should be possible to efficiently update the index when new documents are acquired or when existing documents are updated.

All modern search engines are using *inverted index* data structures. Other index structures have been used in the past are signature files, and spatial data structures such as k-d tree. Explaining all the different types of index structures used by a search engine is beyond the scope of this thesis but more information can be found in [4, 16]. An inverted

[1]The process of removing stopwords from the text is known as *stopping* in IR.

index is similar to a book index, it consists of two types of data structures, namely *lexicon* and *postinglists*. Lexicon is a collection of unique words or tokens extracted from the text. Each lexicon, which is also known as index term or token, is assigned a list representing documents containing that particular index term. A postinglist contains the information about the location of the occurrence of a token in the collection. An index stores more information than just a list of documents containing that particular index term such as the frequency of the term in the document, and the section of the document (title, abstract, content) where that term appears. Such information are extensively used by the ranking component to compute the weight of a particular term in the document. Inverted index data structure is considered the most efficient and the most flexible index structure in IR engines. [4, 16, 40, 43].

A query received from the user is transferred to a query processing component, which acts as an interface between the user and the search engine. One of the tasks for this component is to accept user queries and transform each query into index terms. This includes refining a vague or misspelled query. Further, searching is becoming increasingly complex. A query may include phrases, questions, whole passages, or documents. Thus, another task of the query processing component is to deliver consistently superior results by understanding the exact intent of the user query. Query processors must also know what information is available, how it relates to the query, and where it is located. The ranking component takes the output of the query processing (which is list of documents), scores the list of documents, and send it to the broker which in turn sends the results to the users. The task of generating presentable results to user is carried out by the broker.

1.2 Monolithic vs. Distributed Search Engine

The search engine architecture presented in the previous section can be implemented in a centralized or a distributed fashion. The traditional "monolithic search engine" employs centralized single large index architecture as shown in Figure 1.2. The crawler(s) are used to download the desired content from various sources and store them in the repository. The downloaded content is then converted into a single large index by the indexing component of the search engine. Once indexed, the search application on the search-server facilitates search against the single large index and returns consolidated results to the user. An IR server[2] may have more than one core information retrieval process running; for instance, the searcher process (responsible for search operation) and the indexer process (responsible for creating index on data) can run on the same server.

A major benefit of a monolithic search engine is that it is easy to manage a single large index and have simple query processing algorithms. On the other hand, as the volume

[2]IR server is also referred as search-server, node, or simply server in this thesis.

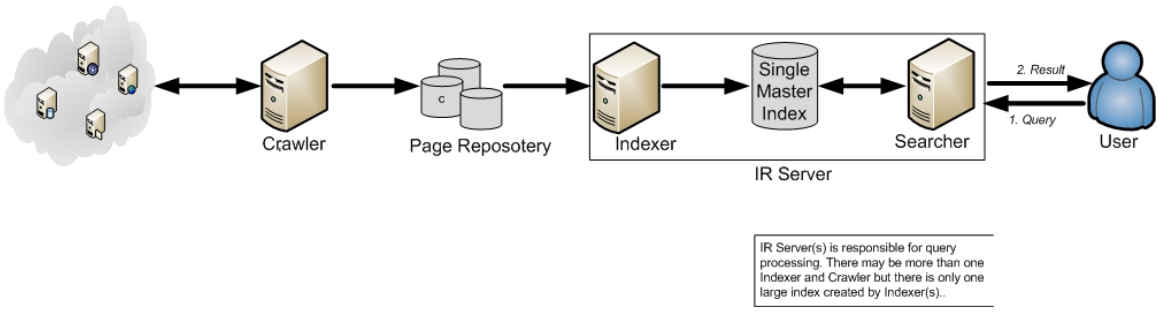

Figure 1.2: Logical organization of Monolithic Search Engine.

of the electronic data and number of query requests increases, it is nontrivial to perform basic IR tasks (such as indexing and searching) on a single machine efficiently. Searching and indexing costs grow with the size of data collection. Moreover, the monolithic search engines neither have the computational power nor storage capabilities to deal with the large data collection and the increasing number of users. Scalable distributed IR system aim to anomer most of the above issues.

Parallel and distributed IR systems typically consist of a set of core IR server processes, like an indexer and a searcher along with a crawler. Such systems are usually deployed on a large cluster of nodes, each of which is responsible for searching index. Figure 1.3 shows the logical organization of the distributed search engine. Along with deploying the IR application on a large cluster, we usually need replication to improve the performance of a parallel IR system. Consider that we have n search-servers in the cluster. By creating n replicas of the index and assigning each replica to a different node, we can increase throughput of the service by a factor of n because multiple queries can be processed in parallel. Such parallelism can improve the service rate (throughput) but does not improve query latency.

In this thesis, we focus on another popular approach to optimize the performance. In this approach, we split the index into n parts and have each node work only on its assigned part of the index. This is referred as *intra-query parallelism* in the literature [36]. In such distributed IR systems, a designated process known as the *broker* or *receptionist* is responsible for accepting user query request. It then forwards each user query to all or some search-servers. All participating servers will process the query and send partially discovered results to the broker. After accumulating all the results received from the participating search-servers, the broker sends the final top-k results[3] to the user. As each search-server is responsible for only a small part of the index and as queries are evaluated in parallel, such system improves throughput as well as query latency.

[3]List of k most query relevant documents.

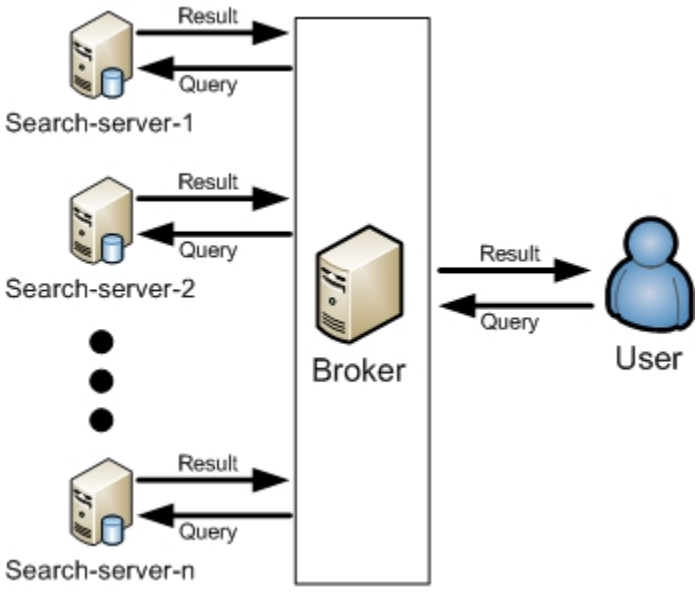

Figure 1.3: Logical organization of Distributed Search Engine.

In a distributed IR system, performance is highly influenced by the way in which the index is partitioned across the search-servers [10]. Two standard techniques for index organization in a distributed environment are document partitioning (DP) and term partitioning (TP) [16, 40, 36]. In *document partitioning*, the data collection is partitioned among the search-servers and each search-server hosts an inverted index for its assigned subset of the documents. For instance, consider a data collection with documents D=$\{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8\}$ partitioned among the search-servers such that search-server1 handles documents $\{d_1, d_2, d_3\}$, search-server2 handles $\{d_4, d_5, d_6\}$, and search-server3 handles $\{d_7, d_8\}$. An inverted index created by search-server3 is as follows:

$$
\begin{aligned}
< t_{1,3}, L_{1,3} > where\ L_{1,3} = \{d_8\},\\
< t_{4,3}, L_{4,3} > where\ L_{4,3} = \{d_8\},\\
< t_{5,3}, L_{5,3} > where\ L_{5,3} = \{d_7\},\\
< t_{7,3}, L_{7,3} > where\ L_{7,3} = \{d_7, d_8\}
\end{aligned}
$$

where t_{ij} is the term t_i located on search-server-j and L_{ij} is the list of documents with term t_i located on search-server$j$.

One of the main advantages of DP is that it is simple and easy to manage as a result of two factors. Firstly, insertion of a new document is trivial as each document is stored on the single search-server. Secondly, the query processing on each search-server is done

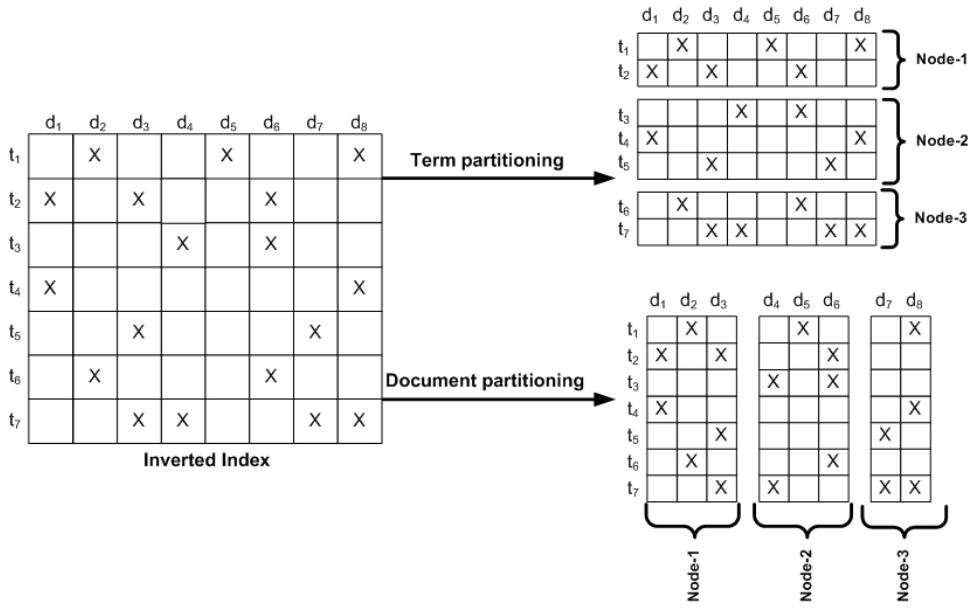

Figure 1.4: Term and document inverted index partitioning.

independently of others. In addition, DP supports availability as query evaluation is uninterrupted even if one of the search-servers becomes unavailable, of course with a loss of some effectiveness[4]. However, this approach incurs a disk access overhead. For instance, for n search-servers and q query terms, $n * q$ inverted lists must be fetched[5]. Another drawback of DP is that the search-servers execute several unnecessary operations when querying a sub-collection, which may contain only few or no relevant documents. Only top-k documents relevant to the query from $n * k$ documents retrieved by n search-servers are transferred to the user by the broker, hence some of the computation done by the search-servers is futile and results in an increase in the intercommunication and overhead on the broker.

Alternative to DP is TP. A TP approach creates an index on the entire collection and then *range partitioned* the inverted list by terms among the search-servers. Following the previous example, consider the same data collection with the set of documents, D. The inverted index stored on the search-server3 is as follows:

$$< t_6, L_6 > where L_1 = \{d_2, d_6\}$$
$$< t_7, L_7 > where L_2 = \{d_3, d_4, d_7, d_8\}$$

where t_i is the term in the vocabulary and L_i is list of documents containing the term t_i.

[4]Effectiveness measures the ability of the search engine to find the right information. The two most common effectiveness metrics are recall and precision.

[5]Additional disk seek are in parallel and each server only do q disk-seek.

A major drawback of the TP approach is an uneven distribution of the load across the search-servers, i.e., it is vulnerable to hotspots. This affects the throughput. Also, this approach does not scale well because inserting a new document requires the index to be updated on all the search-servers. Another problem is that as each search-server is responsible for a certain set of terms, T_n from the vocabulary, unavailability of a search-server will halt the query processing for the terms stored on the unavailable search-server. Further, for query processing, postinglists are often required to be transferred over the network to the broker resulting in an increase in the intercommunication cost. There are, however, some advantages of this approach in the query processing phase as query is forwarded only to the search-servers which contains postinglist(s) for the query term(s). Thus, it significantly reduces disk access and volume of data exchanged (between the disk and main-memory) resulting in an efficient utilization of the resources.

1.3 Problem Statement

The volume of the Web and the number of queries submitted to a Web search engine by the users make inverted indexes partitioning a critical issue for the performance. In an ideal scenario, what we want to achieve from a search engine is that (i) it should be scalable with respect to the size of the data collection and resources, (ii) it must support efficient resource utilization by avoiding hotspots and distributing load evenly among the available resources, and (iii) it should maintain availability at all times, without compromising on the performance (effectiveness and efficiency[6]). The two index partitioning strategies we have named so far (TP and DP) cannot address all the above mention scenario collectively. Therefore we propose a hybrid and optimized index partitioning approach that makes decision based on all these four attributes: high performance, scalability, availability, and resource utilization.

1.4 Main Contributions of the Thesis

The main contribution of this thesis can be summarized as follows:

- We present the document over document partitioning strategy for a distributed search engine which is based on the DP approach.
- We present a new hybrid approach based on the term and document partitioning which support scalability, availability, and efficient utilization of the resources. We

[6] Efficiency measures how quickly things get done. The two most common efficiency metrics are throughput and query latency.

show experimentally that the postinglist pruning technique for the hybrid index partitioning approach can improve the performance.

- We provide insight on how to measure throughput in a steady state. We also compare the time taken by different approaches to reach the steady state by varying the number of simultaneous query requests.
- We discuss the effect of concurrency on different inverted index organization approaches. One of our goals is to devise an inverted index partitioning strategy which supports higher concurrency with minimum effect on the throughput and latency.

1.5 Organization of the Thesis

The organization of the rest of the thesis is as follows. Chapter 2 describes related work and describes various strategies used to tackle IR system problems stated in Section 1.2. In Chapter 3, we present the approach developed to tackle the problem. We analyze various methods for efficiency in Chapter 4 and conclude with directions on future work in Chapter 5.

Chapter 2

Related Work: Distributed Inverted Indexing Techniques

As outlined in Section 1.2, the inverted index of a large document collection can be distributed across the cooperating search-servers in two different organizations: *document partitioning* and *term partitioning*. In this chapter, we describe in detail document partitioning in Section 2.1 and term partitioning in Section 2.2 followed by a comparative study presented in the literature in Section 2.3. We discuss various optimization performed on both approaches in Section 2.4. We describe various hybrid approaches presented in the literature in Section 2.5.

2.1 Document Partitioning (DP)

The idea behind the DP approach is to partition the data collection into several smaller sub collections, one per server, and build an inverted index on that sub-collection. In the literature, DP is also known as *horizontal partitioned index* or *local index*. The DP approach is the most commonly used approach by a distributed Web search engine [4, 7, 8, 13]. An incoming user query is received by the frontend server; also known as the broker, which schedules the query on the search-servers based on the scheduling strategy. In DP, the broker may choose among two possible strategies for scheduling a query. The more common approach is to broadcast the query to all the underlying search-servers [18]. As all the search-servers are involved in the execution of the query, this method has the advantage of enabling an even load balance among all the search-servers [29]. On the other hand, it has a major drawback of processing the query submitted by the user on all the search-servers, resulting in many random disk access.

Another query scheduling schema is to forward the query only to the search-server(s) that contains a relevant document(s). Thus, we select only a subset of the machines in the cluster. The problem of selecting such a subset is challenging and is known as "collection selection" and is described in more detail in Section 2.4.1. Moreover, the performance of the collection selection strategy depends on how documents are partitioned among the search-servers. Various data collection partitioning strategies are described in more detail in Section 2.4.1.

Despite its disadvantages due to inefficient utilization of the resources and disk access problem, DP has several advantages which make it a favorable choice. Some of these advantages are listed below:

- **Locality:** In DP, all the search-servers operate independently of each other because each search-server has self contained search index. Thus, the main advantage of this approach is its simplicity as it can be easily deployed on a loosely coupled environment. This locality is extremely convenient for query processing because complex queries can be conveniently solved, as the comparisons between the postinglists of the query terms to find relevant set of documents is local. Moreover, the index is easy to maintain since insertion of a document is done locally.

- **Dynamic Index:** So far we have assumed that indexing is a batch process. However, in practice, most of the documents are constantly changing and updated. Also the collection tends to get bigger over the time as every day there is more news and more emails. Thus, we require that the index partitioning techniques must be able to respond to any dynamic collection efficiently without affecting the performance of the system. We can solve this problem with two techniques, *index merging* and *result merging*. In index merging, we make a new smaller index I_2 and merge it with the old index, I_1 to make a new index I. This is a reasonable update strategy when index updates comes in a large batch. For a single or a small batch update, it is not a good strategy. For these small updates, it is better to maintain small separate index, I_2 for a new data and delay merging until the second index becomes sufficiently large. User queries are evaluated against both indexes, I_1 and I_2, and later, the results are merged to find top-k results. This is called *result merging*. The DP can adapt to index merging and to result merging without any complication because each node operates independently.

- **Scalability:** In the case of distributed IR systems, scalability must address two issues, (1) adding new node(s) to an existing system, and (2) adding new data to an already existing node with minimum effect on the performance while maintaining the availability of the system. As the collection becomes larger, the sub-collections of each node and index will also become larger. Larger index increases computation

cost on a search-server. In such case, adding an extra node(s) seems to be more feasible alternative. As mentioned earlier, in DP it is easy to add new node(s) without effecting the availability and with reasonable affect on the performance, as there is no internodal dependency. Further, addition of a new document requires index update only on the node where document is added or changed.

- **Availability:** As the DP approach can be deployed on a loosely coupled environment, query evaluation can proceed even if one of the search-servers is unavailable. However, this might result into loss of effectiveness considering possibility of unavailable server containing most relevant query result(s).

- **Load Balancing:** Majority of the proposed strategies in the literature adopts straight forward approach where documents are randomly partitioned among the search-servers. We have seen that with random distribution, average load is smooth in the case of DP [29], but does not guarantee an even load. There is not much work in the area of load balancing in DP. However, DP has been shown by many researchers to be the best choice among parallelization schema [36].

2.2 Term Partitioning (TP)

The idea behind the TP is to create an index on the entire collection and then to range partition the lexicon and the corresponding array of the postinglists among the nodes. In the literature, TP is also known as *vertical partitioned index* or *global index organization.*

Although DP is the most commonly used index partitioning technique, it can reveal its potential only if the index is stored on low latency devices, such as main memory or flash memory. The TP deals with the disk access problem by dividing the index by terms instead of by documents. In TP, each search-server n_j is responsible for a certain set of terms $T_{i,j}$. A node participates in a query processing only if one or more terms from that query belongs to the set $T_{i,j}$ of the node n_j. Thus, the numbers of disk access in the case of term partitioned index are fewer as compared to the document partitioned index.

Although TP has some advantages over DP, it has several limitations which constrains it from being used in practice. Some of these limitations are listed below:

- **Complexity** The execution of a multi-word query is complex as it requires both transfer of long postinglists between the nodes and merging them. Further, in TP, adding or updating documents might require the index to be update which might be spread across different servers making it a complex operation.

- **Dynamic Index:** Usually, indexes are rebuilt from scratch after the update of the documents. But it might not be the case for special document collection like news articles or blogs, where updates are very frequent. This can affect the performance of the system since update operation requires locking of index using a mutex. This problem worsens in the case of TP, as postinglists of terms that are required to be updated might be spread across many nodes. Instead of creating the entire index again, Ribeiro-Neto *et al.,* [35] presented better approach to generate term partitioned index in which each server needs to separately index a disjoint part of the collection, and then negotiate and execute a set of pairwise exchange of the postinglists. Either way, is more complex and requires extra processing as compared to the document partitioned index.

- **Scalability:** As stated in Section 2.2, for scalability, system must be able to address two issues: adding a new node to the existing system and adding a new document to an already existing node with minimum effect on the performance and availability of the system. In the case of TP, it is easy to add new nodes and copy index to the newly added node without affecting the availability and performance. However, this dynamic structure for adding new documents or updating the existing documents constraints the capacity and response time of the system as we previously explained in the case of dynamic index.

- **Availability:** In the case of TP, unavailability of search-server(s) will halt the query processing for queries containing term present on unavailable server.

- **Load Imbalance:** The TP suffers from load imbalance among the search-servers. Load balancing of a partition is not governed by a prior analysis of relative term frequency, but rather by the distribution of the query terms and their co-occurrence, which can drift with time. For instance, if a term has a long postinglist and appears more frequently in the search query, then the corresponding index node may experience much higher load than any other nodes in the distributed search engine. This can drastically affect the performance of the system. Moffat *et al.* in [29] tries to address this problem of load balancing on the term partition index. They showed that it is possible to balance the load on the term partitioned index by exploiting the information on the frequency of the terms occurring in the query logs.

2.3 Comparison between Term Partitioning and Document Partitioning

Barla *et al.* [10] investigated the performance of the TP and DP with respect to the response time and throughput on MPI-based parallel query processing implementation.

They measured the difference in the performance of both the approaches by varying the number of terms in the query and cluster size. Their result indicates that the throughput is better in TP than DP in the case of batch queries. They also concluded that TP achieves better throughput than the DP as the number of nodes in the cluster increases.

Jeong and Omiecinski [21] investigated the performance of the two inverted index partitioning strategies on a shared-everything multiprocessor machine with multiple disk. They investigated the performance in a simulated environment with different workloads generated by varying the terms frequency in the document and query and also by varying the number of disk and multiprogramming level. The result of their simulation shows that the term based partitioning is better in terms of throughput when the term distribution is less skewed, whereas DP performs better when the terms are highly skewed.

Yates *et al.* in [2] investigated the two partitioning techniques on a share-nothing parallel system. Their result demonstrates that TP performs better than DP in terms of throughput in the presence of a fast communication link and a more powerful broker. Nevertheless, as stated in [3], *although DP and TP have been widely studied, it is still unclear on the circumstances under which each one is suitable.*

2.4 Optimizing Inverted Index Partitioning Strategies

In practice, neither the TP nor DP approach is used without additional optimization. In this section we first present some of the work done in the literature to optimize the performance of the DP and TP techniques.

2.4.1 Optimizing Document Partitioning Approach

For an optimized document partitioned IR system, the major goal is to partition the index such that the number of contacted servers is minimal, resources are utilized efficiently, and there is an even distribution of load among the search-servers by avoiding hotspots. To achieve these goals, it is necessary to minimize the number of servers contacted for query processing by sending queries only to the servers which contain the relevant documents. Such problem in literature is known as the collection selection problem. Minimizing the number of servers contacted for the query processing depends on how documents are partitioned among the servers.

Partitioning Data Collection

Deciding how to divide the collection to create good document clusters is complex. The majority of the proposed approaches in the literature adopt a simple approach, where documents are randomly partitioned among the servers, and each query is evaluated by all the servers. However, distributing documents randomly across the servers does not guarantee an even load distribution [1]. Further, random distribution of the document results in an inefficient utilization of the resources. This is because a server may execute several operations unnecessarily when querying the sub-collection, which may contain only few or no relevant documents.

Different approach is adopted by the peer-to-peer (P2P) IR systems where users independently collect documents of their interest. The document collections generated in such a manner are overlapping, redundant, incomplete, and limited to peer's interest. Further, as partitioning is done on an interest basis, smart collection selection approach is required to route the query to the appropriate peer, which contains query relevant documents. Many peer-to-peer IR systems are presented in the literature [31, 38, 42]. The P2P-IR systems cannot guarantee high performance because most of the time all peers participate in the query processing as the document collection is partitioned inefficiently.

The query log is a vital source of information which can help to create a good document clusters. Puppin *et al.* [33] presented a naive strategy to cluster documents according to the information coming from a query log. The approach is explained briefly in the next section.

Bhagwat *et al.* in [5] presented an approach based on a feature based clustering of the documents. When a document is ingested into the repository, a small number of partitions are chosen to store the features of the document. The authors presented new data structures called *feature indices* for all the documents in the repository. The index key is the feature itself. Each feature points to the list of files that it occurs in. The decision as to which partitions the document should be routed to (for storing at ingestion time, and for similarity based search at query time) is solely based on the features of the document or query.

Another approach is to do content based or topic base document clustering. As stated by Puppin *et al.* in [33],

> pSearch [39] performs an initial LSI(Latent Semantic Indexing) transformation that is able to identify concepts out of the document base. LSI works by performing a Singular Value Decomposition (SVD) of the document-term matrix, which projects the term vector onto a lower-dimensional semantic space. Then, the projected vectors are mapped onto a Content Addressable Network(CAN) [34]. Search and insertion is done using the CAN space. In other

words, when a query is submitted, it is projected, using the SVD, onto the CAN, and then neighbors are responsible for answering with the documents they hold.

In [30], Patel proposed a document clustering strategy based on the document classification technique namely Latent Semantic Indexing (LSI) [14] and Latent Dirichlet Allocation (LDA) [6] using a query log which is explained later in the section. Although, LSI might be computationally more efficient than most of the topic mixture models, but it is limited to synonymy problem. It fails to address polysemy(same word with different meaning) which is efficiently dealt by enhancement of LSI known as *probabilistic latent semantic indexing*(pLSI) [19]. The most criticized shortcoming of pLSI, it is not a proper generative model for new documents. This leads to generative model namely LDA, which immediately attracted a considerable interest from the statistical machine learning and natural language processing communities. The basic generative process of LDA closely resembles pLSI.

Collection Selection

Once the documents are partitioned (clustered), we need to devise an efficient collection selection strategy that minimizes the number of servers contacted for a query processing. Many approaches are proposed in the literature which deals with the collection selection problem [33, 9, 17, 30, 5, 23, 26]. In [9], the authors presented a collection selection approach based on a Bayesian probabilistic inference network called *Collection Retrieval Inference Network* (CORI). In CORI, retrieval effectiveness is compared between the partitioned collections. The CORI approach assumes that the best collections are the one that contains most documents relevant to the query. The ranking of the collection is done based on an inference network where the leaves represent document collection and intermediate nodes represent the terms that occurs in the data collection. Their experiment shows that there was no impact on the effectiveness (recall and precision) when compared with a centralized IR system.

As stated earlier, Puppin *et al.* in [33] presented a strategy to partition a document collection and to perform collection selection based on a query vector [32] which is derived from the query log. They performed co-clustering on query vectors to group together similar documents. Their results show that this technique outperforms CORI [9]. However, their technique is biased because when a new document or query arrives they use *TF.IDF* metric to decide which clusters are the best match (each dictionary file is considered as a document, which is indexed with the usual TF.IDF technique). Their technique uses simple text (lexical) matching approach, which is inherently inaccurate. This is because there are many ways for a user to express a given concept using different words and also because most of the words have multiple meanings.

In [30], Patel proposed a strategy based on LSI for document clustering and collection selection using query log. In this strategy, authors introduce the term-query-cluster matrix, which is formulated from a query log where a row represents a term and a column represents a query-cluster (collection of similar queries derived from co-clustering algorithm). The query-cluster were scored using vector model based on the normalized frequency. Since in the models like "the term count model" and "classic vector space model," the terms with high occurrences are assigned more weight than the term occurring few times in the documents and are vulnerable to keyword spamming. In such process, ranking and retrieval is compromised. These term vector models can be made less susceptible to keyword spamming by normalizing frequencies [14, 32]. When a query is submitted to the broker, strategy tries to find servers which contains most relevant documents based on similarity between the query-clusters and query. Although this policy might be computationally efficient, it is limited by limitation of LSI which fails to address polysemy.

2.4.2 Optimizing Term Partitioning Approach

For designing an optimized TP based IR system, the major goal is to partition the index such that the load is equally spread across all the available servers; it must support scalability and availability; it lowers the cost due to intercommunication; and the effectiveness of results is not compromised. Moffat *et al.* [29] presented a new pipelining architecture based on an optimized TP approach. In this architecture, queries(batch of query) are processed in term at a time basis. As each query at any given time is processed by only one node, pipelining architecture does not require intra-query communication, resulting in higher response time.

Moffat *et al.* [29] concluded that a lack of natural load balancing in the pipelined approach is a serious bottleneck and a random assignment of the terms to the servers is a risk to the performance. To address this problem, the authors [29] showed that it is possible to balance the load by exploiting the information on the frequency of the terms occurring in the query log for distributing terms among the search-servers and for postinglist replication. They considered the problem of partitioning the vocabulary in a TP as a bin-packing problem (the bin packing problem is NP hard), where each bin represents a partition, and each term represents an object to put in the bin. Experimental results shows that the performance of a new pipelining approach benefits from the strategy since it is able to distribute the load on each search-server more evenly than a traditional TP. However, it is not good enough to outperform DP. It also has some advantages over DP as it offers efficient memory utilization. Experiments also shows that DP achieves higher throughput than the pipelining approach.

Perego *et al.* [26] showed that the knowledge mined from the query logs can be used to feed the objective function for partitioning terms among the servers. The original bin-

packing problem simply aims at balancing the weights assigned to the bins. In this case, the objective function depends both on the single weights assigned to the terms (the objects) and on the co-occurrences of the terms in the queries. The main goal of this function is to assign co-occurring terms in the queries to the same index partition. This reduces both the number of servers queried and the communication overhead. However, like TP, this approach is not scalable as it requires building a central index.

2.5 Hybrid Partitioning

Kane and Tompa in [22] presented a comparative study between DP and TP and also introduced a new hybrid partitioning technique. In their hybrid approach, machines are split into groups and document distribution is used between the groups and term distribution is used within the group. We refer this hybrid approach as *ak-hybrid* approach.Simulation experiments showed that the ak-hybrid approach performed better than the TP and DP approach. Like TP, ak-hybrid approach also requires entire postinglists of the query terms to be transferred over the network within the group. Thus such approach cannot guarantee high performance for a large data collection. Nevertheless, simulation does not take in account the effect on the performance due to concurrent (multithreading) query requests. The performance of multithreading is not only affected by the overlapping of memory latency with useful computation, but it also strongly depends on the cache behavior and the overhead of multithreading (e.g., thread management and context-switch costs). In particular, multithreading affects the behavior of the caches and intercommunication, and thus, the overall performance in a nontrivial fashion.

To counter the problem of load imbalance and to improve performance, Xi *et al.* [41] proposed new hybrid partitioning approach in which the inverted list (postinglist) of a term is divided into chunks of equal size[1] and are randomly distributed among the nodes as shown in the Figure 2.1. The query evaluation requires fetching all the chunks of all the query terms by centralized machine, the broker. There is no clear advantage of such an approach as the broker is still a uni-processor, processing all the queries and requires all the chunks of all the query terms. Such an approach cannot guarantee high performance.

Figure 2.1 shows that all the chunks are required by the broker for query processing in the case of their hybrid partitioning approach. For instance, query with terms a, b requires transfer of inverted list from nodes 1,2,3, and 4 to some centralized machine (say a broker) for the query processing.

With the above discussion, we aimed to discuss our index-partitioning strategy based on [22] in details in the next chapter.

[1] All chunks are of equal size except for the last chunk which can be smaller.

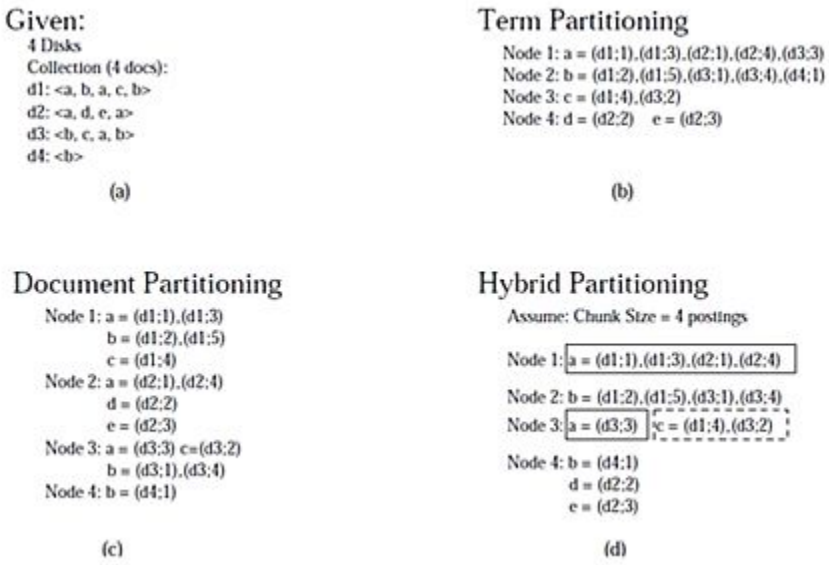

Figure 2.1: Inverted index partitioning schemes [41].

Chapter 3

Hybrid Inverted Index Partition

In Section 3.1 and Section 3.2 of this chapter, we present and explore an alternative approach to index distribution, which tries to address the shortcomings of the term and document partitioning presented in Chapter 2. In Section 3.3 we presents various design knobs that affects the performance of these index partitioning approaches.

3.1 Document over Document Partitioning (DOD)

The *document over document* (DOD) partitioning approach is based on the DP approach, but requires some of the search-servers to take extra responsibilities which relieves some load on the broker. In DOD, we divide the search-servers in groups[1] (sub-clusters) and the data collection in smaller sub-collections. Once a sub-collection is assigned to each group, the DP approach is adopted to divide documents among the search-servers in that group. For instance, if we have total of 12 nodes in the cluster with a 300GB data collection, then we can divide the nodes into 3 sub-clusters (4 search-servers each) and the data collection into 3 sub-collections (100GB each). Each sub-collection is assigned to a group is document partitioned among the search-servers in that group. Each search-server creates an index for the assigned documents and is responsible for evaluating queries against that index.

In each sub-cluster, one of the search-servers acts as a head-search-server[2]. The head-search-server takes extra responsibility of forwarding a query received from the broker to the other search-servers in the group and sending the top results received from the search-servers back to the broker.

[1] Group is interchangeably used with sub-cluster.

[2] It is possible to design a system where different query might have different head-search-server.

Once the broker receives the query issued by the user, it is forwarded to the head-search-server of all the participating sub-clusters. The head-search-server further broadcasts the query to the search-servers. The search-server evaluates the query and sends the top-l most relevant query results[3] back to the head-search-server of their group. After receiving the results from all the search-servers, the head-search-server forwards the top-r results to the broker. Once the broker receives results from all the sub-clusters, it computes a presentable list of top-k documents and forwards it to the user. The steps stated below provide the details of the DOD approach to distributed query evaluation and Figure 3.1 illustrates how the query might be routed through the cluster.

1. Broker receives a query submitted by the user.
2. Each incoming query received by the broker first undergoes text-transformation, i.e. parsing, removal of stop words, and stemming. This transformed query is then forwarded to the head-search-server of all the sub-clusters.
3. Each head-search-server that receives the query performs the following tasks:
   - obtains the information about active search-servers in the sub-cluster, and
   - broadcasts the query to all the active search-servers in the group.
4. After receiving the query, each search-server
   - fetches the postinglists for all the query terms stored in the index,
   - computes the list of documents containing those terms and calculates the similarity score, and then
   - forwards a list of top-l documents to the head-search-server.
5. Once the head-search-server receives the list of top-l documents from all participating search-servers for the query, it computes a list of top-r documents and forwards it to the broker.
6. After receiving top-r results from all sub-clusters, the broker prepares a list of top-k documents and forwards it to the user.

As a derivative of the DP approach, DOD inherits all the benefits of the DP approach. For instance, DOD supports scalability, availability, maintainability, and even distribution of load among the search-servers. However, the DOD tries to address some of the limitation of the DP approach. Mainly, intercommunication cost and load on the broker is significantly less in the case of DOD as a result of two factors. First, instead of all the search-servers

[3]The value of top-k and top-r are the same whereas the value of top-l is less than top-r.

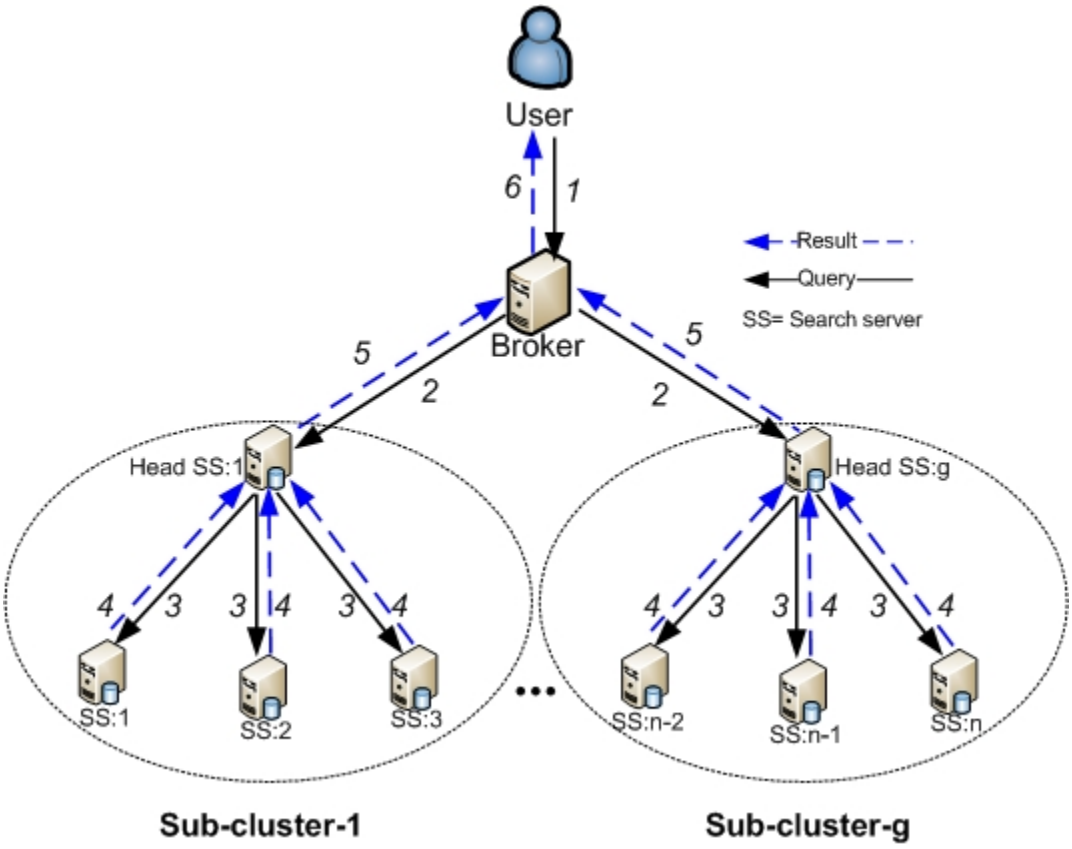

Figure 3.1: Query evaluation in DOD.

as only the head-search-server of each sub-cluster forwards the list of results to the broker, there is a significant decrease in the amount of data to be transferred to and processed by the broker. Secondly, the load on each search-server is also reduced as each search-server is not responsible to generate top-k results. On the other hand, the addition of an extra layer between the broker and the search-servers results in extra overhead. However, this overhead is the tradeoff made by the DOD design to gain performance advantage when the workload is high as show in our experimental evaluation.

3.2 Document over Term Partitioning (DOT)

The term partitioning approach is practical for small data collections but becomes impractical if the data collection is too large, as the query processing cost is dominated by the network cost. On the other hand, most widely used document partitioning techniques become inefficient if the ratio "$(document\ collection\ size)/(number\ of\ servers)$" is too small and disk accesses dominate the query processing cost [36]. Thus, the DOT design adopts a middle ground approach as does the ak-hybrid approach [22], by first dividing the larger data collection into smaller sub-collections, i.e. document partitioning, and then applying the term partitioning on all the smaller sub-collections.

Like ak-hybrid approach, the DOT scheme splits a set of search-servers into groups and then performs document partitioning between the groups and term partitioning within each group. For instance, if we have a total of 12 nodes supporting a 300GB data collection, then we can first divide the nodes into 3 sub-clusters (4 nodes in each sub-cluster) and document partition the data collection into 3 sub-collections (100GB each). Each group creates an index for its assigned sub-collection. This index is then range-partitioned by terms among the search-servers of that particular sub-cluster. As this approach is based on both the term and document partitioning approaches, we also refer to it as the *Hybrid Inverted Index* scheme. Each search-server is responsible for evaluating the query against the assigned range of term index. One of the search-servers in each group acts as the head-search-server. The head-search-server takes up the extra responsibilities of forwarding the query to other search-servers in the group and evaluating query after receiving postinglists from the participating search-servers.

Although DOT is based on the ak-hybrid approach there are many differences between the two. In the case of DOT, one of the search-server acts as a head-search-server in each sub-cluster where as in the case of ak-hybrid there are none. In the case of DOT, small chunks of pre-processed postinglists are transferred over the network to the head-search-server for query processing whereas in the case of ak-hybrid approach, entire postinglists are transferred to one of the search-server. In DOT, broker management is simple as the broker simply broadcast the query to all the head-search-servers whereas in the case of the

ak-hybrid approach, broker has to compute the routing list for each query and for each sub-cluster.

In the case of DOT, once a query is received from the user, the broker forwards the query to the head-search-server of each group. Each head-search-server extracts information about the query terms from the vocabulary table and prepares a routing list of search-servers responsible for those terms. The head-search-server forwards the query to the search-servers on the routing list. Upon receiving the request, the search-server forwards the postinglist(s) (of specific size) for the requested term(s) to the head-search-server. After receiving all the requested postinglists, the head-search-server evaluates the query and generates a list of top-r documents. This list is then forwarded to the broker. After receiving $g * top - r$ documents where g is the number of groups or sub-clusters in a cluster, from all participating head-search-servers, the broker sends a list of $top - k$ results to the user. The steps stated below provides the details of the DOT approach to a distributed query evaluation and Figure 3.2 illustrates how the query might be routed through the cluster.

1. Broker receives a query submitted by the user.
2. Each incoming query received by the broker first undergoes text-transformation, i.e, parsing, removal of stop words, and stemming. This transformed query is then forwarded to the head-search-server of all the sub-clusters.
3. Each head-search-server that receives the query performs the following tasks:
   - obtains the information for all the query terms from the vocabulary table to identify the search-servers that store the relevant index information and prepares a routing list of search-servers storing those terms, and then
   - forwards the query to the active search-servers on the routing list in the sub-cluster.
4. After receiving the query, each search-server fetches the postinglists for all query terms stored in the index and calculates the partial score. After sorting the list by score, one of the following steps is carried out:

   Case-1: If the search-server contains only one query term then it forwards the postinglist of the query term to the head-search-server as per the specified accumulator limit.

   Case-2: If the search-server contains more than one but not all query terms, then a partial similarity check is performed to generate a common list of documents. This common list, which is constrained by the specified accumulator limit is then transferred to the head-search-server.

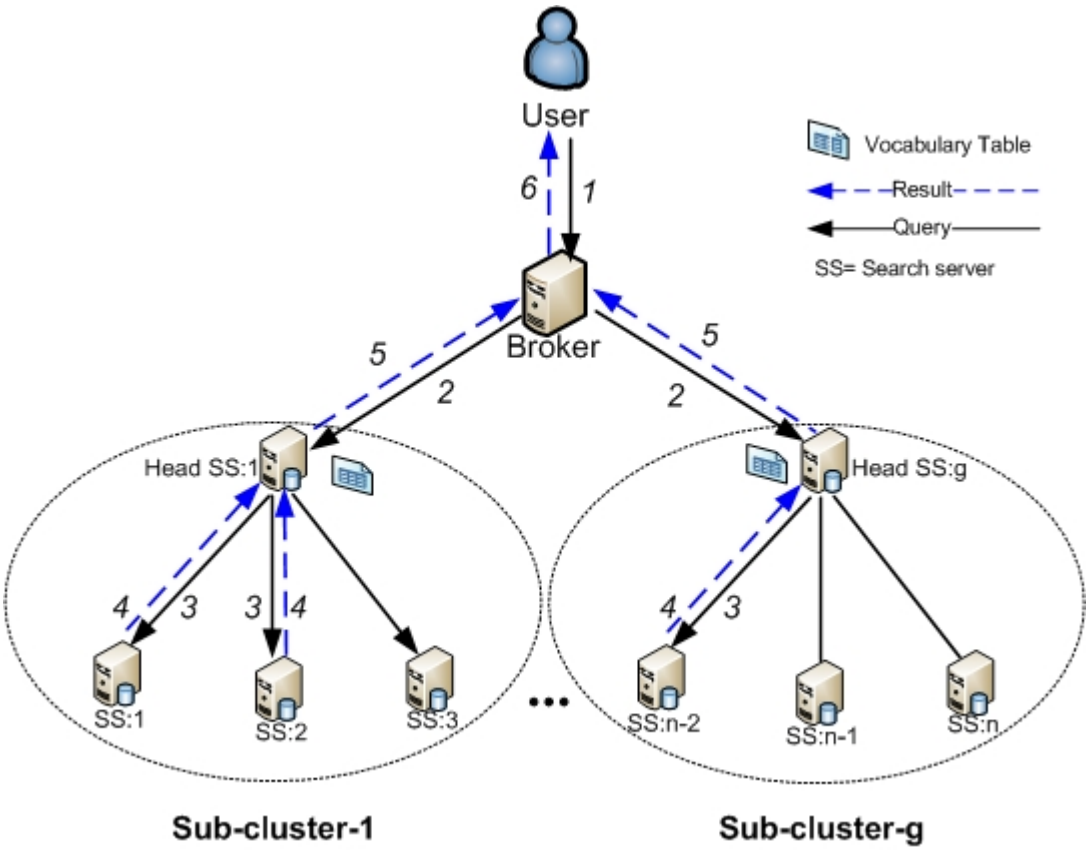

Figure 3.2: Query evaluation in DOT.

Case-3: If the search-servers contain all the query terms, then it will compute the similarity using those postinglists and transfers the top-r most relevant query documents to the head-search-server.

5. Each head-search-server either receives a list of final top-r documents or postinglists from all the participating search-servers. If a head-search-server receives the final top-r documents, then they are directly forwarded to the broker. Otherwise, the head-search-server sends top-r documents after performing a similarity check[4] on all the postinglists of the terms in the query.

6. After receiving top-r results from all the sub-clusters, the broker prepares the list of top-k documents and forwards it to the user.

[4]Comparing postinglists to find common list of documents

3.2.1 Advantages of DOT over TP and DP

The DOT approach retains the benefits of the TP and DP approaches. However, the more important question is how DOT remedies the limitation of both the approaches. We discuss the advantages of DOT over the two other approaches below.

As compared to TP where there is a risk that the broker or the network will become a bottleneck, DOT reduces this risk as a result of two factors. First, as each sub-cluster deals with only a small portion of the index, the size of the postinglists which are required to be transfer over the network for query processing is small. This reduces the traffic on the data communication links. Secondly, the load on the broker is significantly lower as the head-search-servers takes up the responsibility of performing similarity computation between the postinglists of the query terms. The network can still become a bottleneck.

In the case of DOT, as all the sub-clusters participate in the query processing, overall data transfer in the entire systems is still the same. To reduce the amount of data to be transferred, only top x% of the documents from the postinglist of a query term is transferred. This process is repeated until top-k results are obtained for the sub-cluster (still in the worst case, the entire postinglist might be transferred). This reduces the network overhead and the load on the head-search-server without compromising effectiveness.

In TP based IR systems, performance risk due to load imbalance is high. This risk is minimized in the case of DOT as each sub-cluster is responsible only for a small part(sub-collection) of the data collection and the query is processed in parallel against the assigned sub-collection. This results in a distribution of query processing load among the sub-clusters. Further, unlike TP, a search-server processes the document list before it is transferred over the network (if all or more than one query terms are present on the search-server). Thus, in the case of DOT, the search-server is more than just a disk controller. However, DOT minimizes the effect due to a load imbalance, but cannot eliminate the problem. This problem can be addressed by partitioning the index based on a query log and will be addressed in future work.

Moreover, as each search-server is responsible for a certain set of terms T_n from the vocabulary, unavailability of a search-server halts the query processing for the terms stored on the unavailable server. This is a big risk for TP but not for DOT as query processing can still function at the other sub-clusters with the loss of some effectiveness. Again, unlike TP, the addition of new data or updating existing data on existing search-servers will only effect search-servers of the sub-cluster (in which data is added or updated). Thus, like DP, DOT supports scalability and a dynamic index without the loss of availability.

Unlike DP, in DOT, resources are more efficiently utilized, since query is only forwarded to the search-server(s) which possesses required index for query processing.

3.3 Design Knobs

To improve the overall performance of IR systems, various design strategies are adopted such as replication, caching, data collection partitioning, collection selection, etc. However, performance of DOD and DOT index partitioning strategies can also be improved by tuning various configurable parameters or *design knobs*. The value of these parameters depend on various factors such as data collection size, size of the cluster, and workload. By configuring these design knobs, we would like to study their effect on the performance and also how sensitive the technique are to changes in these parameters. Some identified design knobs for DOT and DOD are listed below.

- **Number of search-servers:**
  Considering that the number of search-servers is constant in the cluster, an increase or decrease in the number of sub-cluster will affect the size of the sub-clusters. An increase in the number of sub-clusters will increase the computing load on the broker, but will decrease the load on the head-search-server. The size of the sub-cluster has a sparse affect on the performance making it an important design parameter. An increase or decrease in the size of a sub-cluster affects various system parameters, but mainly the load on the head-search-server and the volume of communication. Normally, we expect that with the increase in the number of servers, throughput increases and latency decreases. However, the cost due to communication and load on the head-search-server will also increase with the increase in the number of search-servers. For instance, in the case of DOT, with an increase in the number of search-servers, distribution of the term will get more skewed resulting in a further increase in communication cost. This will increase the load on the head-search-servers as it is required to do more similarity based computation. In the case of DOD, as all the search-servers participate in the query processing, the time to evaluate query is determined by the slowest machine. Thus, adding more servers increases the variance between the average and worst performing servers. Also, the load on the head-search-servers increases since it needs to process results received from all the search-servers in the sub-cluster. Another factor which affects the size of the sub-cluster is the size of the data collection. Over assigning servers to a smaller data collection will not necessary improve the performance. Thus, the size of the sub-cluster is directly proportional to the size of the data collection assign to that sub-cluster.

- **Number of results per query per search-server:**
  An important question is, how should the per search-server result set size k be chosen with respect to the number of search results m requested by the user. To guarantee top results, the value of k usually equals m. However, the number of results returned by each search-server has a non-negligible effect on the network, query processing

(on the search-server), load on the head-search-server and load on the broker as a result of two factors. First, if a user has asked for k results, it is highly unlikely that all of the top-k results come from the same search-server. Since, as the number of results per search-server increases, the amount of work per query increases though at different rates. This increase in the work load on a search-server is due to the performance heuristics like MaxScore[5]. Second, by making each search-server return the top-k results, one incurs more load on the network. This design parameter has comparatively less influence on DOT partitioning as only the head-search-server returns the top-k result to the broker.

- **Number of concurrent query requests:**
  Multithreading has emerged as one of the most promising and exciting techniques used to achieve high performance while using resources more efficiently. However, the performance of multithreading is not only affected by the overlapping of memory latency with useful computation, but also strongly depends on the cache behavior and the overhead of multithreading (e.g. thread management and context-switching costs). In particular, multithreading of query requests affects the behavior of caches and thus the overall performance in a nontrivial fashion. Thus, we would like to study how significant are the improvements and to what design knobs is the multithreading sensitive.

- **Accumulator limit for DOT:**
  In the case of the DOT based IR system, the postinglists of terms are required to be transferred over the network for query processing. Transferring the entire postinglists of terms will significantly affect the performance (in terms of throughput and latency) as the cost due to communication and load on the head-search-server will increase. Deciding the size of postinglist that needs to be transferred over the network for similarity computation without loss of effectiveness is a challenging problem. The database research community have long studied the issue of efficient processing of top-k queries. The TPUT algorithm [11] proposed by Cao and Wong addresses this problem. The algorithm uses three phases in order to find the k objects with the highest aggregate value for the query in the distributed environment. Trace-driven study shows that the traffic of TPUT is a few magnitude less than existing algorithms like Threshold algorithm [15]. Such data pruning technique that reduces network load can be applied to our approach, however we opted for simple naive method to avoid complexity. Setting the postinglist transfer size too small will affect the effectiveness and setting it too high will waste resources. Thus, the lower the value of this design knob, higher the efficiency and lower the effectiveness. To reduce the gap between

[5]MaxScore is a threshold method to compare the maximum score that documents could have in the final list of results [36].

efficiency and effectiveness, we transfer the top x% of list of the documents containing the terms until we get a specific number of results (for that sub-cluster).

In the next chapter, we present several experimental results obtained by employing our approach and we try to analyze these design knobs in light of those results.

Chapter 4

Experiment

In this chapter, we present several experiments designed to study the effectiveness and efficiency of our approaches. We want to know whether our index-partitioning strategies help in lowering the cost when compared with the TP (term partitioning) or DP (document partitioning) approaches. If so, we would also like to study how significant are the improvements and to what design knobs are the technique sensitive.

4.1 Hardware and Software

The hardware used for all experiments reported in this thesis is a Beowulf-style cluster of 16 nodes called Shiraz. Shiraz is comprised entirely of Sunfire X4100 servers. Each node is comprised of two dual-core AMS Opteron 280 processors with 8GB of RAM and two 72 gigabytes (GB) disks, connected via 1Gbit network. The local disks operate in RAID-0 (mirror) configuration to maintain availability of the node during disk failure. A NAS with 6 internal and 12 external disks are used to provide extra storage of 1.8 terabytes (TB) shared between all the nodes of the cluster. The nodes run the OpenSuSE 10.1 operating system.

The Nutch 1.0 search engine application was used to implement all our techniques. Nutch is an open source search application implemented in Java and is based on the Lucene library. Nutch consists of all three of the major components of a search application, i.e. crawling, indexing and query processing. More information about Nutch can be found in [27].

4.2 Test Data

The data collections use in our experiments are derived from the GOV2 collection. GOV2 is a TREC test collection built in early 2004 by crawling Web pages and documents from .gov domain of the US government. The collection is 426GB in size and contains 25 million documents [12]. As the index is stored in main memory instead of on the disk of a server, the GOV2 data collection was reduced to 224GB. The 224GB data collection is referred as DC/01. Dividing DC/01 into half gives the data collection of size 112GB and is referred as DC/02. Similarly, data collection of size of 28GB is referred as DC/08 and 56GB as DC/04. Table 4.1 gives information about the size of the indexes and number of documents in each data collection. Round robin approach was adopted to partition each

Table 4.1: Information about index size and number of documents for various sized data collections.

Attribute	Data Collection			
	DC/08	DC/04	DC/02	DC/01
Size (GB)	28	56	112	224
Documents (* 10^6)	1.66	3.24	6.5	13.1
Index(GB)	3.5	6.6	9.9	20

of the above mentioned data collections (i.e. DC/08, DC/04, DC/02, and DC/01) to form sub-collections. The number of sub-collections equals the number of search-servers in a cluster. For instance, to create 10 equally sized partitions for 10 search-servers from the DC/02 (112GB) data collection, every 10th file of the collection was extracted into the first partition, every tenth plus one file into second collection, and so on. Such data partitioning methods results in a homogeneous spread of data between the sub-collections [29].

4.3 Test Queries

A query is essentially a sequence of tokens or phrases that are combined together using boolean operators. The Million Query (1MQ) Track was used to evaluate all approaches. The track contains ten thousand (10,000) queries, including 264 queries that overlap with those used by the relevance feedback track. Each query was known to have had a *.gov document clicked on after the query was issued. This means there is some evidence that the query will have relevant documents in the GOV2 collection. Since the data collection was reduced to 224GB we cannot use the entire set of 10000 queries as some queries would not generate any results. Thus we run all 10000 queries against the index of the DC/01 collection and randomly select 5000 queries which have relevant documents in the data

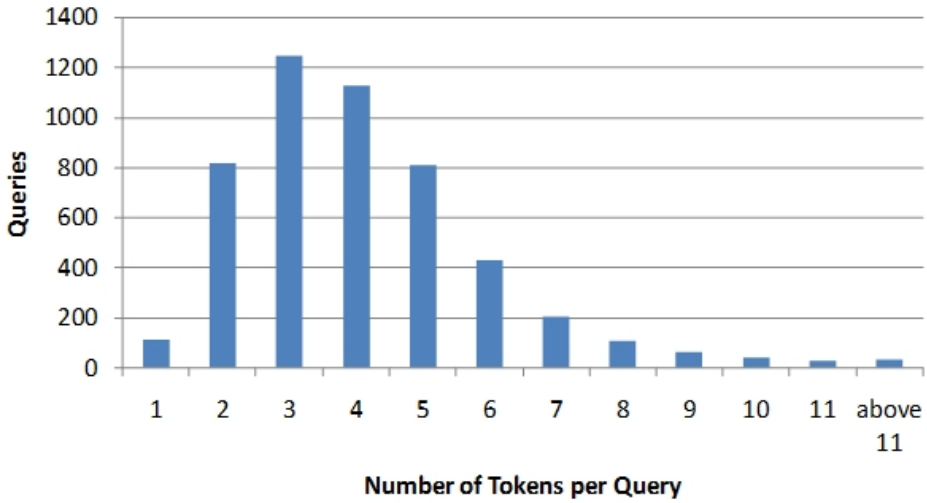

Figure 4.1: Distribution of tokens per query for MillionQuery TREC.

collection. Just to get an idea, Figure 4.1 highlights the number of tokens per query distributed for 5000 queries from the Million Query Track.

4.4 Accumulator Limit

In the TP based IR system, postinglists of terms are required to be transferred over the network for query processing. It is challenging to decide the amount of the postinglists that need to be transferred over the network for a similarity computation to have no loss of effectiveness. One may argue that it should be specific to the number of results you are expecting from the system. But that alone does not help since we are still required to transfer postinglists of some size to generate the specified number of results. Transferring the entire postinglists of terms will significantly affect the throughput and increase query latency. Further, setting the postinglist transfer size too small will affect the effectiveness of results and setting it too high will use resources redundantly. Nicholas *et al.* [25] found that the GOV2 data collection requires a postinglist transfer size of approximately 400000, when accessed using 2004 TREC Terabyte topics and mean average precision. Through experiments they confirmed that the loss in the *effectiveness* when the list size is reduced to 100000 is insignificant. Moffat *et al.* [29] also carried out a similar study by comparing different index partitioning models using the notion of ranking dissimilarity. Their study confirmed the limit of 100000 as suitable conservative value with very small variance in ranked results. Based on the above mentioned study, we took 100000 as our starting point for postinglist size. The next question is what happens when we reduce this limit as the

volume of the data collection reduces from DC/01 to DC/08. It seems reasonable to vary the limit in proportion to the volume of a data collection.

4.5 Measurement

Two primary metrics to evaluate search are *effectiveness* and *efficiency*. Effectiveness measures the ability of a search engine to find user intended information and efficiency measures how quickly we can get this information. In this thesis, we present performance from the perspective of efficiency while maintaining effectiveness prospective of Nutch. It is very important to determine exactly what aspect of efficiency we want to measure and how we want to measure it. Two of the most commonly used efficiency metrics are *throughput* and *latency*. Throughput refers to the number of queries executed in a unit time. Query latency is the amount of time a client waits after issuing a query before receiving a response. As stated in [16, 40], throughput values are comparable only if the same collection and queries are processed on the same hardware. As the number of search-servers may vary independently of the data collection and index size, more meaningful metrics are required to compare like with like. Hence, to facilitate meaningful comparisons, throughput values reported in all the subsequent experiments follow the metric *terabyte index queries per machine second* which is known as *normalized throughput*. That is, if n processors are able to handle q queries in s seconds for the index of size iT Terabytes, then useful work done is measured as $(q * iT)/(s * n)$. Larger rate means higher throughput and thus better performance. Further, average response time in second can be given by $(t * s)/q$, where t is the number of queries running concurrently system wide. Such analysis between workload, computing power available and time taken to process data to complete the task provides strong baseline for the investment. Even better performance metrics will also account for other cost such as elapsed indexing time, temporary storage space while indexing, cooling cost, and software costs. But this would complicate it even more and thus we stopped at the index size and number of search-servers.

Various experiments were carried out to get insight on the performance of various approaches i.e. DP, TP, DOD, and DOT are as follows:

- Increasing the size of the collection while keeping the number of search-servers and concurrency level constant will allow us to quantify the effect of data growth on the throughput.
- With the number of search-servers and data collection size held constant, increasing the number of simultaneous query requests will allow us to quantify the maximum concurrent requests the system can handle efficiently.

- Keeping the size of the data collection constant and increasing the number of processor will allow us to identify overhead due to interprocess communication.
- Scalability of the method can be evaluated by varying the number of processors and volume of a data collection in proportion.

After deciding what to measure, the next question is how to measure. The best approach would be to measure the performance of the system in the *steady state*. In our case, the steady state is reached as soon as a specified number of queries are concurrently running system-wide and ends when specified number of queries were executed. For instance, if the concurrency (threading) level for the experiment is set to 500 and number of queries to be executed is set to 2000, steady state is reached as soon as we have 500 active queries running system wide and ends when 2000th query is executed. The time elapsed before we reached steady state is ignored. In the case of steady state, the results of the initial part of the experiment are not included, a process is known as *transient removal* [20]. There are many methods suggested in the literature for transient removal and most of them are based on some heuristics. Our experiment shows that information system based on in memory index, transient time is approximately 0.25% to 1% of total execution time and depends on the concurrency level. Thus, the transient time for memory based IR systems is very low compared to disk based IR systems [37]. All the experiments were run on the same set of 5000 queries derived from the Million Query Track and each experiment is run 5 times and the mean of the experiments is reported in this thesis. Further, to study the performance of the approaches in a pure form, the caching of the query results was not done while performing experiments.

A Comparative study between index partitioning approaches in terms of scalability and utilization of resources is carried out using *efficiency* and *speedup* metrics respectively.In distributed and parallel systems, efficiency ratio quantifies the number of valuable operation performed by machine while evaluating queries in parallel and is give by:

$$E = S/n$$

where S is speedup and n is number of search-servers. Speedup is used to express how many time a parallel approach works faster than the best sequential approach to solve the same problem. Speedup is given by

$$S = t_{sequential}/t_{parellel}$$

where $t_{sequential}$ is time taken by sequential approach to execute task ant $t_{parellel}$ is time taken by parallel approach to execute task.

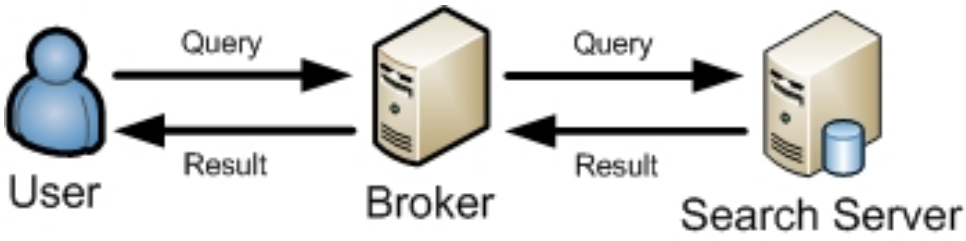

Figure 4.2: Query evaluation in a monolithic search engine.

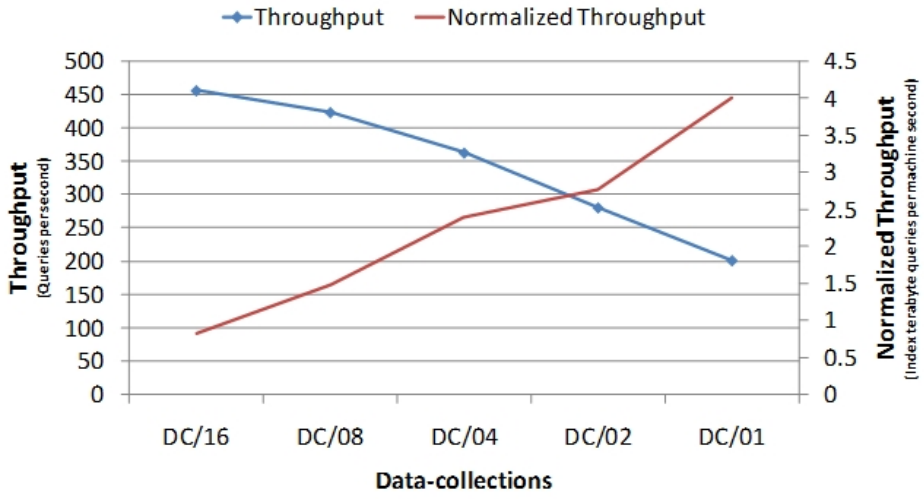

Figure 4.3: Performance of Monolithic Search Engine.

4.6 Evaluating Monolithic Architectures

To experiment with the monolithic architecture, we first configure Nutch as shown in the Figure 4.2. A set of 5000 queries was executed sequentially on a single search-server. In order to make meaningful comparisons with other approaches, the index against which the search operation is performed is stored in the main memory of a single search-server.

In Figure 4.3, we observe that with the increase in the volume of a data collection, throughput decreases and normalized throughput increases[1]. This gradual increase in normalized throughput clearly indicates the efficient utilization of resources. These numbers are later used to calculate efficiency and speedup ratio.

[1] As index size of DC/02 and DC/01 cannot fit into the memory of a single search-server, we extrapolated the results.

4.7 Effect of Threading on DP, DOD and DOT

Threading has emerged as one of the most promising and exciting techniques for exploiting parallelism. To investigate the effect of concurrency on throughput, we ran a set of experiments with DC/01 collection and 12 search-servers[2]. Throughput and normalized throughput was recorded for different values of t, where t is the number of queries executing concurrently system-wide. In the case of DP and DOD, since all servers participate in query processing, the number of threads running on each server is potentially t. However, in practice, because some threads finish their tasks before others, the actual active load per server is less than t (we recorded it around 75% of t). In the case of DOT, the server(s) are only involved in query processing if it contains index for one or more query terms. Thus, the number of threads running on each server is comparatively very low.

This experiment helps to identify the appropriate threading level necessary to achieve high throughput and also helps us to set concurrency level for subsequent experiments for all approaches.

Table 4.2: Normalized throughput for DP, DOD, and DOT. Set of experiments carried out by varying concurrency level from 1 to 500; number of search-servers n=12 and data collection DC/01. All values shown are mean over 5 runs and were recorded when the system was in steady state.

Concurrency Level (t)	DP	DOD	TP	DOT
1	0.47	0.29	0.005	0.028
2	0.72	0.43	0.006	0.031
25	0.83	0.93	0.007	0.079
50	0.94	0.94	0.007	0.082
100	0.92	0.97	0.007	0.083
250	0.90	1.00	0.007	0.082
500	0.82	0.96	0.007	0.080

From Table 4.2 and Table 4.3, it is clear that in the case of DP, DOD, and DOT, throughput and normalized throughput increase with the increase in the concurrency level and peaks at t= 50, 250, and 100 respectively. From Table 4.3, it is clear that due to concurrency, DP's throughput rises by 99%, DOD's by 247%, and DOT's by 154% when compared to sequential query processing. These results clearly indicate the importance of threading in an IR system. However, concurrency affects the behavior of caches which affects overall performance in a nontrivial manner. Thus, beyond a certain threshold,

[2]For DOD and DOT, 12 search-servers where split into 2 sub-clusters (groups) of 6 search-servers each.

Table 4.3: Throughput for DP, DOD, TP, and DOT. Set of experiments carried out by varying concurrency level from 1 to 500, number of search-servers n=12 and data collection DC/01. All values shown are mean over 5 runs and were recorded when the system was in steady state.

Concurrency Level (t)	DP	DOD	TP	DOT
1	282.5	172.2	3.21	16.7
2	433.8	260.3	3.62	18.5
25	495.7	558.5	4.05	47.4
50	561.2	564.1	4.07	49.1
100	552.7	582.3	4.18	50.0
250	542.4	597.1	4.08	49.2
500	494.6	578.7	3.90	48.1

increase in the concurrency level increases cost due to communication, cache coherence and context switching resulting in a drop in performance.

In terms of approaches based on term partitioning, from Table 4.2 and Table 4.3 it is clear that the DOT approach offers much better performance than the traditional term partitioned method. This is because, in the case of TP, the broker becomes a bottleneck as it is uni-processing all the queries and load imbalance among the search-servers. Further, as entire postinglist of query terms are required to be transferred over the network for query processing, load on the communication link increases. This is not the case with DOT as a result of three factors. Firstly, query processing cost is shared between head-search-servers, broker, and search-server. Thus results in more balanced distribution of load. Secondly, in the case of DOT, search-servers are more than just a disk controller as they participate in query processing. Lastly, search-servers sends per processed list of documents as per accumulator limit instead of entire postinglists.

4.8 Effect of Most Optimal Concurrency Level

From the multi-threading experiment, it is clear that the DP, DOD, and DOT performed better when the concurrency level is set to 50, 250 and 100 respectively. To further investigate the effect on performance of the concurrency level by varying the number of search-servers for each data collection, a set of following experiments were designed.

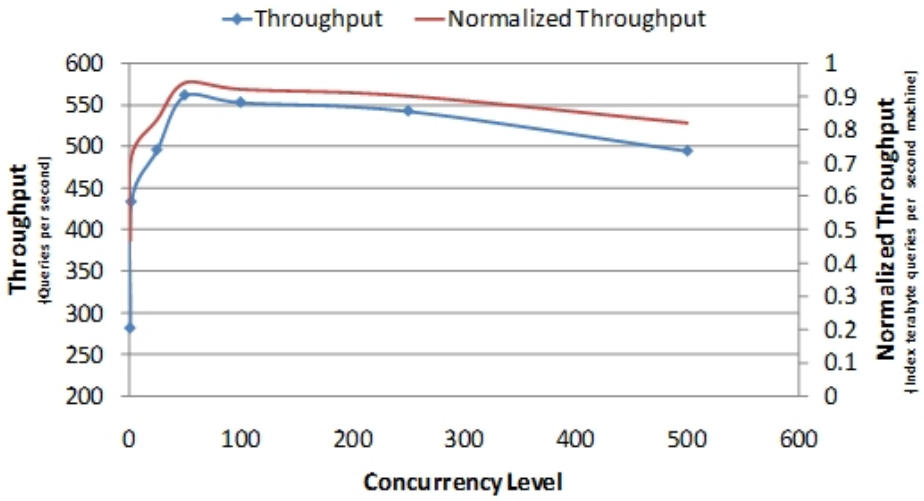

Figure 4.4: Effect of threading on performance of DP.

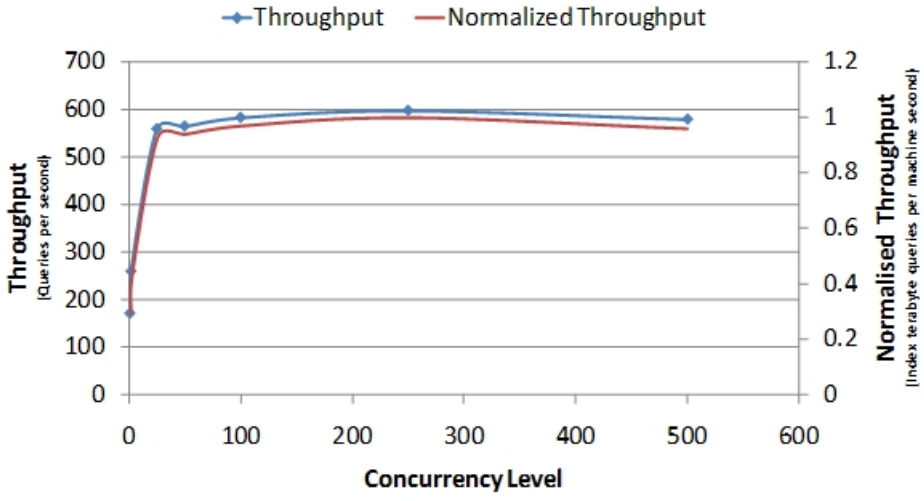

Figure 4.5: Effect of threading on performance of DOD.

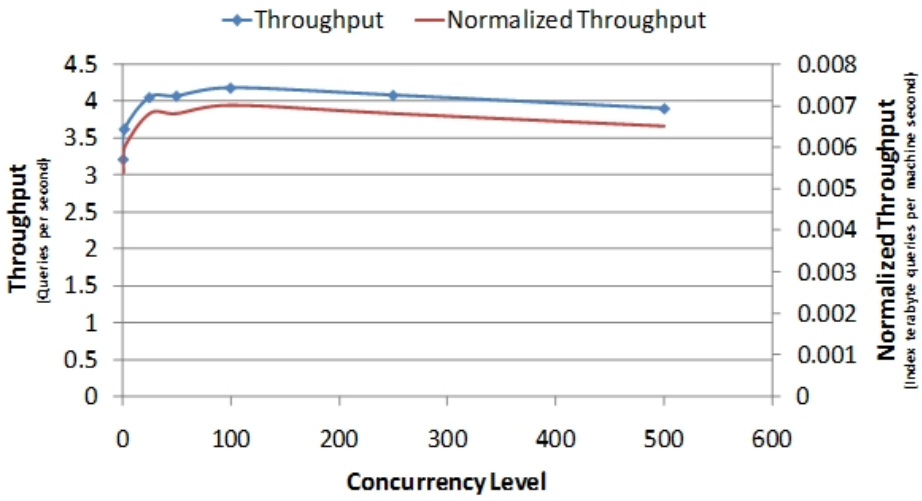

Figure 4.6: Effect of threading on performance of TP.

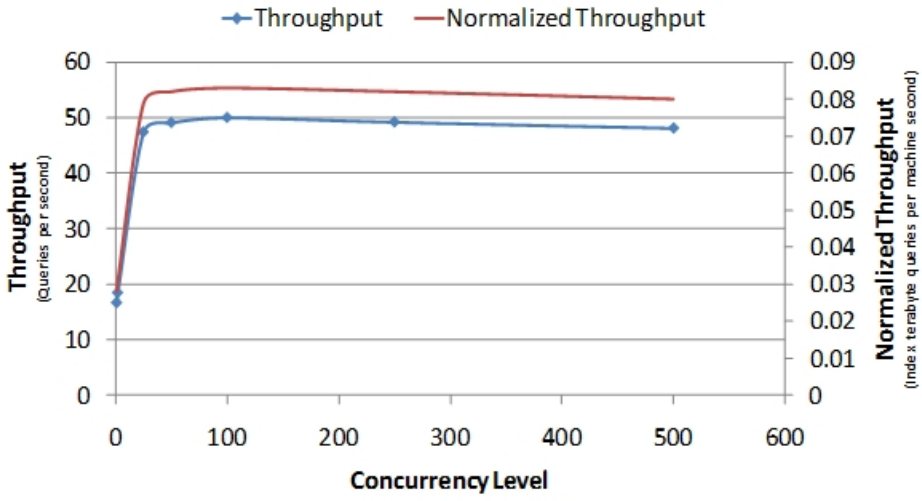

Figure 4.7: Effect of threading on performance of DOT.

Table 4.4: Normalized throughput for DP. A set of experiments were carried out by varying the number of search-servers for each data collection while keeping concurrency level fixed at t=50. All values shown are mean over 5 runs and were recorded when the system was in steady state.

	Number of Search-servers (n)				
Data Collection (GB)	4	6	8	10	12
DB/08	0.66	0.44	0.32	0.21	0.18
DB/04	1.19	0.70	0.52	0.39	0.31
DB/02	1.53	1.14	0.79	0.59	0.48
DB/01	2.53	1.99	1.51	1.16	0.94

DP

The first row in Table 4.4 represents the number of search-servers involved in query processing. The first column in the table represents the data collections. Starting at the (n=4 and DC/08) entry indicates that DC/08 data collection was divided equally among 4 search-servers each consisting of (DC/08)/4 of data.

We observe in Table 4.4 and Figure 4.8 that the normalized throughput decreases with the increase in the number of search-servers. This observation demonstrates that the approach is more efficient and utilizes resources more efficiently when data is stored on fewer search-servers. One of the reasons for the initial increase in the normalized throughput, as explained before, is that with the increase in the number of search-servers, the volume of data assigned to each server decreases, resulting in smaller postinglists and faster query processing. When evaluating queries concurrently, there is an extra cost involved due to thread-management issues like context-switching, race conditions and cache behavior. This cost increases even more when things are done more quickly. Other costs involved in adding extra servers which effect the performance are communication costs and overhead on the broker. Figure 4.8 shows that the addition of more search-servers beyond certain threshold has little effect on the throughput irrespective of the volume of a data collection. Further, with the increase in the volume of the data while keeping the number of servers constant, normalized throughput increases indicating efficient utilization of the resources.

In terms of scalability, looking along the diagonal, normalized throughput tends to be same, which implies that the approach is scalable. For instance, the (n=4, DC/02) entry is same as the entry (n=8, DC/01) in Table 4.4.

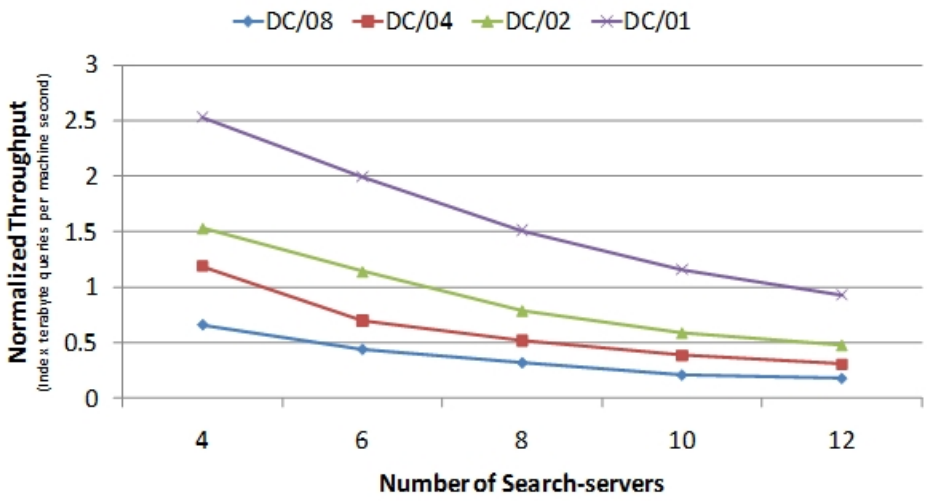

Figure 4.8: Effect of concurrency query request (t=50) on DP.

DOD

To analyze the performance of DOD, we experimented with the configuration as shown in Figure 3.1. First row in Table 4.5 gives information about the number of groups and the number of search-servers in each group. For instance, 2G*3SS is read as: two groups, each consisting of 3 search-servers. One of the search-servers in each group behaves as the head-search-server.

Table 4.5: Normalized throughput for DOD. A set of experiments were carried out by varying the number of search-servers for each data collection while keeping concurrency level fixed at t=250. All values shown are mean over 5 runs and were recorded when the system was in steady state.

	Number of Search-servers(n)				
Data Collection(GB)	$2G*3SS$	$2G*4SS$	$2G*5SS$	$2G*6SS$	$4G*3SS$
DC/08	0.41	0.21	0.09	0.13	0.18
DC/04	0.59	0.35	0.37	0.30	0.34
DC/02	0.84	0.69	0.47	0.30	0.39
DC/01	1.98	1.35	0.77	1.00	0.96

We observe in Table 4.5 and Figure 4.9 that the normalized throughput decreases with the increase in the number of search-servers and then again increases slightly. For instance, in the case of DC/01, normalized throughput (in Figure 4.9) decreases from 2G*3SS to

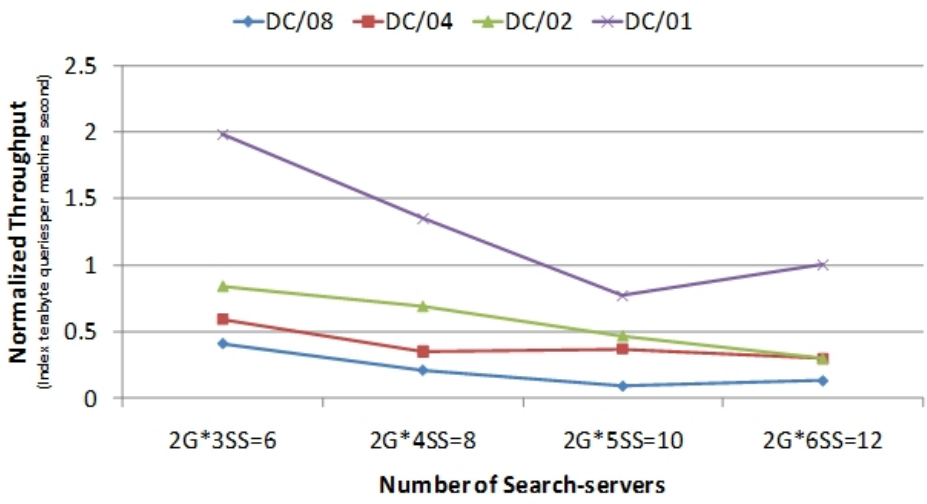

Figure 4.9: Effect of concurrency query request (t=250) on DOD.

2G*4SS to 2G*5SS and then increases for 2G*6SS. At first, this is a surprising result, since as we increase the number of servers, we expect throughput to either increase and then flatten or decrease gradually. Thus, to investigate the matter, we observed the pattern of throughput when concurrency level was set to 1, i.e sequential query processing. We found that during sequential query processing, for instance, in the case of DC/01, throughput increases with the increase in the number of search-servers and peaks at 2G*5SS and then starts to decrease. This pattern is totally reverse of what we observed for the same data collection when the concurrency level is set to 250. This clearly indicates the effect of cache management issues on the performance of the system.

Looking at Figure 4.9, it is clear that the approach is more efficient when data is stored on fewer search-servers because the addition of each search-server results in additional communication costs, some extra cost on the head-search-server, and additional cost due to thread management.

We also observed in Figure 4.9 that when the volume of data grows in proportion to the number of search-servers, normalized throughput increases indicating scalability of the approach. For instance, the entry (2G*3SS, DC/02) is 0.84 which is less than the entry (2G*6SS, DC/01) is 1.

DOT

To analyze the performance of DOT, we experimented with the configuration shown in Figure 3.2. We observe in Table 4.6 and Figure 4.10 that the normalized throughput decreases

Table 4.6: Normalized throughput for DOT partitioned index organization technique. A set of experiments were carried out by varying the number of search-servers for each data collection while keeping concurrency level fixed at t=100. All values shown are mean over 5 runs and were recorded when the system was in steady state.

	Number of Search-servers(n)				
Data Collection(GB)	$2G*3SS$	$2G*4SS$	$2G*5SS$	$2G*6SS$	$4G*3SS$
DC/08	0.110	0.080	0.060	0.050	0.050
DC/04	0.173	0.125	0.084	0.059	0.057
DC/02	0.172	0.104	0.071	0.059	0.051
DC/01	0.199	0.131	0.088	0.083	0.049

with the increase in the number of search-servers. At first, this is a surprising result, since as we increase the number of servers, we expect throughput to increase considering the benefits gained due to concurrency in the term partitioning strategies. Part of the problem is that the head-search-server becomes a bottle-neck. The head-search-server essentially becomes a uni-processor (in the group) processing all the queries as the term index is spread over many search-servers and is skewed. Adding to the problem, the head-search-server has to wait for the postinglists, which is largely affected by the slowest search-server, and also has to deal with thread management issues.

Looking at Figure 4.10, it is clear that our approach is more efficient when data is stored on fewer search-servers because with the fewer search-servers, the possibility of the required postinglists for terms to be present on same search-server is more. If more than one term required to evaluate the query are present on the same search-server, then instead of sending postinglists of all terms present on that search-server, we will only send a common list of documents which contains all the terms, reducing network cost as well as load on the head-search-server.

It is clear from Figure 4.6 that as the volume of data grows in proportion to the number of search-servers, how wasteful this approach became while incorporating concurrency. Although DOT performed better in terms of throughput when compared with the traditional TP approach, it resulted in a loss of scalability.

4.9 Effect of Sequential Query Processing

Concurrency affects the behavior of the cache, and thus the overall performance in a nontrivial fashion. Hence, it also becomes necessary to quantify the performance by running queries sequentially. To study the effect of concurrency on the performance without any

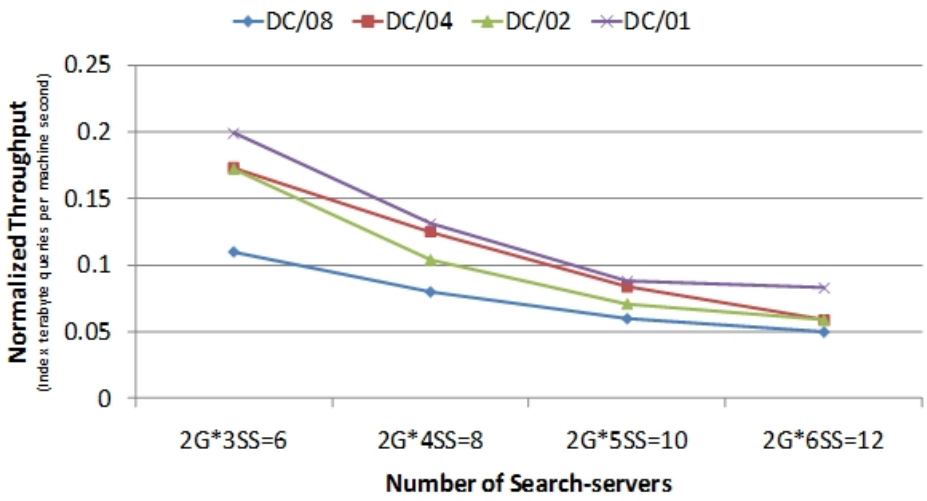

Figure 4.10: Effect of concurrency query request (t=100) on DOT.

thread management issues, we ran a set of experiments varying the number of search-servers for each data collection while keeping the concurrency level fixed at t=1.

This set of experiments will help us investigate the effect on throughput and resource utilization, by varying the volume of data while keeping the number of search-servers constant and vice-versa i.e. by varying the number of search-servers while keeping the volume of data constant. Further, by varying the number of servers and volume of data in proportion will give us insights on scalability. It will also help us find the best configuration for sequential query execution and thus help us investigate how this configuration behaves for different concurrency levels.

DP

To analyze the performance of DP, we experimented with the similar configuration shown in Figure 1.3.

Looking horizontally across Table 4.7 and Figure 4.11, we observe that the normalized throughput decreases with the increase in the number of search-servers. This indicates that the approach is more efficient when data is stored on fewer search-servers. This is because communication cost and overhead on the broker increases with the increase in the number of search-servers. Thus, resulting in inefficient utilization of available resources.

A different way of looking at the table is along the diagonal. When the volume of data grows in proportion to the number of search-servers, normalized throughput tend to be same. For instance, (n=6, DC/08) entry is same as (n=12, DC/04) entry also entry (n=4,

Table 4.7: Normalized throughput for DP. Set of experiments were carried out by varying the number of search servers for each data collection while keeping concurrency level fixed at t=1. All values shown are mean over 5 runs and were recorded when the system was in steady state.

	Number of Search-servers (n)				
Data Collection(GB)	4	6	8	10	12
DB/08	0.33	0.21	0.15	0.12	0.09
DB/04	0.62	0.37	0.30	0.20	0.16
DB/02	0.78	0.53	0.40	0.30	0.24
DB/01	1.38	0.96	0.73	0.59	0.47

DC/02) is approximately same as (n=8, DC/01) entry[3]. This confirms scalability of the approach.

DOD

Table 4.8: Normalized throughput for DOD. A set of experiments was carried out by varying the number of search-servers for each data collection while keeping concurrency level fixed at t=1. All the values shown are mean over 5 runs and were recorded when the system was in steady state.

	Number of Search-servers (n)				
Data Collection(GB)	$2G*3SS$	$2G*4SS$	$2G*5SS$	$2G*6SS$	$4G*3SS$
DC/08	0.14	0.09	0.07	0.06	0.06
DC/04	0.21	0.15	0.12	0.09	0.10
DC/02	0.30	0.23	0.18	0.16	0.18
DC/01	0.57	0.44	0.37	0.29	0.29

Looking horizontally across Table 4.8 and Figure 4.12, we observe that the more efficient metric normalized throughput, decreases with the increase in the number of search-servers within the group from $2G*3SS$ to $2G*6SS$. One of the reasons for poor performance is that sequential query processing increases the communication costs and gains nothing from load sharing between the broker and head search-servers. Further, sequential query processing results into under utilization of the resources as all the servers may not contain

[3]There is some variance in normalized throughput as doubling the size of data does not result in the index size being doubled. For instance, index of DC/08 (which is 3.5GB) is not half of index DC/04 (which is 6.6GB).

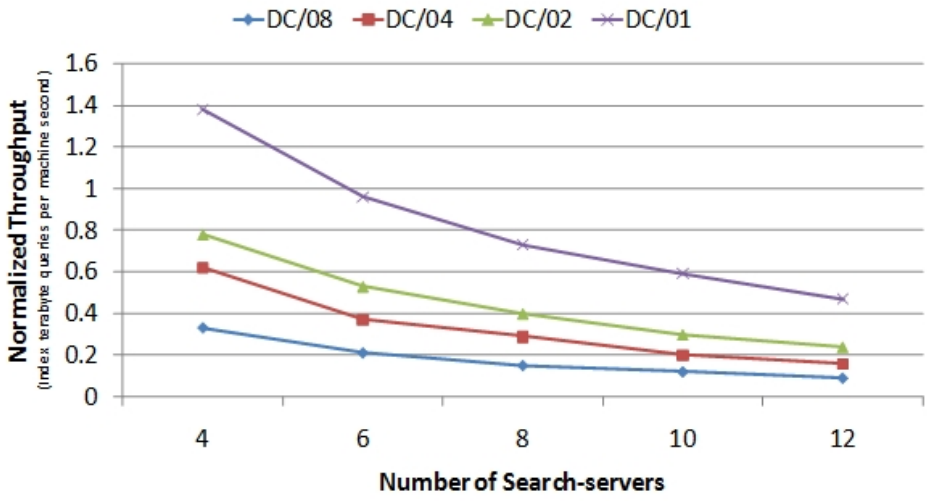

Figure 4.11: Effect of sequential query processing on DP.

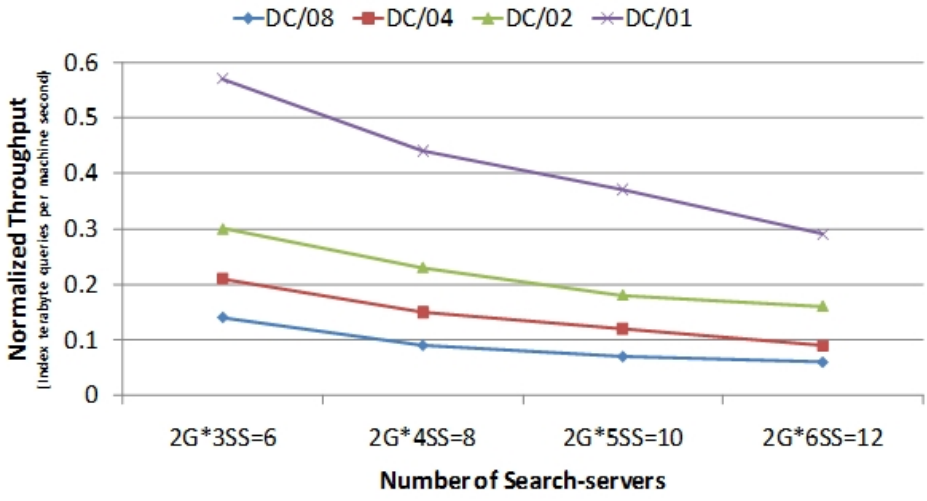

Figure 4.12: Effect of sequential query processing on DOD.

query relevant results. From the Figure 4.12 it is clear that adding more computing power will not help; on the other hand it will suffer from communication costs and increased overhead on the head-search-server of a group. Thus, results in inefficient utilization of resources.

In terms of scalability, as the volume of data grows in proportion to the number of search-servers, the general pattern shows that normalized throughput tends to be the same. For instance, entry (2G*3SS, DC/02) is approximately same as entry (2G*6SS, DC/01). Further, normalized throughput is same when the number of sub-clusters and the volume of data grows in proportion. For instance, entry (2G*3SS, DC/02) is same as entry (4G*3SS, DC/01).

DOT

Table 4.9: Normalized throughput for DOT. A set of experiments were carried out by varying the number of search-servers for each data collection while keeping the concurrency level fixed at t=1. All values shown are mean over 5 runs and were recorded when the system was in steady state.

	Number of Search-servers (n)				
Data Collection(GB)	$2G*3SS$	$2G*4SS$	$2G*5SS$	$2G*6SS$	$4G*3SS$
DC/08	0.020	0.020	0.010	0.010	0.010
DC/04	0.031	0.021	0.014	0.011	0.011
DC/02	0.034	0.022	0.017	0.014	0.014
DC/01	0.056	0.042	0.033	0.028	0.028

Looking horizontally across Table 4.9, we observe that again the normalized throughput decreases with the increase in the number of search-servers (in the group). This is because, with the increase in the number of search-servers, the postinglists to search-servers assignments get skewed. Thus, most of the times, postinglists are sent to the head-search-server for query processing. This results in an increase in the costs due to communication and load on the head-search-servers. Hence, this indicates that the resources are more efficiently utilized when data is stored on fewer servers.

In terms of scalability, as the volume of data grows in proportion to the number of search-servers, the general pattern shows that the normalized throughput tends to be constant. For instance, in Table 4.9, entry (2G*3SS, DC/02) is approximately the same as the entries (2G*6SS, DC/01) and (4G*3SS, DC/01) indicating the approach is scalable. Further, normalized throughput tends to be the same when the number of sub-clusters and volume of data grows in proportion. For instance, in the Table 4.9, entry (2G*3SS, DC/02) is same as entry (4G*3SS, DC/01).

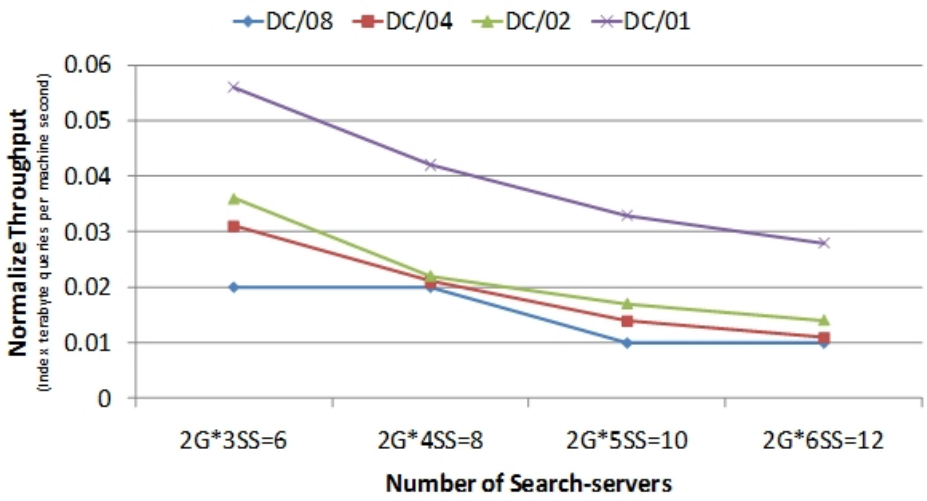

Figure 4.13: Effect of sequential query processing on DOT.

4.10 Comparison between Index Partitioning Approaches

In this section, we compare all methods (DP, DOD, and DOT) with respect to high performance, scalability, resource utilization, and availability. The efficiency ratio is used to compare the approaches in terms of resource utilization and speedup performance metric is used to prove scalability of the approach. Efficiency and speedup performance metrics are explained in detail in Section 4.5. Efficiency ratio is per search-server and should not be confused with the entire system (cluster).

4.10.1 Concurrency

Figure 4.14 shows that the DOD approach performs better than the other two approaches when the concurrency level (t=500) is set high. In the case of DP, the rise in concurrency level results in the increase in the load on the broker and network, because all n search-servers send $top - k$ results to the broker, which are accumulated and processed by the broker before sending those results to the user. On the other hand, for DOD index partitioning, load on the broker and network is low as a result of two factors. First, as only the head-search-server of each group (sub-cluster) sends results to the broker, it needs to compare only $g * top - k$ results where g is the number of sub-clusters. This results in a significant decrease in the amount of data to be transferred to the broker and processed by the broker. Second, the effect of overhead due to thread management issues is lower than for DP due to the distribution of computational load between the broker and

head-search-server(s).

Figure 4.14 shows that the DOT approach performs worst because the head-search-server(s) becomes a bottleneck as it is essentially uni-processing all the queries; although the search-servers in the group are serving postinglists, their contribution is insignificant because they are largely idle. Also, when the number of search-servers in the sub-cluster increases, term distribution gets skewed, resulting in an increase in the network traffic as more postinglists are transferred to the head-search-server.

4.10.2 Scalability

In terms of scalability, from Figure 4.15(a,b,c,d) it is clear that irrespective of concurrency level, both DP and DOD show a rise in the speedup ratio and thus are scalable. Further, the DOD partitioning approach performers better than DP for large data collections (DC/01, 12-search-servers) as shown in Figure 4.15(b,c,d).

Further, the DOT approach is not at all scalable as the speedup ratio drops significantly (halves) when the volume of data and number of search-servers grows in proportion as a result of two factors. First, as the number of search-servers in the sub-cluster increases, term index gets more skewed. This results in an increase in the number of transfers of postinglists over the network for query processing. Second, the head-search-server becomes a bottleneck as it is essentially uni-processing all the queries.

4.10.3 Resource Utilization

The cost due to communication, load on the broker, and processing at the head-search-server increase with the increase in the number of search-servers. Therefore, as seen from Figure 4.16, it is clear that all the approaches utilize resources more efficiently when data is stored on fewer search-servers. Data assignment to each server decreases with the rise in the number of servers, resulting into smaller index and faster query processing. Faster query processing and higher concurrency leads to thread management issues which results in inefficient utilization of resources.

From Figure 4.16(b) it is clear that the DOD approach utilizes resources more efficiently compared to the other approaches as a result of three factors. First, in the case of DOD, cost due to communication is low when compared to the baseline DP approach, because the addition of a layer between the broker and search-server doubles the bandwidth, however the amount of data required to transfer is still the same. Second, each search-server is entitled to generate only x% of the results (x=95% of top-k). Thus, the processing cost on a search-server is low as compared to DP. Third, the amount of results processed by

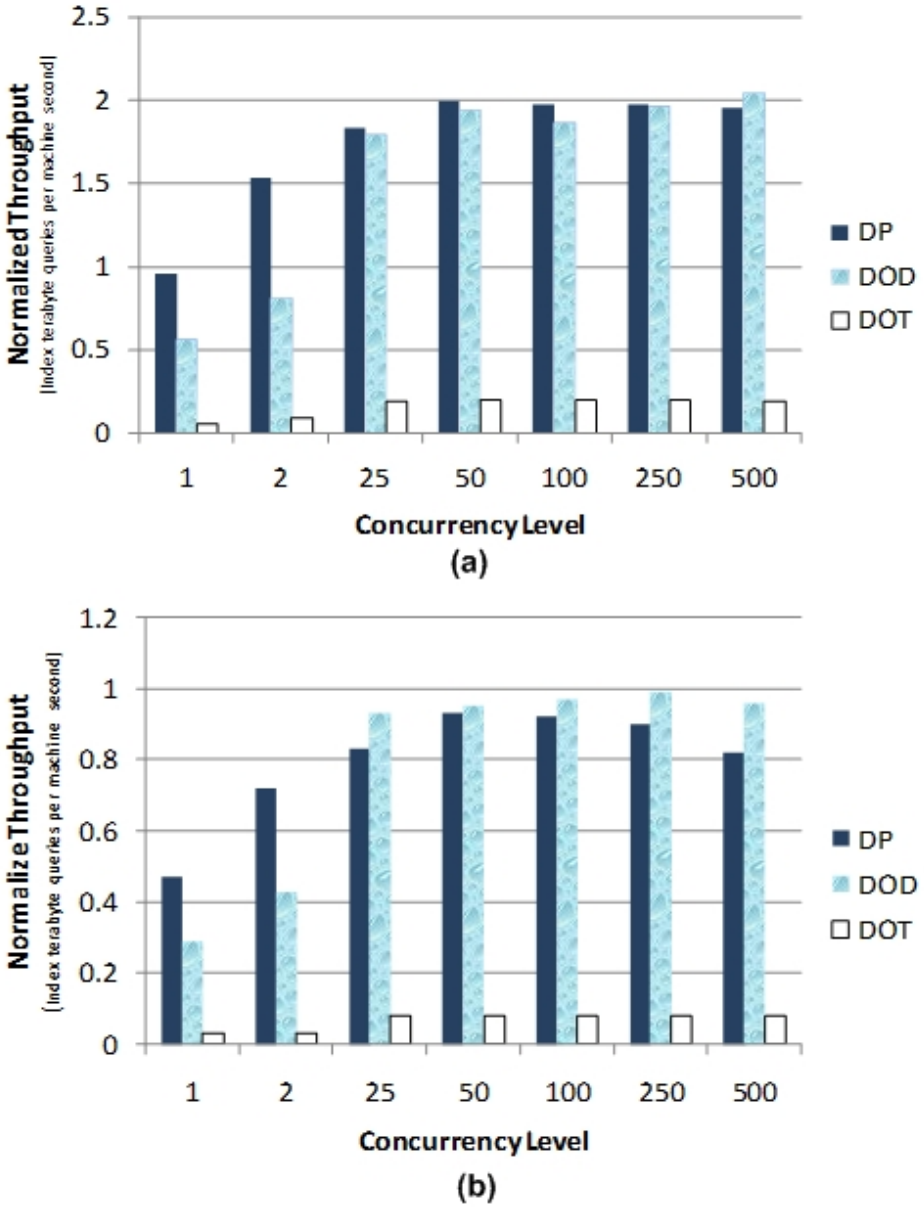

Figure 4.14: Effect of Concurrency on DP, DOD, and DOT. Experiment carried out by varying concurrency t from 1 to 500 and fixed data collection DC/01. (a) 6 search servers. (b) 12 search servers.

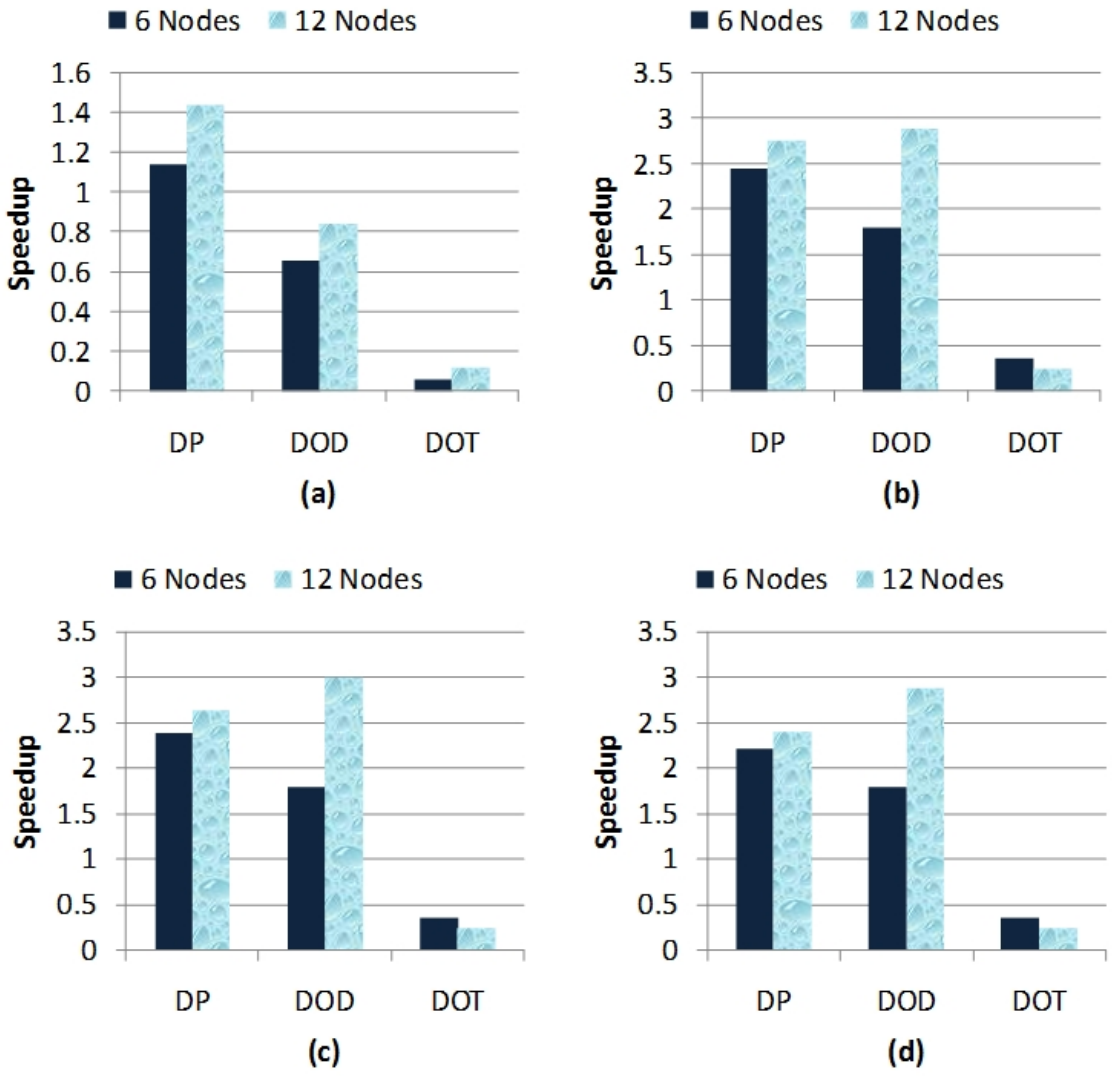

Figure 4.15: Comparison in terms of scalability. Set of experiments carried out by varying the number of search-servers for each data collection from (DC/02,6) to (DC/01,12). (a) sequential query processing (t=1). (b) t=50. (c) t=250. (d) t=500.

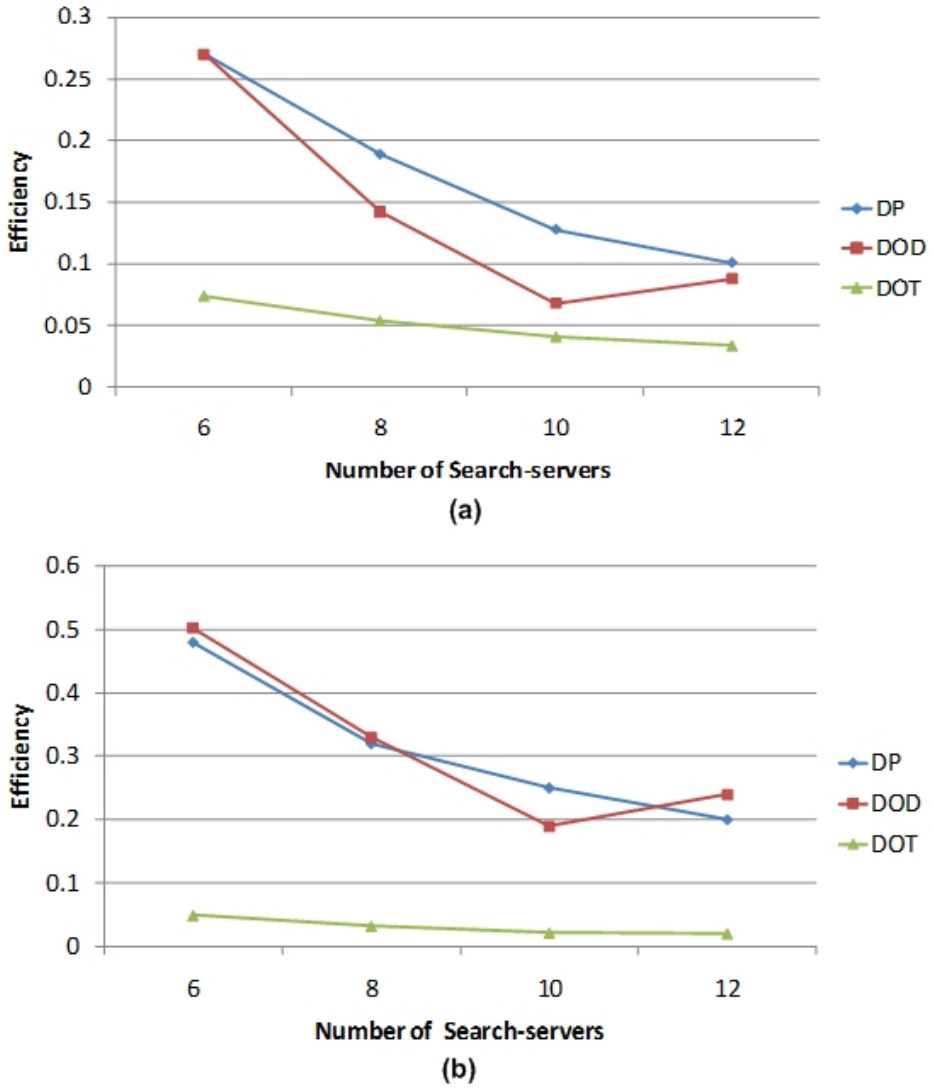

Figure 4.16: Comparison in terms of resource utilization. (a) by partitioning DC/02 data collection among search-servers. (b) by partitioning DC/01 data collection among search-servers.

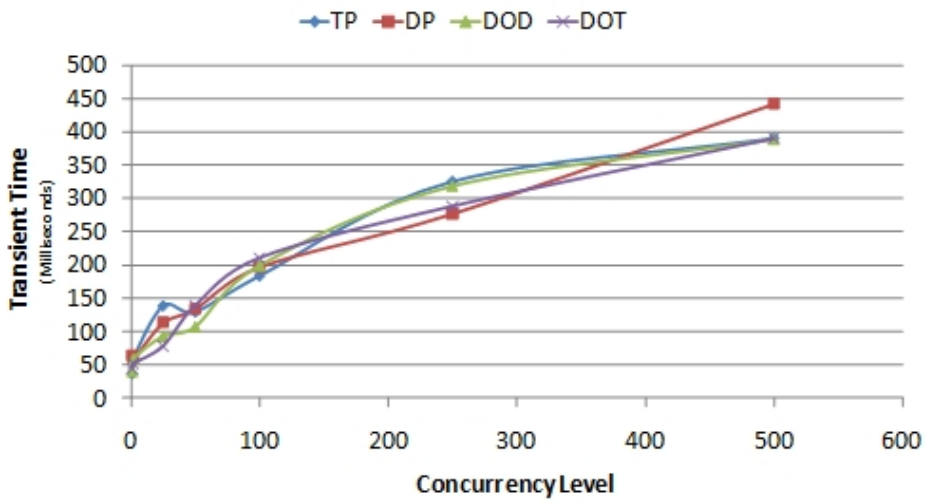

Figure 4.17: Effect of concurrency on steady state.

the broker is less as the computational cost is shared between the broker and head-search-server(s). On the other hand, DOT performs worst as a result of load imbalance.

4.10.4 Availability

IR systems build on top of distributed environment are vulnerable to the non-availability of the search-servers. Except for the traditional TP approach, all index partitioning strategy supports availability with loss of some effectiveness. However, ext1ending the index to be fault-tolerant is another crucial improvement which can be achieved by replicating index.

4.11 Other Comparative Experiments

4.11.1 Steady State

From the Figure 4.17, it is clear that the time require to reach steady state increases with the increase in the computing load and is approximately same for all index partitioning technique. However, IR system base on the memory index takes fewer time to reach steady state.

Table 4.10: Effect of number of results generated per search-server on Throughput. Concurrency level t= 500, n=12 search-servers and DC/01 data collection. Unit:- queries per second.

Results	DP	DOD	TP	DOT
10	494.6	578.2	3.9	48.1
50	462.0	562.4	3.9	46.7
100	449.6	551.1	3.9	45.1

4.11.2 Design Knob: Number of Results per Search-server

From the Table 4.10, we observer that the throughput for DP, DOD, and DOT decreases with the increase in the number of results generated by each search-server. This indicates that load on the server-server, broker, and network increases with the increase in the number of results. Same is not true for TP because amount of data required to be transferred to the broker and process by the broker is still the same. In the case of TP, broker has to sort entire list of relevant documents by score before sending it to the user. Thus forwarding top 10 results or top 100 results to the user has a negligible effect on the broker.

Chapter 5

Conclusion and Future Works

In this thesis, we designed and implemented two high performance inverted index partitioning schemes for a distributed Web search engine, running on the top of a cluster of machines. We analyzed two existing inverted-index partitioning strategies to help in characterizing the behavior of an IR system based on those approaches. This knowledge is taken into account while developing new efficient index partitioning strategies. Our *Document Over Term* approach retains the disk access benefit of the *term partitioning* (TP) approach; and sharing computational load, scalability, maintainability, and availability benefits of the *document partitioning* (DP) approach. We also introduced the *Document Over Document* strategy which retains all the benefits of the DP approach, but more effectively reduces the computational load and uses resources more efficiently. The detailed experimental comparison carried out between the existing and proposed approaches is described in Chapter 4 of this thesis.

In our experimental analysis, we make use of a large volume of data rather than extrapolating from small-scale experiments; use a realistic query set instead of synthetic query set; and record results when the system was in a steady state; to demonstrate the efficacy of the approach. We explore several design knobs, such as concurrent query request, cluster size, sub-cluster size, accumulator limit, and results sets; and studied how significant are the variation in the performance in term of throughput and resource utilization; and to what design knobs is the technique sensitive.

We compared all the methodologies experimentally, and proved that the DP and DOD performs significantly better than the others. Our experiments have demonstrated that at a high concurrency level, the DOD approach improves query throughput by 17% and efficiency(in terms of resource utilization) by 20% over a baseline DP approach.

Our experiments have demonstrated that the DOT approach offers much better performance than the traditional TP approach. The DOT approach improves query throughput

by 12 times over TP. However, when compared to the DOD and DP at a high concurrency level, the DOD approach improved query throughput by over 12 times and DP by over 10 times. On the other hand, DOT offers efficient utilization of search-servers and lowers the volume of disk access. These desirable attributes mean that further work should be done to address the load balancing problems by either distribution lists smartly or via selective lists replication guided by query log analysis.

Surprisingly, from experiments we discover that the IR system based on an in memory index takes very little time to warm-up. This indicates that the time taken by a system to reach its full potential after a failure is very low (few hundred milliseconds). Thus, recovery to full potential is quick.

We strongly believe that with some improvement in the area of data collection partitioning, the proposed architecture (DOD and DOT) has the potential to utilize resource more effectively and efficiently by reducing the computing cost of solving queries.

There is a lot of scope for future work in this area. Index partitioning using DOT or DOD seems to be a promising, which motivates further research. We sketch some possible research directions focusing on increasing the effectiveness of these methods in a test setting of larger scale:

- Considering the size of Web, GOV2 is not a large data collection. Further experiments on a larger data set and bigger cluster will be performed as a part of ongoing investigations along with the comparison with other hybrid approaches.

- Optimization of the index structure of the DOT strategy to avoid real time calculation of partial score can enhance the performance.

- Effectiveness of ranking algorithm is unclear for Nutch. Examining tradeoff between effectiveness and efficiency will be address in the future works.

- The current work provides guarantees on the performance in terms of efficiency. Work on guaranteeing performance in terms of other metrics like effectiveness, using a similar index partitioning approach would be interesting and challenging as those metrics may depend on other system parameters. Designing a strategy to improve the effectiveness by sharing information among the sub-clusters without affecting the performance might be useful.

- Caching is aimed exclusively at reducing the computing load of the system and thus improving the overall system efficiency. Running a distributed IR system on top of the DOD or DOT approach results in at least two levels of caches, one in the broker and one in the head-search-server(s). The challenges of making effective use of caches on the broker (or any frontend server) by deciding which query result sets to store

are well studied. Designing a caching policy which maintains exclusivity among the contents of the caches in multi-layer architecture like DOT and DOD will enhance the performance and will lead to efficient resource utilization.

- Collection selection strategy minimizes the number of search-servers contacted for query processing. Preparing the routing list of search-servers containing the relevant documents incurs extra load. This cost increases with the increase in the number of servers in a cluster. Thus, designing a collection selection strategy for DOD, which deals with finding sub-cluster(s) containing relevant documents instead of search-servers, will be computationally less expensive and at the same time use resources more efficiently.

- Effectiveness and efficiency of a collection selection strategy directly depends on how documents are partitioned among the servers. If documents are randomly partitioned, the collection selection strategy will end up sending a query to almost all servers. In [30], authors propose algorithms, which are explained briefly in Section 2.4.1, to perform data collection partitioning and collection selection using query log analysis which is based on a document classification techniques. Work on guaranteeing performance in terms of effectiveness and efficiency for the DOD approach, using data collection partitioning technique along with the collection selection strategy proposed in [30] would be interesting and challenging.

- The main shortcomings of the DOT architecture are hotspots and load imbalance. It is possible to improve the balance of the load by exploiting the information on the frequency and co-occurrence of the terms in the query logs for distributing terms among the search-servers and for postinglist replication.

- In the case of DOD, instead of using heuristics for specifying the number of results to be generated by each search-server, a probabilistic approach could be adopted to guarantee that each head-search-server sees at least the top-k results.

References

[1] Claudine Santos Badue, Ricardo A. Baeza-Yates, Berthier A. Ribeiro-Neto, Artur Ziviani, and Nivio Ziviani. Analyzing imbalance among homogeneous index servers in a web search system. *Inf. Process. Manage.*, 43(3):592–608, 2007. 15

[2] Claudine Santos Badue, Ricardo A. Baeza-Yates, Berthier A. Ribeiro-Neto, and Nivio Ziviani. Distributed query processing using partitioned inverted files. In *SPIRE*, pages 10–20, 2001. 14

[3] Ricardo A. Baeza-Yates, Carlos Castillo, Flavio Junqueira, Vassilis Plachouras, and Fabrizio Silvestri. Challenges on distributed web retrieval. In *ICDE*, pages 6–20, 2007. 14

[4] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999. 2, 3, 4, 10

[5] Deepavali Bhagwat, Kave Eshghi, and Pankaj Mehra. Content-based document routing and index partitioning for scalable similarity-based searches in a large corpus. In *KDD*, pages 105–112, 2007. 15, 16

[6] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003. 16

[7] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998. 10

[8] Brendon Cahoon, Kathryn S. McKinley, and Zhihong Lu. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *ACM Transactions on Information Systems*, 18(1):1–43, 2000. 10

[9] James P. Callan, Zhihong Lu, and W. Bruce Croft. Searching distributed collections with inference networks. In *SIGIR*, pages 21–28, 1995. 16

[10] Berkant Barla Cambazoglu, Aytul Catal, and Cevdet Aykanat. Effect of inverted index partitioning schemes on performance of query processing in parallel text retrieval systems. In *ISCIS*, pages 717–725, 2006. 6, 13

[11] Pei Cao and Zhe Wang. Efficient top-k query calculation in distributed networks. In *PODC: 23th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 206–215, 2004. 28

[12] Nick Craswell. *GOV2 Test Collection*, 2004. `http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm`. 31

[13] Owen de Kretser, Alistair Moffat, Tim Shimmin, and Justin Zobel. Methodologies for distributed information retrieval. In *ICDCS*, pages 66–73, 1998. 10

[14] Susan T. Dumais. Latent semantic indexing (lsi) and trec-2. In *TREC*, pages 105–116, 1993. 16, 17

[15] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *JCSS: Journal of Computer and System Sciences*, 66(4):614–656, 2003. 28

[16] William B. Frakes. Introduction to information storage and retrieval systems. In *Information Retrieval: Data Structures & Algorithms*, pages 1–12. 1992. 2, 3, 4, 6, 33

[17] Luis Gravano, Héctor GarcMolina, and Anthony Tomasic. The effectiveness of GIOSS for the text database discovery problem. 1994. 16

[18] Donna Harman, Wayne McCoy, Robert Toense, and Gerald Candela. Prototyping a distributed information retrieval system that uses statistical ranking. *Inf. Process. Manage.*, 27(5):449–460, 1991. 10

[19] Thomas Hofmann. Probabilistic latent semantic indexing. In *SIGIR*, pages 50–57, 1999. 16

[20] R. Jain. The art of computer systems performance analysis. *John Wiley & Sons, Inc.*, 1991. 34

[21] Byeong-Soo Jeong and Edward Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):142–153, 1995. 14

[22] Andrew Kane. *Simulation of Distributed Search Engines: Comparing Term, Document and Hybrid Distribution*, 2009. `http://www.cs.uwaterloo.ca/~arkane/main.shtml`. 18, 23

[23] John D. King and Yuefeng Li. Web based collection selection using singular value decomposition. In *Web Intelligence*, pages 104–110, 2003. 16

[24] Steve Lawrence and C. Lee Giles. Accessibility of information on the web. *Intelligence*, 11(1):32–39, 2000. 1

[25] Nicholas Lester, Alistair Moffat, William Webber, and Justin Zobel. Space-limited ranked query evaluation using adaptive pruning. In *WISE*, pages 470–477, 2005. 32

[26] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. Mining query logs to optimize index partitioning in parallel web search engines. In *Infoscale*, page 43, 2007. 16, 17

[27] Chris Mattmann. *Nutch*. `http://wiki.apache.org/nutch/`. 30

[28] T. Mayer. *Our blog is growing up - and so has our index*, 2005. `http://www.ysearchblog.com/archives/000172.html`. 1

[29] Alistair Moffat, William Webber, Justin Zobel, and Ricardo A. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Information Retrieval*, 10(3):205–231, 2007. 10, 12, 13, 17, 31, 32

[30] Hiren Patel. Document partitioning and collection selection in distributed search engine. Technical report, 2009. 16, 17, 57

[31] Ivana Podnar, Martin Rajman, Toan Luu, Fabius Klemm, and Karl Aberer. Scalable peer-to-peer web retrieval with highly discriminative keys. In *ICDE*, pages 1096–1105, 2007. 15

[32] Diego Puppin and Fabrizio Silvestri. The query-vector document model. In *CIKM*, pages 880–881, 2006. 16, 17

[33] Diego Puppin, Fabrizio Silvestri, and Domenico Laforenza. Query-driven document partitioning and collection selection. In *Infoscale*, page 34, 2006. 15, 16

[34] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001. 15

[35] Berthier A. Ribeiro-Neto, Edleno Silva de Moura, Marden S. Neubert, and Nivio Ziviani. Efficient distributed algorithms to build inverted files. In *SIGIR*, pages 105–112, 1999. 13

[36] Gordon V. Cormack Stefan Bttcher, Charles L. A. Clarke. *Information Reterival: Implementing and Evaluating Search Engines*. MIT Press, 2010. 1, 3, 5, 6, 12, 23, 28

[37] Trevor Strohman and W. Bruce Croft. Efficient document retrieval in main memory. In *SIGIR*, pages 175–182, 2007. 34

[38] Torsten Suel, Chandan Mathur, Jo wen Wu, Jiangong Zhang, Alex Delis, Mehdi Kharrazi, Xiaohui Long, and Kulesh Shanmugasundaram. Odissea: A peer-to-peer architecture for scalable web search and information retrieval. In *WebDB*, pages 67–72, 2003. 15

[39] Chunqiang Tang, Zhichen Xu, and Sandhya Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *SIGCOMM*, pages 175–186, 2003. 15

[40] Trevor Strohman W. Bruce Croft, Donald Metzler. *Search Engines: Information Retrieval in Practice*. Addison Wesley, 2010. 1, 2, 3, 4, 6, 33

[41] Ming Luo Wensi xi, Ohm Sornil and Edward Fox. Hybrid partition inverted files for large-scale digital libraries. In *Beijing Library Press*, pages 404–418, 1993. x, 18, 19

[42] Yiming Zhang, Dongsheng Li, Lei Chen 0002, and Xicheng Lu. Collaborative search in large-scale unstructured peer-to-peer networks. In *ICPP*, page 7, 2007. 15

[43] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006. 4