SEMANTIC ROUTED NETWORK FOR DISTRIBUTED SEARCH ENGINES

A Dissertation

by

AMITAVA BISWAS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2010

Major Subject: Computer Engineering

SEMANTIC ROUTED NETWORK FOR DISTRIBUTED SEARCH ENGINES

A Dissertation

by

AMITAVA BISWAS

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Rabi N. Mahapatra
Committee Members,	Eun Jung Kim
	Radu Stoleru
	Behbood Zoghi
Head of Department,	Valerie E. Taylor

May 2010

Major Subject: Computer Engineering

ABSTRACT

Semantic Routed Network for Distributed Search Engines. (May 2010)

Amitava Biswas, B.Tech.(Hons), Indian Institute of Technology;
M.S., Concordia University;
M.B.A., Indian Institute of Management
Chair of Advisory Committee: Dr. Rabi N. Mahapatra

Searching for textual information has become an important activity on the web. To satisfy the rising demand and user expectations, search systems should be fast, scalable and deliver relevant results. To decide which objects should be retrieved, search systems should compare holistic meanings of queries and text document objects, as perceived by humans. Existing techniques do not enable correct comparison of composite holistic meanings like: “evidences on role of DR2 gene in development of diabetes in Caucasian population”, which is composed of multiple elementary meanings: “evidence”, “DR2 gene”, etc. Thus these techniques can not discern objects that have a common set of keywords but convey different meanings. Hence we need new methods to compare composite meanings for superior search quality.

In distributed search engines, for scalability, speed and efficiency, index entries should be systematically distributed across multiple index-server nodes based on the meaning of the objects. Furthermore, queries should be selectively sent to those index nodes which have relevant entries. This requires an overlay Semantic Routed Network which will route messages, based on meaning. This network will consist of fast response networking appliances called semantic routers. These appliances need to: (a) carry out sophisticated meaning comparison computations at high speed;

and (b) have the right kind of behavior to automatically organize an optimal index system. This dissertation presents the following artifacts that enable the above requirements:

(1) An algebraic theory, a design of a data structure and related techniques to efficiently compare composite meanings.

(2) Algorithms and accelerator architectures for high speed meaning comparisons inside semantic routers and index-server nodes.

(3) An overlay network to deliver search queries to the index nodes based on meanings.

(4) Algorithms to construct a self-organizing, distributed meaning based index system.

The proposed techniques can compare composite meanings ~10^5 times faster than an equivalent software code and existing hardware designs. Whereas, the proposed index organization approach can lead to 33% savings in number of servers and power consumption in a model search engine having 700,000 servers. Therefore, using all these techniques, it is possible to design a Semantic Routed Network which has a potential to improve search results and response time, while saving resources.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF FIGURES

FIGURE Page

FIGURE Page

FIGURE Page

FIGURE Page

FIGURE Page

LIST OF TABLES

CHAPTER I

INTRODUCTION

1.1 Motivation

1.1.1 Demand for meaning based search

Searching for unstructured textual information has become an important activity on the internet. Currently the demand for "web search" service is 13 billion search queries per month and growing 38% annually [1]. With this rising demand, user's expectations are also increasing. Users prefer more sophisticated **meaning based (semantic) search** capabilities that go beyond keyword matching based searches [2]. For example, they expect that a search for "healthy lifestyle" (the query) should retrieve documents/web pages related to "nutrition", "wellness", "diet", "exercise", "fitness" etc., even though the user's query and the returned documents/pages may not share any common keywords. This kind of search involves **comparing meaning** of a search query against meanings of all available objects (e.g. text document, web pages, etc.) to identify which objects are similar to the query in terms of meaning (Fig. 1.1). The objects, whose meanings are similar, are retrieved and presented to the user as search results.

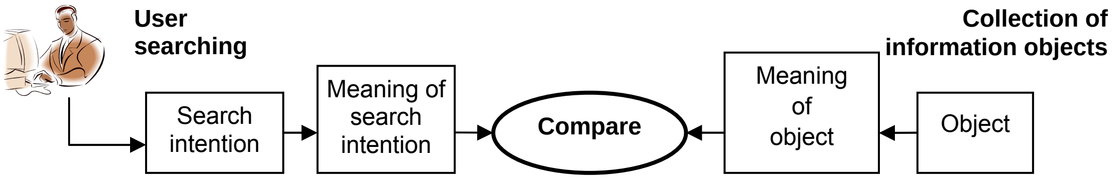

User searching
Search intention
Meaning of search intention
Compare
Meaning of object
Object
Collection of information objects

Fig. 1.1 The meaning comparison processes required for meaning-based search

This dissertation follows the style of *IEEE Transactions on Parallel and Distributed Systems*.

In this process, the quality of search results will depend on how well computers can represent and compare meanings as perceived by human mind. Existing web search engines have adopted some forms of meaning comparisons but those are yet to deliver high quality meaning based (semantic) search results in all situations. Hence there are opportunities to make improvements.

1.1.2 Distributed search engines

Both kinds of searching, simple keyword based and semantic searching, involves **intensive computations** and require large scale computing infrastructure. Hence, to match the growing demand of search service, internet search engines are striving to scale up their infrastructure. The number of webpages/documents indexed by a typical internet search engine is in order of tens of billions [3]. These search infrastructures also serve several billions of queries per month. For example, Google serves 9 billon queries per month or 3500 per second [1]. To cater to that kind of service demand, large search engines are integrated as distributed systems in data centers [4]. The distributed design imparts the needed scalability and speed. For example, Google take a short time of 200 milliseconds on average, to deliver search results [5] and have over half a million servers/computers (estimated) in multiple data centers [6]. As a result these data centers consume very large amount of power (~mega watts).

For efficiency and scalability, large search engines use an **index system**. By using index system one can avoid comparing the search query with each and every available object. The index system drastically reduces the number of comparisons required and yet manages to identify the best matching objects. This index is the heart of the search engine. The index map queries (search intentions) to objects. Thus the index system is a collection of map entry pairs of search keywords and object locations. For scalability, this index collection is usually broken up into multiple pieces by **randomly distributing** index entries over multiple pools of servers. Here each pool hosts a portion of the index called *index shard*, as shown in Fig. 1.2. More details about this design are available in [4]. Here, each index shard resides in a server pool which can be viewed as a small functioning search engine or index system.

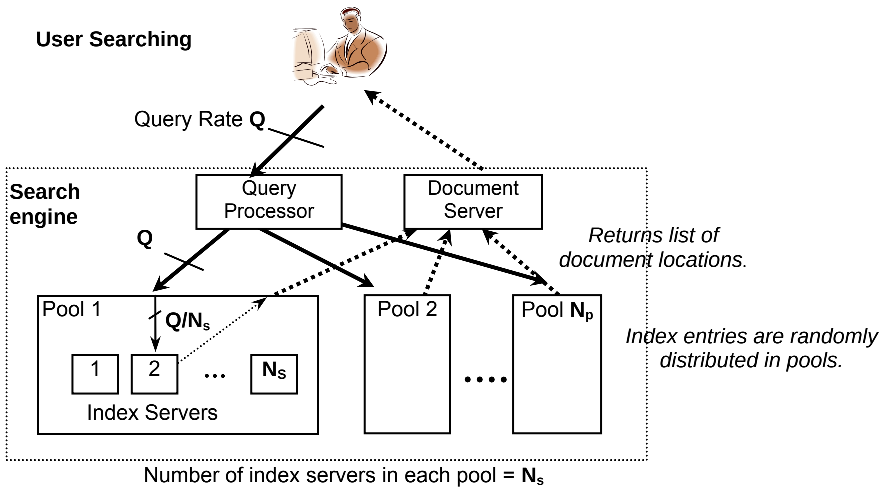

User Searching
Query Rate Q
Search engine
Query Processor
Document Server
Q
Returns list of document locations.
Pool 1
Q/N_s
1
2
...
N_S
Index Servers
Pool 2
....
Pool N_p
Index entries are randomly distributed in pools.
Number of index servers in each pool = N_s

Fig. 1.2 Index distribution and query delivery in a typical distributed search engine

In this system, the user provides search keywords to the query processor. The search keywords serve as a meaning representation of user's search intention. The query processor creates an internal representation of the search intention. This representation may be a term vector, latent semantic vector or a set of search keywords [7]-[9]. This search intention is broadcasted to all the index shards or pools, as shown by heavy arrows in Fig. 1.2. The index pools which have matching entries return locations of objects to the document server (shown by broken arrows in Fig. 1.2). The document locations are expressed in terms of document identifiers (ids) and each document id is mapped to a unique web URL [4]. The document server is the map between the document ids and their web URLs. On getting the document ids, the document server returns all the relevant URLs to the user as search results (shown by a single broken arrow at the top in Fig. 1.2).

Each individual index server pool contains a large number of index servers, all having the same index map (i.e. replica of the same index server). These multiple index servers together enable load sharing for scalability. The incoming search query traffic is uniformly distributed across all the servers located within a pool. Hence the throughput of the entire pool is given by the individual throughput capacity of each server multiplied by the number of servers in the pool. The number of index servers chosen to ensure that the total throughput of the pool matches the query traffic rate.

In this kind of distributed index system, one key problem is how to distribute the queries at high rate (3500 per second) to all the index pools. Broadcasting to all index pools, as explained earlier, is a simpler solution, but it is not the most efficient method due to the broadcast mode of operation (Fig. 1.2). This approach implies that the number of index servers in each pool have to be scaled up to handle the full query traffic rate. This results in a larger number of servers in each pool and higher power consumption. Last few years, power consumption has become a key concern in data center planning and management [10]. So there is a need to deploy power efficient index distribution and query forwarding techniques that would satisfy the throughput requirements.

1.1.3 Energy efficient distributed index and role of semantic router

Instead of query broadcasting, an energy efficient approach will be to **systematically distribute** index entries to the pools based on the meaning of the objects (as in Fig. 1.3), so that objects (documents) having similar content in terms of meaning, are indexed in the same index pool.

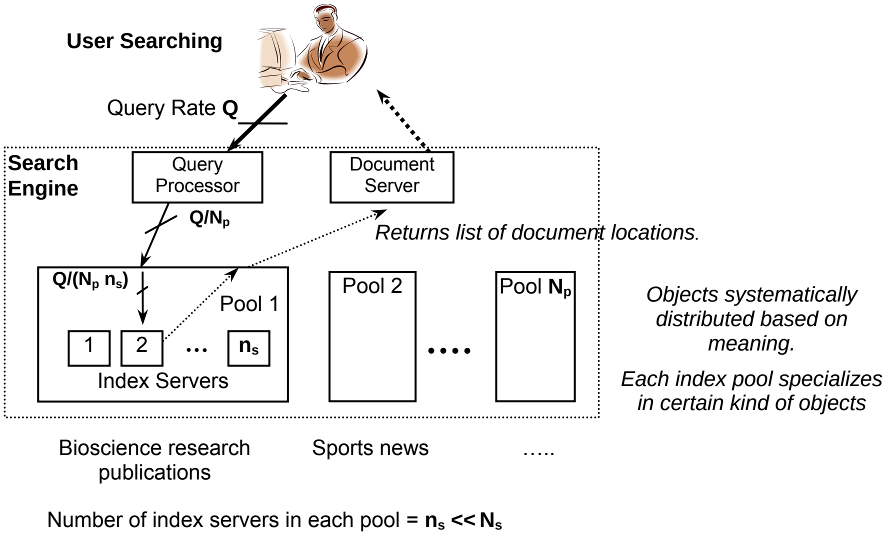

User Searching
Query Rate **Q**
Search Engine
Query Processor
Document Server
Q/N$_p$
Returns list of document locations.
Q/(N$_p$ n$_s$)
Pool 1
1
2
…
n$_s$
Index Servers
Pool 2
Pool **N$_p$**
….
Objects systematically distributed based on meaning.
Each index pool specializes in certain kind of objects
Bioscience research publications
Sports news
…..
Number of index servers in each pool = **n$_s$** << **N$_s$**

Fig. 1.3 Proposed index distribution and query delivery model

Here the index pools can be viewed as specializing in certain kind of objects. For example, index entries on webpages related to sports news will be in one pool, bioscience research publications, in another and so on. This naïve example is mentioned to convey the notion. In actual systems, the index pools may not have a simple topic as mentioned in this example. Rather the categorization topic of an index pool will be defined by a model object, and all other objects that are similar to this model object will be indexed in that pool.

In this approach, the query has to be send to only a single index pool (or few of them) which is (are) likely to have relevant objects, unlike the system in Fig. 1.2. Thus this method will obviate unnecessary query traffic to all pools thereby allowing index server pools to be smaller and have lower power consumption. In this example, the query traffic rate Q per second gets equally distributed. Thus the query rate encountered by each pool is Q/N_P, assuming all documents are equally likely to get queried and these are equally distributed across all pools. In that case, the number of servers needed in each pool can be n_S (Fig. 1.3), which is $1/N_P$th fraction of the number of servers N_S in the system shown in Fig. 1.2. When N_P is in order of 1000, this scheme can significantly reduce the number of index servers.

Materializing such efficient index distribution (as in Fig. 1.3) will necessitate two things:

(1) An *automatic mechanism to systematically distribute index entries* among the available index fragments (pools); and

(2) A *query delivery network* to deliver a query to a specific pool(s) based on the meaning of the query.

In fact both: object distribution and query delivery mechanism, can be built if an underlying meaning based message routing/forwarding network is available. A Semantic Routed Network (SRN) can be used to implement this mechanism, as shown in Fig. 1.4.

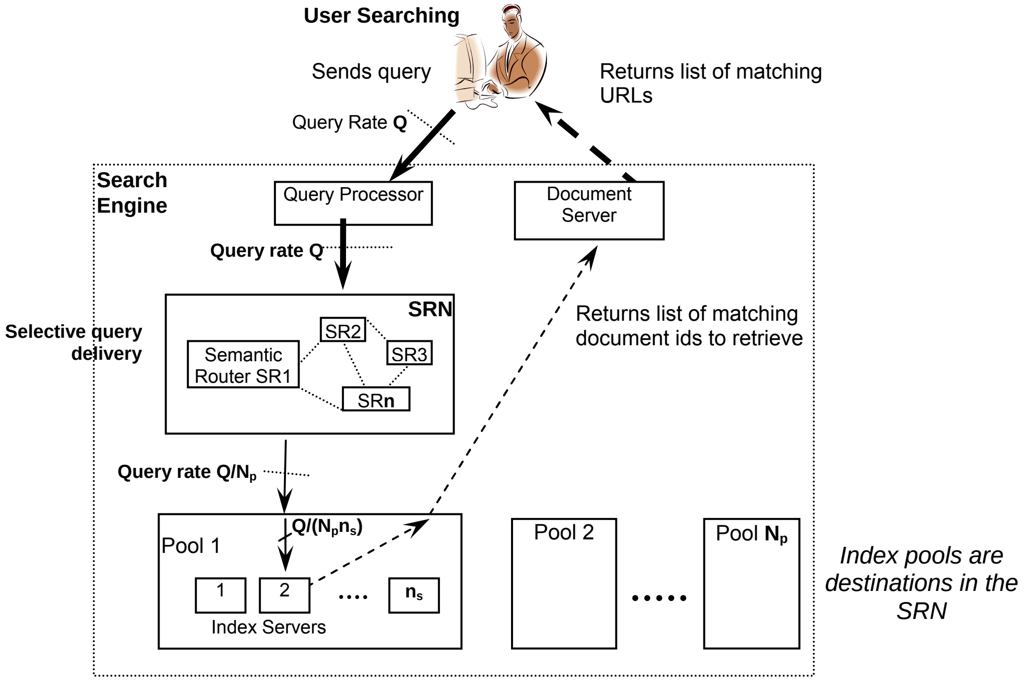


Fig. 1.4 Role of Semantic Routed Network in the proposed distributed search engine

In this system, the index pools are considered as the destination nodes from SRN's viewpoint. Within SRN, messages are routed, using a fast response application protocol level message forwarding appliance called *semantic routers*. These semantic router appliances are distinct from IP routers. These semantic routers will delivery queries and route messages between index pools to enable co-ordination between the pools and their reorganization. These semantic routers can be either placed within a single data center on top of a clustered system or on the internet connecting index servers across multiple data centers. When a message arrives, the semantic routers will compare the meaning contained in message against meanings of all the available destination's specializations, and then forward the message to the destination (index pool) whose meanings is most similar. Such semantic routers need a fast meaning comparison scheme to do meaning based destination lookup. The index pool specialization, meaning comparison and meaning based destination lookup mechanisms are analogous to IP addresses, IP longest prefix matching in the IP routing table lookup processes.

1.2 Requirements and challenges

1.2.1 Challenges in meaning similarity computation

1.2.1.1 Role and criticality of meaning comparison

Given the kind of meaning based message forwarding as explained in the previous section, the effectiveness of the proposed index organization scheme will depend on the sophistication of the meaning similarity comparison capability of the semantic routers. Effectiveness and efficiency of the meaning comparison method will in turn depend on the kind of meaning representation technique that is chosen. In this regard, the existing meaning representation and comparison techniques, as embodied in the web search engines and other search systems, are not very effective. This is evident from the fact that, the existing search engines do not yet deliver high quality semantic searching. Some examples are presented in subsequent sections to illustrate this problem.

1.2.1.2 The meaning composition problem

A major limitation of the existing meaning comparison methods is that these are largely based on meaning comparison of individual keywords, their synonyms, hyponyms, hypernyms and contextually similar or related words (as in latent semantic indexing [8]). Searching based on meanings of individual keywords, are effective when the search intentions are simple. To search with more complex search intentions, which are conveyed by longer query phrases or by couple of sentences, a more sophisticated meaning comparison technique is needed. This sophistication can be engendered, if, computers can represent and compare holistic meanings as perceived by human mind.

For example, a user searching for cause of diabetes among the Caucasian people, may use a search phrase: “evidences on role of DR2 gene in development of diabetes in Caucasian population”. This phrase conveys a composite meaning. This meaning is composed of elementary meanings that are conveyed by: “evidence”, “DR2 gene”, etc. In such situations, a single “bag of words” aggregation of the individual keywords [11] may not always convey the composite meaning of the entire object, as assumed in popular meaning representation models (set or vector models [9]). As instances of holistic *composite meanings* arise in description of objects and search intentions, therefore we need techniques to **represent and compare composite meanings**. This need for capturing the **composition aspect** is illustrated by an exaggerated example [12], presented below. Here the same set of keywords, convey two very different meanings.

Object 1: It was not the sales manager who hit the bottle that day, but the office worker with the serious drinking problem.

Object 2: That day the office manager, who was drinking, hit the problem sales worker with a bottle, but it was not serious.

Vector or set based information retrieval models, which suffer from the "bag of word" problem, will report the two texts presented below, to be similar in meaning. If meaning based search systems employ adequate mechanisms to represent and compare composite meanings, for example, a technique which could have distinguished the two aforementioned texts as having different meaning, then that would improve the specificity of holistic meaning representation and improve the quality of search results. In other words, adoption of high quality meaning representation and comparison technique will improve relevance of search results.

1.2.1.3 Need for generative composition, meaning interchangeability and speed

In human mind, the composition of meaning is often generative. This means elementary meanings are composed to generate a composite meaning, then several of such composite meanings may be further composed together to generate the next higher level composite meaning, and so on, till a multiple hierarchical level composition is generated to get the final composite meaning. Interchangeability of similar meanings is common in natural language expressions, communications and interpretations. Thus along with this generative composition aspect, interchangeability of synonyms also has to be incorporated. Hence we need a speedy, unified, generative mechanism that can simultaneously:

(1) compose elementary meanings to represent composite meaning; and

(2) recognize the semantic (meaning) relationships between dissimilar text strings,

to yield consistent results unlike Table 1.1 [13] & Table 1.2. Here these two tables illustrate the weakness of the state-of-art search system due to poor meaning representation quality. We are interested in meaning representation and comparison techniques for search systems, hence we are using examples from state-of-the art search systems to demonstrate the weaknesses of the deployed techniques. These weaknesses get exposed even when small composite meanings are used as search intentions, therefore problems will be more acute with larger composite meanings. In both these cases, as shown in these tables, composition and interchangeability of similar meanings appear to be are un-coordinated.

Table 1.1 Limitations of the scientific publications search at Pubmed [14]

Keyword used in "All Database" searching	Objects returned	Comments
"diabetes" AND "PPAR"	1979	Indicates that objects are available
"auto immune disease" AND "PPAR"	0	This result should be a superset of the result above, so the number of results should not be less than the number in 1st row.
"diabetes" AND "nuclear receptor"	406	This number of results should not be less than in 1st row

Note: "PPAR" is a "nuclear receptor", "diabetes" is an "auto immune disease". Observed on Jul, 2008.

In case of Table 1.1, the user wanted to search for a scientific publication on role of "PPAR" which is a type of "nuclear receptors" that plays a role in causing "autoimmune diseases" and its subclass "diabetes" disease. In this experiment, three searches were carried out which are reported as three rows in the table. The keywords that were used are reported in the first column and the number of items that are presented as search results are reported in the second column of the table. The results with different search keyword combinations are inconsistent. This indicates that meaning composition, as implemented by AND operator (e.g. "diabetes" AND "nuclear receptor") is not working with meaning interchangeability (e.g. when "PPAR" is used instead of "nuclear receptor"). Thus Table 1.1 illustrates the limitations of a set based information retrieval model.

On the other hand, Table 1.2 shows limitations of a popular web search engine which appears to be using latent semantic indexing which is a vector model.

Table 1.2 Limitations of a popular internet search engine

Keywords used	Top 20 results	Comments
rodent supplier	Some are relevant	Presence of some results indicates that objects are available.
mouse supplier	Irrelevant	Returns web pages on computer mouse suppliers.
supplier animal mouse medical experimentation	Irrelevant	Fails to return relevant results even though the term "mouse" is disambiguated by adding "animal".

Note: Observed on Apr 24, 2009.

The 3rd row of Table 1.2 illustrates the need of hierarchical/generative composition. Usage of the word "animal" along with "mouse" to generate the composition "animal mouse", should have enabled disambiguation of the term "mouse", so that it is not to be confused with computer device mouse. Next, when a higher level composition between "animal mouse" and "supplier" is used as the search query, the search engine should have yielded relevant search results. However this did not happen even though required objects were available (2nd column of the 1st row shows that objects were present). Co-coordinating all these features together is computation intensive and not easy. These examples indicate that satisfying all three requirements: generative composition, interchangeability of similar meanings, and speed, together in a meaning representation is not trivial. In addition, we need to also think how to best apply the meaning representation and comparison techniques in a search system, once they are designed. This draws our attention to the second group of challenges.

1.2.2 Challenges in designing meaning based message delivery network

1.2.2.1 Need for novel network organization

Traditional routable networks such as IP networks are organized as hierarchy of sub-networks, where addresses are numeric variables. At every level, the sub-networks are assigned a range of numeric addresses, based on which the messages are routed up and down the hierarchical *k*-ary tree (a tree where each node has *k* children instead of only two in a binary tree node). In this case the tree traversal works because the numerical address keys support three required logical relationship operators: less than, greater than or equal to, between them. These logical operations are implicitly carried out during longest prefix match during IP routing table lookup. However none of these three logical operators can be defined for meanings, because one can not claim that one meaning is greater or less than the other. Thus it is not possible to construct a hierarchical network topologies (or *k*-ary trees) for meaning based routing.

Meaning representations support only a single non-logical operator which is meaning comparison operation that yields a meaning similarity value. This value is non-binary in nature because the similarity value is a continuous valued variable. For example, similarity values between 0 and 1, define different levels of similarity from absolutely no similarity to exact or 100% similarity. Currently vector models provide such similarity values [9].

In IP network, the numeric address space is partitioned into ranges, specific ranges are allocated to sub-networks and the hierarchical IP address space is built on this premise of range partitioning. Conceptually, we can conceive an analogous meaning (semantic) space where each point in that space represents a meaning and the distance between two points represents the extent of similarity between the meanings represented by those two points. But the meaning space can not be partitioned in a manner in which we can partition the IP address space. This is because meaning is a non-numeric entity. This also pose a severe challenge in terms of how to organize a routable network where messages can be routed based on meaning or how destination nodes can be addressed based on their meaning (i.e. the meaning of their content). This means that the usual networking technique can not be used to implement a meaning based message delivery in the Semantic Routed Network. A new kind of topology and routing logic has to be innovated and/or adapted. In addition, as the meaning representation method is yet to be defined, so this network design problem is not trivial.

1.2.2.2 Need for speed and efficiency

Existing web search engines deliver search results very quickly (within ~200 millisecs) [5], hence they have set the standards of user expectations. Therefore the Semantic Routed Network has to deliver queries within a small fraction of that 200 millisec window, to be acceptable. Thus the semantic routers have to be fast and resource efficient. It is not preferable to deploy large clustered system to implement each semantic router to gain speed. Because that kind of approach will take away the hardware savings gained by the proposed index organization scheme. Therefore, we have to innovate technologies, which will enable us to design a semantic router that can readily fit inside a small enclosure or a data center rack.

1.3 Scope of this dissertation and addressed challenges

To materialize this meaning based index and query delivery network (Semantic Routed Network), the following problem areas need to be addressed:

1. *Design of meaning representation data structure and comparison technique*: A composite meaning has to be represented as a computable data structure called *semantic descriptor* or *semantic key*. This data structure will represent meanings of: user’s search intentions,

objects and index pool specializations. This data structure will be used as the message address (analogous to IP address in IP routing layer) in the proposed query delivery overlay network. The key challenge, that will be addressed here, is how to enable composition of elementary meanings to represent composite/complex meanings and interchangeability of similar meanings, while allowing fast meaning comparison. This composition aspect is important to enable semantic routers which will enable meaning based clustering of index entries and query delivery to appropriate index pools. The quality of these operations will be reflected in the search performance and relevance of search results.

2. *Design of algorithms and a high level architecture for the accelerator hardware to speedup the proposed meaning comparisons* inside computers. This hardware will sit inside semantic routers. This hardware will compare semantic descriptors that represent users' search intentions against those descriptors that represent index pools' specializations during for query routing operations. Additionally, this hardware may be incorporated in the index servers to take part in the object retrieval decisions and result relevance ranking operations. The challenge that is being addressed here is how to enable extremely fast meaning comparison computations to enable low search response time.

3. *Design of an overlay networking scheme which will deliver messages based on their meaning.* Here networking protocol and the proposed networking stack will be presented which will make use of semantic routers.

4. *Designing algorithms to enable construction of semantic index system* using an underlying meaning based message forwarding network. Objects that are related or have similar meanings will be automatically grouped/reorganized together inside a single index pool in this index system. This index infrastructure will be a network of individual computers in a

data center or even small specialized internet search engines. Here the idea is to design a co-ordination scheme that will integrate services of multiple indexes, index fragments and/or several small search engines.

In this dissertation, we would present the aforementioned techniques to substantiate that it is possible to design a meaning based message routing network. Then we will show that, a systematically organized distributed index system which uses a Semantic Routed Network, requires smaller number of index servers and consumes less power to operate.

1.4 Organization of this dissertation

Chapter II analyses the existing state of art in: meaning representation, meaning comparison, meaning based indexing, index scaling strategies, distributed search technologies and network science domain. These are relevant to distributed index systems and message forwarding network design.

Chapter III discusses the proposed meaning representation, meaning comparison model, the necessary algorithms and their rationale. Chapter IV describes the information processing architecture for a meaning comparator hardware which forms the basis for actual hardware design of the semantic router core.

Chapter V identifies the requirements for the Semantic Routed Network (SRN) mechanism and presents its design. Whereas, Chapter VI presents necessary algorithms and optimization techniques that are needed to materialize a practical SRN.

Chapter VII presents the claims made in this dissertation, evaluation approach to substantiate the claims, necessary experiments, their rationale, results and discussions.

Finally, Chapter VIII concludes this dissertation by identifying the contributions of this dissertation and the open problems.

CHAPTER II

PROBLEM EXAMINATION AND ANALYSIS OF PRIOR ART

To design an effective meaning representation data structure and a comparison technique, we need to understand how human mind actually comprehends meanings. Similarly, to devise a network which can forward messages based on meaning, we need to know the necessary network science concepts and the pros and cons of various networks which had been employed in past for distributed searching. In this chapter we analyze these prior arts to get a better understanding of the problems and issues involved. We introduce the relevant terms and concepts used in this dissertation and present the relevant cognitive science and linguistic theories on how humans process meanings. We also describe existing meaning representation and comparison techniques, relevant designs of scalable index system and explain the connection between distributed index and peer-to-peer networking. Based on these we identify the key research problems and imperatives, which are addressed by this dissertation.

2.1 Search basics: Definition of terms

2.1.1 Collection of objects and keys

A database being searched is considered as a collection of key and object/record pairs (Fig. 2.1). The "object" is the data item or entity which we want to retrieve during the search process [15]. In database systems, a record is an object, whereas in the web, objects are less structured and more heterogeneous. The object may be a document file, a webpage, or a multimedia object (picture, audio, video files), etc. The "key" is the identifier or handle for an object which is used in the search process. This key may be a single numerical value, text string as in case of relational database systems, or a set of descriptive keywords to tag pictures or video files or a vector of keywords/terms, as used in vector space information retrieval models [9]. In essence, this key is a descriptor that is a compact and structured description of the object which enables search and retrieval.

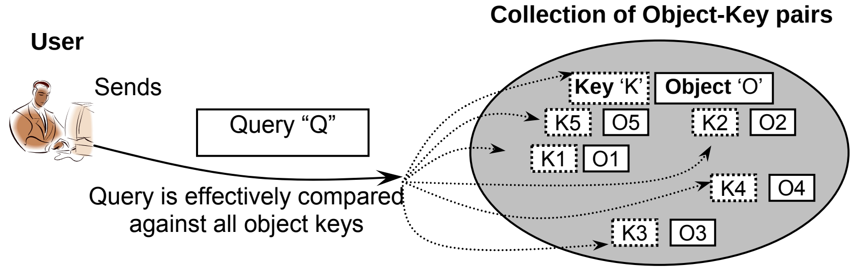

Collection of Object-Key pairs
User
Sends
Query "Q"
Query is effectively compared against all object keys
Key 'K'
Object 'O'
K5
O5
K2
O2
K1
O1
K4
O4
K3
O3

Fig. 2.1 The search paradigm

2.1.2 Search process, query and key comparison

To perform a search, the user has to provide a search criterion. This search criterion is represented by an entity (expression) called search query. The search engine effectively checks all keys to identify which objects have keys that satisfy this criterion and present those objects as results. Looking from another view point, we can also perceive this search query as a representation of what the user wants, i.e. a description of user's intended object. This query is compared with all the keys, which are the descriptors of the objects. When the query is found somewhat similar to a key, then that object is retrieved as result.

Comparing numerical or text string keys are straightforward, but comparing complex descriptors requires sophisticated algorithms. Numerical value and text string keys comparisons results in discrete Boolean values 0 (not equal) or 1 (equal), but complex descriptor comparisons may generate continuous values between 0 and 1 (e.g. vector based information retrieval models [9]).

In those cases of continuous values, a higher comparison value means greater similarity between the object and the search key.

2.1.3 Role of index

To enable faster searching across larger collections, pre-computed indexes are used. Index serves the function of a catalog in a library [15]. It is a data structure of key and object pointer pairs, which is ordered based on the key (e.g. Fig. 2.2 [15]). The ordering and organization of the index drastically reduces the number of comparisons required to search the entire collection. For example, in Fig. 2.2, the hierarchical index structure enables searching using only ($\log_k N$) comparisons instead of N comparisons when index is not used. Here N = number of objects and k = the number of children in each index tree node.

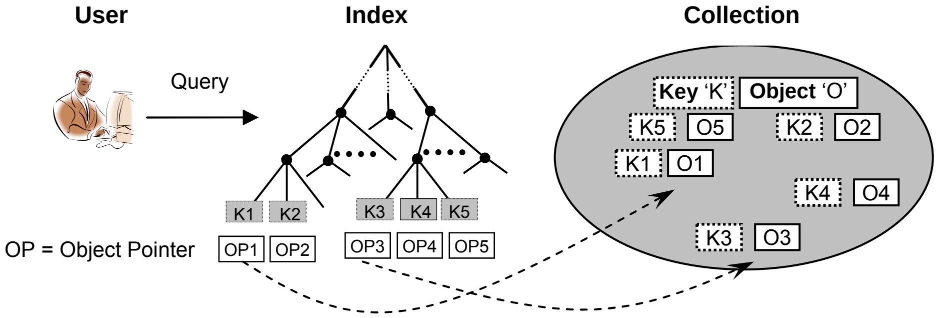


Fig. 2.2 Role of index

2.1.4 Result ranking

Searching in structured database, produce results that exactly satisfied user's search criterion, whereas the results from the advanced search technologies are a collection of probable items that finally may or may not match user's need. In this situation, the best way to reduce user's effort, is to rank the results in terms of relevance so that users can examine only the top few results in the ranked list, to satisfy their need. This requires computing the relevance of searched items and rank of the items. In some cases, for example, in vector space models [9] the relevance computation is based on descriptor comparison that yields continuous values. Objects whose descriptors are more similar to the search key are attributed with higher relevance values. In these cases, the key similarity computation itself gives the relevance metric to rank results. We will use this notion of meaning similarity rankings to decide meaning based message routing and delivery, i.e. deliver messages to those nodes for which the meaning similarity values rank higher.

2.2 Search and retrieval performance metrics

The following metrics are useful to evaluate the performance of any search and retrieval systems. We need to design the meaning based index system in a manner so that these metrics are improved. Hence we need to understand these metrics.

2.2.1 Precision

It is the fraction of the retrieved objects that are relevant to the user. This metric is calculated as:

$$\text{precision} = \frac{\text{number of retrived objects that are relevant}}{\text{number of all retrived objects}}$$

Our objective should be to improve this precision metric.

2.2.2 Recall

It is the fraction of the available relevant objects in the collection that were successfully retrieved. This metric is computed as:

$$\text{recall} = \frac{\text{number of retrived objects that are relevant}}{\text{number of all relevant objects in the collection}}$$

A higher recall value is preferable.

2.2.3 Search response time

This is the time a user has to wait to get all objects from the retrieval system to satisfy his/her need [15]. The user do not have to wait for all available objects to be retrieved, he has to wait for a shorter duration to get the minimal amount of objects that will just satisfy his/her needs. A lower search response is desirable.

2.2.4 Complete recall (response) time

This is the time required to complete the search and recall all the available and relevant objects in the collection [15]. This duration is larger than search response time. We would like to have a smaller complete recall time.

The two aforementioned response time metrics are not traditionally used as retrieval performance evaluation criteria. So far information retrieval has been limited within relatively small centralized systems, where these two times are small enough to be ignored. However these two time responses will become significantly large and distinct in large distributed index systems. Hence we require to measure and manage both of them through superior distributed index system designs, in addition to improving precision and recall performances.

2.3 Research imperatives

To extend the traditional information retrieval paradigm for meaning-based searching we need to compare meaning of the search intention against meaning of all objects. Therefore the key should be a "semantic descriptor" data structure that represents the meaning of user's search intention (query) and the object. The key comparison should ascertain the similarity between meanings represented by two semantic descriptors. Whereas the index should be a scalable infrastructure designed to improve the search speed (response) and recall. Materialization of this paradigm needs: (1) appropriate design of keys; (2) key comparison method; and (3) design of an index system. The four research objectives, as mentioned in Chapter I, are related to these three problems. The design of key and comparison method is addressed by the first two research objectives and the design of the index system is addressed under the last two research objectives. We have to design these three things in a way so that we achieve favorable precision, recall and response time values.

2.4 Significance of "meaning" in human cognition and language

This section presents how human beings comprehend and convey meanings. These understandings are necessary to design realistic meaning representation that can ultimately support human friendly searching. Here we integrate and develop notions assimilated from diverse domains like: cognitive science, neuroscience, linguistics, anthropology and mathematics. Specifically we discuss supporting evidences and ideas that are the key premises for the design of a psychologically realistic descriptor data structure that can represent meanings and enable their comparisons [16].

2.4.1 Sense of meaning

"Meaning" denote ideas and thoughts in human mind that are usually conveyed by natural languages. Human beings convey and comprehend this meaning using concepts. A concept is mental representation of an abstract idea or a physical object (e.g. Car) that is stored, recognized, comprehended and manipulated in human memory in terms of its attributes (e.g. wheel, engine, transportation, etc.) which are used as manipulation handles [17]. A concept can be either generalized or specified to a class having only a single object (e.g. “President Kennedy’s car”). Therefore comparison of meaning actually means comparison of concepts.

2.4.2 Existence of meaning composition process

Earlier we had asserted that meaning composition is an important aspect in meaning representation and composition, and this aspect was not being properly taken care of in existing meaning representation and comparison techniques. Here we present the following evidences to substantiate the existence and importance of the meaning composition process in human thoughts:

1. Meaning composition process is very intrinsic to humans and intelligent primates. Behavioral, neuro-scientific [18], [19] and linguistic [20], [21] studies indicate that humans combines elementary thoughts and ideas to generate more complex ones (composition of concepts), which forms the basis of reasoning, learning [22] and language comprehension ability [23].

2. Human brain has a physical site which is involved in meaning composition and interpretation (semantic processing). This semantic processing is distinct from the language interpretation process which involves part of speech components of the language (syntactic processing). This distinction between syntactic and semantic abilities of the human brain, their separate brain sites and their dissimilar brain activity patterns are supported by several neuro-scientific observations. For example, Broca’s aphasia is caused by damage to some particular areas of the brain known as Broca’s area [24]-[27]. These aphasia patients can comprehend complex meaning which indicates presence of complex thoughts. But they have severe difficulty in communicating them using

language or in interpreting language utterances or writings [27]. This situation is different from Wernicke's aphasia, which is caused by damage of Wernicke's area of the brain. This area is distinct from Broca's area [24], [26]. Wernicke's aphasia patients speak fluently but their words do not have any meaning and the patients have difficulty in comprehending and discerning meanings of sentences.

Brain scans and other neurological studies also indicated that interpretation of language (syntactic processing) and meaning comprehension (semantic processing) are two distinct neurological processes that use different parts of the brain and neural pathways [18], [19], [23], [28], [29], [30]. However during language processing these two neurological processes challenge and test each other for incongruence or look for support and cues in case of ambiguity [19], [23].

3. It seems that a common meaning processing process is involved in all kinds of communications and interpretation of visual sensory signals. Studies on fluent bilinguals, monolinguals, monolinguals with different levels of second language competence suggests that though sites for syntax processing varies for different languages and competency levels, but the neural site, which is used for semantic processing is common. Interestingly this common site is also the Wernicke's area of the brain [31], [32]. In addition, neuro-scientific findings suggest that human brain uses similar neurological processes to comprehend meaning from text and visual imagery [23].

All these evidences indicate the importance of meaning composition in meaning (semantic) processing and comprehension within human mind. This meaning composition is a key cognitive process which comes into play during information processing, interpretation and learning which are intrinsically related to the information search task. Therefore this aspect of meaning compositions has to be supported by the descriptor data structure which will be used to enable meaning representation and comparison.

2.4.3 Notion of complex concepts

We denote the composite meanings that are generated by composing elementary meanings, as “complex concepts”. As these composite meanings are intrinsic in human thought process, so natural language texts involves complex concepts. Therefore these complex concepts will be also involved in describing a text object and user’s search intentions.

2.4.4 Relationship between language and meaning

Natural language utterances are stimuli which invoke thoughts and cognitive processes that reconstruct the meanings within human mind. These processes constitute meaning interpretation of the natural language instance (writing or verbal utterance). This is supported by the fact that sometimes language syntax alone does not sufficiently represent the entire meaning, but yet humans understand the full meaning [21], [22]. For example the following sentence invokes more than one meaning:

Jane attempted to pass the test

There is a tacit meaning which indicates that Jane took a test, in addition to the explicit one which is about her attempt to pass. The compositional principles, that govern composition of semantics, are not always evident in the language’s explicit syntactic representation, but nonetheless they definitely come into play during communication, just before language generation or during language comprehension. So linguists hypothesize that there must be another parallel composition process (Fig. 2.3) which is taking place in the human mind in addition to syntactic processing. This argument is also congruent with the evidences presented in section 2.4.2. This notion is fundamental to language generation and meaning comprehension process (Fig. 2.3) [33].

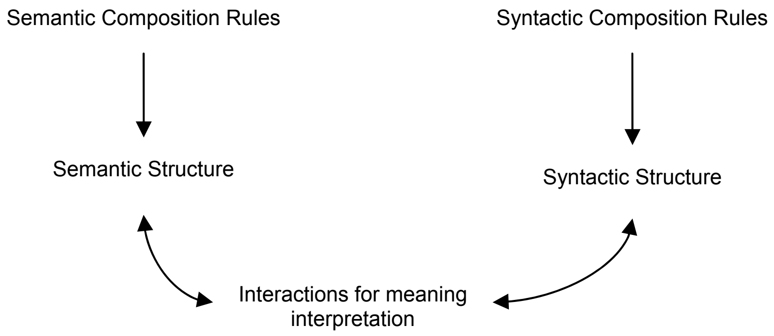

Semantic Composition Rules
Syntactic Composition Rules
Semantic Structure
Syntactic Structure
Interactions for meaning interpretation

Fig. 2.3 Semantic and syntactic processings are distinct

2.4.5 Simple principles of meaning composition

“Simpler syntax hypothesis” [21] and “parallel architecture” theory [22] argue that in human mind, semantics or meaning has its own rules of composition, which are different from grammatical composition (syntactic) rules. Simpler syntax hypothesis [21], [22] proposed that rules for composing meaning are inherently simpler. A simple collection of elementary meanings is good enough to represent a complex meaning. There is no need to specify the ordering of the elements or the exact nature of association between individual elements. This is supported by evidences like: presence of compound words in modern natural languages; and speech of children, pidgin language speakers, late language learners who communicate complex ideas and meanings with a collection of simple words and terms which are devoid of any grammatical relationship. Some examples are illustrated below:

Children: “Walk street; Go store”; “Big train; Red book” [34]

Late language learner raised in the wild: “Want milk”; “Big elephant, long trunk” [34]

Pidgin language: “And too much children, small children, house money pay”; “What say? Me no understand” [34]

English compound words: “winter weather skin troubles”; “health management cost containment services”; “campaign finance indictment” [35]

All of these utterances are just collection of words, and each word conveys a simple elementary concept. These collections of words together convey a complex concept. Here the meaning composition is happening without aid of sophisticated syntactic (grammatical) rules. This indicates that a simple collection of words by itself can represent a complex meaning. This notion is also supported by [30]. Here the words stimulate elementary thoughts, meanings or concepts in human mind which then combine and invoke the complex meaning [20]. Ref. [23] and [34] proposes that ability for this kind of semantic composition is intrinsic to human brain and this is also the basic tier of language comprehension ability.

2.4.6 Generative mechanism of meaning composition

“Parallel architecture” theory proposes that a generative (recursive) mechanism for semantics [22] exists in mind that generates complex semantic structures based on two simple rules. The first rule allows representation of complex meaning as a collection of elementary meanings, and the second one allows composition of collection of meanings to form higher level collections. By applying these two rules, a hierarchical collection (tree or nested set structure) of elementary concepts / meanings can be generated. This is explained with a simple example as in Fig. 2.4. In this example, we assumed that the entire text object consists of only a single sentence, therefore the hierarchical structure as shown in the figure can represent the meaning of this single sentence object. This simple example (Fig. 2.4 [13]) was provided to convey the notion of hierarchical meaning composition, but it should not construed that we are advocating that all sentences present in a given text, should be converted to this kind of structure to represent meaning of the entire text. In Chapter III we shall propose that only this structure should be generated for the central theme of the text, not for each and every sentence. This kind of hierarchical structures can be used to represent a complex concept/composite meaning within computers.

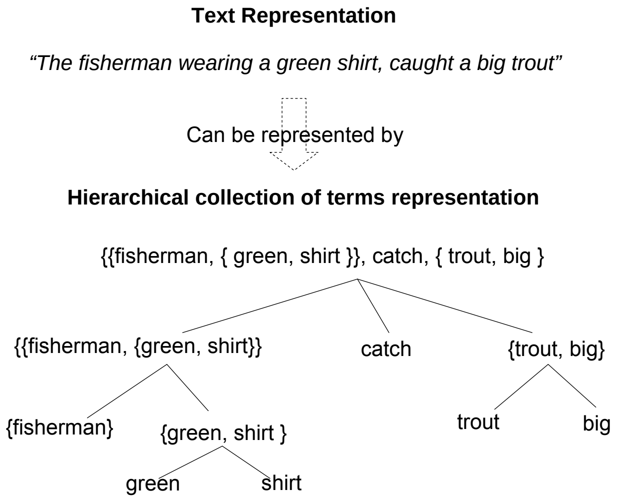

Text Representation
"The fisherman wearing a green shirt, caught a big trout"
Can be represented by
Hierarchical collection of terms representation
{{fisherman, { green, shirt }}, catch, { trout, big }
{{fisherman, {green, shirt}}
catch
{trout, big}
{fisherman}
{green, shirt }
trout
big
green
shirt

Fig. 2.4 Hierarchical collection of meanings

2.4.7 Simple composition and memory models in cognitive science

The notion that collection of elementary concepts can be a suitable representation of complex concepts is also supported by cognition science research like [30]. This notion is also coherent with the spread activation model of associative semantic memory [36]-[38]. Each of the elementary thoughts stimulate independent spread activations in the human semantic memory which all of such activations acting together finally gives rise to a thought in a associative semantic memory [36]. This spread activation model explains how a collection of elementary concepts can invoke a complex meaning.

This notion also corroborates well with the semantic memory related empirical observations made by [17], [39]. These studies indicate how multiple elementary concepts are useful in retrieving complex concepts through node activations and how these complex concepts are manipulated in human mind using these elementary concepts as search handles. In fact a larger number of elementary concepts are likely to generate better activation (recalling) or invocation of a complex concept.

2.4.8 Realizations and implications

All these above discussion imply the followings:

1. The phrase “meaning representation” or “meaning comparison” is actually a figure of speech. This is so because, meaning interpretation and comparison are mental processes, therefore they can not be literally described, represented or compared within computers. What can be compared are the stimuli, i.e. the symbolisms that invoke the mental process which we denote as “meaning”. Natural language is one form of symbolism [40] which achieves this. Hence by “meaning representation”, what we actually mean is an alternate and computable form of symbolism that would invoke similar mental process as the text (assuming if we humans had learnt that symbolism as we learn languages). Similarly, by “meaning comparison” we mean comparing this computable kind of symbolism within computers.

2. To generate meaning representation data structures for a given natural language text, we need to first extract meanings from the text. During that process, we can not depend solely on natural language syntax analysis techniques. We also need additional knowledge based mechanisms to mimic the brain function where the meaning of a given text is interpreted using knowledge artifacts (i.e. experience) gained in past. This implies

that a two phase process is necessary, where we first need to create knowledge artifacts to form an extensive corpus (i.e. experience), and then use this artifact corpus to generate the meaning representation of a given text. Both these phases will need supervised, unsupervised and various other kinds of machine learning techniques [41].

2.4.9 Need for a suitable mathematical model

A basic hierarchical tree structure is reasonably good candidate to represent a complex meaning. To use this model for comparing meaning inside computers, we need mathematical logic and a suitable computational model. This computation model is needed to compare two such trees at high speed. To get an idea about what kind of formal logic and models may be suitable for high speed computation, in the next section, we examine some of the existing models that had been used in past to represent meanings within computers.

2.5 Existing meaning representation methods

2.5.1 Descriptors for documents and concepts

Broadly two kinds of descriptors, one for describing objects (documents), another for describing a concept, are available in the literature. Semantic descriptor for documents represents meaning of an entire document object, whereas descriptor for an elementary concept represents the meaning of an elementary concept. Both kinds of descriptors are necessary. The object descriptor designs that are available in literature are based on vector, set and Galois lattice [42],[43] based data structures, whereas the concept descriptor design reported in [44] is based on graphs. On the other hand, the descriptor design which we propose in this dissertation unifies the notion of document (object) and concept descriptors. This particular design views the entire object description as a large complex concept. Therefore the object descriptor can be represented by a concept descriptor. We present this idea later in Chapter III.

Vector and set based descriptor designs are the most adopted ones. We explain the fundamental notions behind these designs in details. This is needed because we extend these notions to design our proposed descriptor. All these existing designs and their criticisms are discussed in the subsequent sections.

2.5.2 Set based (Boolean) model

Here “set” has the same connotation as in formal set theory in mathematics. In this set based model, the document’s descriptor is considered as a set of index terms or keywords which are either picked up from the object (text) or assigned by human indexers to describe the object [7]. The query is a set which is either explicitly expressed as a straightforward set or as an implied set that is expressed in form of Boolean conjunction, disjunction and negation expression involving some of the index terms (Fig. 2.5). For example, in this figure, the query expression “sales” AND “manager” implies those sets which have both terms “sales” and “manager”.

The similarity value between document and query descriptors is always Boolean 0 (not similar) or 1 (similar). It is computed as follows. It can be ascertained by checking whether any of the query descriptors terms are present in the object descriptors or not. If the term is present then the value is considered as 1, otherwise it is 0. This is the way of verifying whether the given Boolean condition expressed by the query is satisfied for the object descriptor (as illustrated in Fig. 2.5) or not. The documents, whose descriptors yield a value of 1 when compared for similarity against the query descriptor, are returned as search results.

In the example in Fig. 2.5, for the document object O_3, we considered the stemmed (base) terms "drink", "sales", "manager", etc., as the terms for indexing.

Objects	**Set based object descriptors**
O_1: "*The sales manager looked at the receipt. Then he took it*"	**D_1** = set { sales, manager, look, receipt, took}
O_2 : *"The sales manager took the order."*	**D_2** = set { sales, manager, took, order }
O_3: "*It was not the sales manager who hit the bottle that day, but the office worker with the serious drinking problem.*"	**D_3** = set { sales, manager, hit, bottle, day, office, worker, serious, drink, problem }
O_4: "*That day the office manager, who was drinking, hit the problem sales worker with a bottle, but it was not a serious*	**D_4** = set { sales, manager, hit, bottle, day, office, worker, serious, drink, problem }

Queries		**Results generated by queries**
Q_1 : "sales" AND "manager"	➤	**R_1** : O_1 , O_2 , O_3 , O_4
Q_2 : "receipt" OR "order"	➤	**R_2** : O_1 , O_2
Q_3 : "took" AND "order"	➤	**R_3** : O_2

Fig. 2.5 Search operation with set based descriptor

Stemming is the process which reduces inflected and derived words to their base or root (stem) form. For example, after the stemming process, the words "use", usefulness" and "useful" become a single word "use". We ignored the terms "the", "he", "at", "then" and "it" because they do not help to distinguish objects from each other. Generally these terms do not carry any distinguishing information because they occur with high frequencies in almost all objects. Considering the set of four document objects: O_1, O_2, O_3 and O_4 as the collection, the three queries: Q_1, Q_2 and Q_3 generated the three result sets: R_1, R_2 and R_3. The document objects O_3 and O_4 are borrowed from [12]. These two texts have similar terms but they convey very different meanings. Thus set based approach would not distinguish these two objects O_3 and O_4. This shows that the set model is not a good semantic descriptor.

2.5.3 Vector based models and associated techniques

In vector models (VMs), the meaning of an object is represented by a vector/tensor in a vector space [8], [9], [12], [45]-[48]. Different kinds of vector models are available. Some models involve large dimensional vector spaces, where each basis vector corresponds to a real world term, concept or a phrase which are either selected from the text object, or from the object's metadata. Other more sophisticated models involve smaller finite dimensional vector spaces, where each dimension is latent semantic feature having only statistical significance, thus they can not be attributed to a single term or real world meaning (e.g., latent semantic dimensions as in Latent Semantic Indexing (LSI) [8]).

Algebraically, the vector/tensor is represented as a sum of scalar (w_i) weighted basis vectors (v_i) (Fig. 2.6). In a simple term based VM, each basis vector denotes an elementary meaning which is a term or a phrase from a controlled vocabulary/dictionary [9]. Within a computer, the entire meaning vector is represented by a set or table of character strings (each representing a basis vector) and corresponding weights (Fig. 2.6). In rest of this dissertation, we call this table as the coefficient table. These VMs assume an infinite dimensional vector space where the meaning representation is a sparse vector having finite number of basis vectors with non-zero weights/coefficients. In the example in this figure, the compound term "sales manager" represents a single entity, hence considered as a single term. To consider such compounds as a single term or not will depend on the implementation of the model. Recognizing compound terms improve the performance of the meaning representation model.

In all these vector models, the magnitude of semantic similarity between two descriptor vectors, D^1 and D^2 that represent two meanings, is computed as the dot (cosine) product D^1•D^2 (Fig. 2.6). A similarity value of zero means that two vectors are dissimilar (orthogonal) and a higher value (≤1) indicates more similarity.

O$_1$: "*The sales manager looked at the receipt. Then he took it…..*"

$$D^1 = w^1_{sales\ \ manager} \cdot \vec{v}_{sales\ \ manager} + w^1_{receipt} \cdot \vec{v}_{receipt} + w^1_{look} \cdot \vec{v}_{look} + w^1_{took} \cdot \vec{v}_{took}$$

Where $\sum_i \left(w_i^1\right)^2 = 1$

Vector/Tensor D^1 in memory

Basis vector V$_i$	**Coeff.**
"sales manager"	$w^1_{\cdot\text{sales manager}}$
"receipt"	$w^1_{\cdot\text{receipt}}$
…..	…..

O$_2$: *"The sales manager took the order…."*

$$D^2 = w^2_{sales\ manager} \cdot \vec{v}_{sales\ manager} + w^2_{took} \cdot \vec{v}_{took} + w^2_{order} \cdot \vec{v}_{order}$$

Where $\sum_i \left(w_i^2\right)^2 = 1$

Vector/Tensor D^2 in memory

Basis vector V$_i$	**Coeff.**
"sales manager"	$w^2_{\cdot\text{sales manager}}$
"took"	$w^2_{\cdot\text{took}}$
…..	…..

Where similarity between two objects =

$$D^1 \bullet D^2 = w^1_{sales\ \ manager} \cdot w^2_{sales\ \ manager} + w^1_{took} \cdot w^2_{took} + w^1_{order} \cdot w^2_{order}$$

Fig. 2.6 Vector model of meaning representation and comparison

2.5.3.1 Assignment of weights

A term which is an important distinguishing factor should have a higher weight. There are several alternatives ways to assign these weights, the most popular being the term frequency-inverse document frequency (TF-IDF) scheme [9]. In this scheme a larger weight is assigned to a term if it occurs frequently in a document but not too frequently in all documents in the collection. This larger weight is assigned because such terms should have higher contribution in distinguishing the document from others.

2.5.3.2 Searching using vector based descriptor

The search query is given as a string of terms or in form of natural language text. A vector based descriptor for the search string is generated and compared with the objects' descriptors using cosine similarity. The result objects are ranked based on the cosine similarity value. This is illustrated in Fig. 2.7 for the same objects used earlier in Fig. 2.5. The collection of terms inside the quotes are treated as a single compound term and terms without quotation are treated as separate terms.

Queries		Ranked results generated by queries
Q_1 : "sales manager"	→	R_1 : $O_2 > O_1 > O_3$
Q_2 : receipt	→	R_2 : O_1
Q_3 : took order	→	R_3 : $O_2 > O_1$

Fig. 2.7 Search operation with vector based descriptor

2.5.3.3 Limitations of vector models

The traditional term vector model does not address the synonymy and polysemy problems [8]. Synonymy means different words having similar or same meaning, and polysemy means a single words being used to connote different meanings. The basic LSI vector model [8] recognizes the meaning context better than term vector approaches, however LSI ignores new infrequently used terms which might convey important meaning information (e.g. in niche scientific domains). In addition, vector models alone are not sufficient, hence vector model based search systems often need additional techniques, for example, methods to identify topic concepts (to generate metadata) from the text [45], [49] or disambiguate terms to resolve the polysemy problem [50], etc.

Vector based models are also computationally expensive because the way the required computations are carried out at present. For example, the TF-IDF computation assumes presence of a centralized corpus. It also implies use of a centralized inverse document frequency computation and the index. Even though computation can be parallelized using map-reduce kind of paradigms [51], but such parallelization is also expensive.

Vector based approach also has an inherent weakness in representing complex descriptions which are based on composite meanings (concepts). This results in failure to discern between descriptions that have common keywords/terms but have very different meanings. This is well explained by objects O_3 and O_4 as in Fig. 2.5, which have similar terms but convey different meanings. A simple vector based similarity computation erroneously reports that these objects are similar (the TF-IDF based cosine similarity is 0.998 when computed against a given text corpus). This problem exists primarily because vector based models are based on a belief that elementary syntactic terms contain the meanings and this meaning can be entirely captured by a flat collection of the terms available in the text.

Considering compound words or named entities like "*sales manager*", "*office manager*", "*office worker*", "*sales worker*" or "*problem sales worker*" as the terms, can improve the situation but will not report absolute zero similarity in all possible cases. Thus considering large numbers of *n*-grams (specific phrases composed of *n* terms) can not entirely avoid this problem in all situations. Instead of machine generated vectors, if the vectors are based on standardized topics assigned by human indexers, then some of these problems can be avoided. However this requires human involvement.

2.5.3.4 Use of vector products and other sophisticated algebraic operators

Some of these limitations have caught attention of researchers and enhancements like vector products [12], [46] and more sophisticated vector operations like [48], [52], [53] are being proposed. However these are still term centric low level approaches which will have limited impact in composite meaning representation of entire large documents. We need new kind designs for semantic descriptors which are more meaning centric.

2.5.4 Graph based model

A composite meaning can be represented as a graph where each elementary concept is a node and the relationships between the elementary concepts are denoted as edges connecting those concepts [44]. This design is based on the notion of conceptual structure in human mind as proposed by [54]. The technique to compose two elementary concepts to get a representation of the complex concept is also available from [44]. It might be possible to compare complex concepts with graph comparison techniques like in [55],[56], however these are ungainly and computation intensive.

2.5.5 Galois lattice model

This document descriptor design [42],[43] is based on Formal Concept Analysis (FCA) theory [57] from mathematics and statistics. A technique to compare similarity of Galois lattice based descriptor is also available from the same researchers [42],[43]. Whereas a technique to compose Galois lattice based descriptor is available from [58]. This Galois lattice based design assumes that concepts are only based on closed classification taxonomy (i.e. all possible concepts are incorporated in the taxonomy and all concepts which are not in this taxonomy are ignored). Based on this over simplistic and closed taxonomy it elaborately models all possible concepts present in the entire context (in other words it defines the entire context in terms of finite number of concepts). This is an unrealistic closed and rigid model that does not reflect the realities of human thought processes. Description of the entire context with all possible concepts in the universe may not be necessary.

Such elaborate rigid formal modeling and related computation does not deliver any substantial gains in terms of meaning comparison confidence and it is also too expensive in terms of computation. This design does not seem to be based on any particular model and current understanding from cognitive science. On the other hand the notion of requirement for parsimony in cognitive processing [59] does not agree with the elaborate FCA based modeling. In addition, harvesting attribute and object relationship from the object text description as suggested in [43] is not a good solution because all the attribute-document relationships required to define the concept may not be available in the document text. This necessitates processing all documents in the corpus to index even a single document. Though a technique to compose Galois lattice is available but it does not elegantly support hierarchical concept composition in a manner that happens in human mind. Therefore Galois lattice design does not achieve anything substantial in terms of meaning representation or comparison, which LSI vector or set based exploded conceptual searching (refer section 2.7.1) can not.

2.5.6 Meaning representation using mainstream semantic web standards

Mainstream semantic web standards like: RDF [60], RDFa [61], microdata [62], microformat [63], HTML5 [64], has begun providing ways to represent meaning in terms of attributes (object) and their relationships (predicate) with the subject. These representations are more structured and hence more computable to enable formal reasoning/inferencing capabilities, thus can be used to enable more precise searching than the vector models. However the way the required computations are currently carried out in sequential processors, makes them computationally expensive and slow.

2.5.7 Unsuitability of existing meaning representation techniques

The aforementioned meaning representation and comparison techniques do not support meaning composition elegantly. They are also not congruent with cognitive and linguistic theories and are not computationally efficient. Therefore there is a need for a new semantic descriptor design and a new descriptor comparison technique.

2.6 Existing tree matching algorithms

From the discussion in section 2.4.6, we realize that nested set or hierarchical tree structure seems to be a promising abstract data structure for meaning representation. We need suitable tree comparison algorithms to compare such unordered trees (i.e. where children can be unordered), which can have more than two children at the nodes, and which have only leaves and no labeled nodes. Two trees are considered same when they have common embedded components, which are either sub-trees or leaves. These algorithms should either provide a single similarity measure between two trees or simply identify the matching trees from a large collection of trees, when one tree is provided as the query. The second option is similar to set based search operation where similarity value is binary, either 1 (similar) or 0 (not similar) and the first one is akin to the vector model based similarity measure. Both approaches suit our purpose for search and information retrieval task. However we prefer to have a technique which gives a continuous similarity measure which can be useful to rank results.

Some suitable techniques are available from phylogenetic tree matching research. These are based on edit distance [65], Robinson-Foulds distance [66], [67] or lowest common ancestor [66] measures. Edit distance indicates how many insert and delete changes need to be made to transform one tree to the other. There are other tree comparison techniques which are for ordered trees, or for trees with labeled nodes, therefore these are unsuitable for our purpose.

All these algorithms are computational intensive and unsuitable for fast processing. Interestingly there had been some attempt to accelerate tree comparison computations by specialized hardware processor [68], unfortunately the underlying technique is for comparing trees with labeled nodes.

2.7 Existing meaning based search techniques

A variety of meaning based search technologies are available, here we discuss a few under two categories: conceptual and semantic searching [16].

2.7.1 Conceptual (exploded) searching

This kind of searching expands the scope of search by automatically including search keywords that are conceptually related to the given search/query keywords/terms. An example is Pubmed's exploded search service [69] where a user can provide a standard Medical Subject Heading [70] "Glucose Metabolism Disorder" as the search key to get all bio/medical science publications (objects) indexed under that topic. The results will also include objects that are indexed under topics like: "Diabetes Mellitus", "Glycosuria", "Hyperglycemia", etc., in addition to those that are indexed under "Glucose Metabolism Disorder". In medical science the concept "Glucose Metabolism Disorder" (a hypernym) encompasses "Diabetes Mellitus" or "Glycosuria" (hyponyms), therefore the search is expanded to include all hyponyms of "Glucose Metabolism Disorder". The hyponym-hypernym relationships between index terms are based on controlled taxonomies.

2.7.2 Semantic searching

This kind of searching has the flexibility to tolerate interchangeable or related search terms as long as these terms broadly convey similar meanings (semantically related). This is best explained with an example. The gene "PTPN22" has several (single nucleotide polymorphic) variants: 1868C, 1858T, etc. The gene "PTPN22" is also known by synonyms: "IPI00298016", "PTPN8: Tyrosine-protein phosphatase non-receptor type 22". PTPN22 is related to a protein "LyP", whose function is to bind with another molecule known as "CsK". Therefore all these terms "PTPN22", "CsK", "LyP", and "1858C", etc., are related. When these related terms are used as interchangeable search keywords the semantic search engine should retrieve a similar set of documents as results in all cases. Latent semantic indexing (LSI) [8] is one method which can implement this kind of searching.

2.7.3 Significance in meaning representation design

There are several lessons to be learnt from these conceptual and semantic search technologies, which are relevant for meaning representation design. These are as follows:

1) We have to incorporate meaning subsumption, so that more general meanings are incorporated in a representation of a specific meaning. This is needed to ensure similarity of a specific meaning with its broader meanings. Preferably this should not only hold true for meaning of individual keywords, but also for holistic composite meanings.

2) We need mechanisms to identify the common underlying meanings, even though they might be conveyed by a different set of keywords. These keywords may be either synonyms or contextually related keywords. This should hold true for simple elementary meanings that are conveyed by a single keywords as well as composite meanings.

3) We should incorporate methods to ensure that the composite meanings, that are conveyed by two different but partially overlapping sets of elementary meanings, should be comparable with each other.

To achieve the above, we need appropriate techniques to extract the meaning and composition information from natural language texts. Some of these are discussed in the next section.

2.8 Relevant natural language processing techniques

We require more advanced natural language processing techniques to extract holistic meaning from text as opposed to harvesting simple index terms. Some of the relevant techniques which are presented below.

2.8.1 Topic identification techniques

A given text can be automatically and successfully classified based on topic of the content by topic identification technologies [45], [49], [50]. These apply machine learning and pattern recognition techniques. These classification techniques can be reused to identify the particular classes which a given document belongs to, and then assign the class names as topics for the document. These techniques will have a role to play in the descriptor generation process as proposed by us in Chapter III.

2.8.2 Presence of semantic structures in natural language texts

Natural language processing techniques often focus on a small portion of the text. This is called a window and this window is moved along as the entire document is processed. Machine analysis (word frequency based) like [71], demonstrates that the terms have high correlation within a window, and this correlation decreases across windows based on the relative distance between windows. This means that some information is available within a window which may be used to define the contextual meaning of all terms within that window. This kind of information may be used to disambiguate the meaning of a term in the given context. In addition, this also indicates that there is a well defined hierarchy of meanings in human written natural language texts. Therefore it may be possible to identify such hierarchy of meanings and it may be possible to transform such meaning hierarchies to a meaning representation suitable for computations (comparison) using some mediating techniques.

2.8.3 Named entity recognition techniques

Atomic meanings that can be categorized to a particular class in the taxonomy are called named entities in natural language processing domain. For example, in the text in Fig. 2.4, the entity "fisherman" will be recognized as a subclass of "person", if that class exists in the taxonomy. Similarly, "Type 1 diabetes" will be recognized as a disease from medical/bioscience related publications. Language syntax (grammatical) analysis and various other machine learning techniques like [72]-[75], are used to disambiguate terms and identify these entities. State-of-art named entity recognition (NER) techniques have low error rates (~6%) which are comparable to human detectors (~ 3%).

2.8.4 Knowledge extraction techniques

In natural language processing domain, the relationship between entities and taxonomic classes (e.g., relationship between gene and disease) is considered as a knowledge artifact. These relationships are formally documented in a knowledge repository (or data structures) called an ontology. Various machine learning and language syntax analysis technologies like [76]-[80] are available to detect a relationship between two or more entities in a given specific portion of text and construct ontologies by processing multiple text documents [81]-[89]. Once constructed, ontologies provide readily available patterns to detect relationships among entities in a given document and recognize context and contextual meanings in a given text. Once identified, these relationships can be used to identify which concepts should be composed together.

2.8.5 Text summarization and discourse analysis

It is possible to automatically create abstracts of text documents [90]-[94]. These technologies first identify the portions of text that contains the essence of the entire document and then built the abstract. Automatic generation of abstracts are interesting because that way it is possible to further process the abstract to identify the broad topic of the text, extract the most important named entities or relationship between entities without getting distracted by rest of the voluminous text. The most important information is concentrated in the abstract, therefore any information that is extracted from it is likely to be more relevant, accurate and represent the overall meaning of the entire document/object. Some of these are based on discourse analysis techniques [91]-[94] that can identify which sentences are more important and which are the most important information bearing part of these sentences.

2.8.6 Interpretation of actual meaning of a sentence

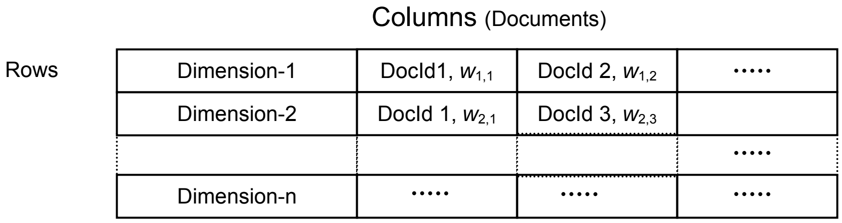
Deconverter technologies [95] are available that can parse a sentence to identify the meaning that is being conveyed by that sentence and then represent the meaning in United Networking Language form that is computable [96]. Using these technologies one may parse sentences of a machine generated abstract to generate their UNL form. Then from the UNL form one can identify the central meaning of an entire document.

2.9 Meaning based indexing

2.9.1 Inverted index for vector models and Galois lattice based indexes

Indexes are pre-computed data structure of key and object pointer pairs which enable faster searching of objects. Set and vector based search and information retrieval systems have their own inverted index scheme which is a colossal incidence matrix. Fig. 2.8 illustrates the abstract view of a typical inverted index, which is a lookup table (or matrix) of basis-vector terms as rows and object/document location as the columns. In this example, the second row in the inverted index table corresponds to “Dimension-2” and has two columns, each of which contains a pair. One of the pair is (DocId 1, $w_{2,1}$). Suppose this table is for a term vector model, and the Dimansion-2 corresponds to a term “sales”, then the document identified by DocId 1 has a term “sales” and has a corresponding weight $w_{2,1}$.

Each terms of the query (e.g. “sales”) is used as the key to identify the documents column (e.g. Doc Id 1, DocId 3) which has these terms and the corresponding weights (e.g. $w_{2,1}$, $w_{2,3}$) of these terms. Once the weights of these terms in each document is identified, then for each identified document, these document term weights are multiplied with corresponding term weights of the query to identify the dot product between the document vector and the query vector. Based on the dot product value the identified documents are ranked and returned as result. In case of latent semantic indexing, instead of terms these basis vectors are latent semantic dimensions [8].


Columns (Documents)
Rows

Dimension-1	DocId1, $w_{1,1}$	DocId 2, $w_{1,2}$	·····
Dimension-2	DocId 1, $w_{2,1}$	DocId 3, $w_{2,3}$	
			·····
Dimension-n	·····	·····	·····

Fig. 2.8 Abstract inverted index in a typical search engine

Similarly, proposals on large (and expensive) Galois lattices as concept based indexes are available in [97]. Neither inverted index nor Galois lattice based index enables incorporation of composite meanings, therefore unsuitable for high quality meaning based searching.

2.9.2 Index scaling techniques

In large search engines, for scalability, inverted indexes are partitioned and distributed into a cluster of nodes, as mentioned earlier in Fig. 1.2 in Chapter I [4]. This index partitioning scheme for load sharing is a proven technique. However improvements can be made, as suggested in Chapter I, on how to partition the index to further improve the scalability for a given amount of resources (i.e. the index can be made to have a higher query servicing capacity).

2.9.3 Equivalence between searching and routing problems

An index structure, for example a *k*-ary indexing tree, can be represented as a tree graph. To search an object, we traverse the index tree and reach a particular leaf and test whether that leaf node corresponds to the required object or not. This index traversal is equivalent to a graph traversal in the tree graph. During index tree traversal, at every node, we decide on which next child node we should move to. This is also a routing decision, where we decide which should be our next node destination. The index nodes can be conceived as routing nodes in a graph/network, and the search key can be conceived as the message which has the search (query) key as the destination address. Thus, every search problem has an equivalent routing problem. Therefore, from algorithmic viewpoint, searching and routing is one and the same problem. Due to this very reason, P2P networks [98]-[113], where a query is routed to a particular node which can best service it, can act as a search index. Therefore in the next section [15], we study P2P networks to understand the routing related challenges from theoretical viewpoint, then solve them to utilize the solutions to construct a meaning based index system.

2.9.4 P2P network as a distributed index

In P2P networks (P2PN) the searching workload is distributed across multiple nodes which store the document objects. These underlying principles of P2P networks can be applied to solve the query delivery problem in a distributed index which has very large number of index pools. Thus these principles may help further scaling of a distributed index. Related rudimentary works on P2P networking and query routing can be found in [98]-[116].

A P2PN can be a scalable search solution because it can simultaneously carry out distributed searching for data objects in multiple computers (nodes). Here the computers (nodes) contain document objects and these computers interact with each other using IP (TCP or UDP) packets. Here a search query is embedded in a query message, which is distributed to all the participating nodes where multiple instances of the same search are simultaneously executed. P2PN is the way how these computers are logically connected and how the search query is routed through them.

Users send a query to any random computer and the computer which receives the search query delivers the query message to those destination computers which might host objects that are being searched for. This query delivery may constitute routing of the query through multiple routing hops till it reaches a node with relevant objects.

2.9.4.1 Search operation

Fig. 2.9 presents an abstract view of a P2PN and its distributed search operation. Here the interactions, message paths and messages are identified with labels and serial number (i) to (iv). A randomly selected computer/node “N2” accepts a message from the user (transaction labeled as “(i)” and indicated by a arrow) and forwards (routes) it to computer “N3”, which finally delivers the message to computer “N6” (transaction (ii) & (iii)). The node (“N6”) that contains the relevant objects directly presents the results to the search initiator (transaction (iv)). In this manner query routing in a P2PN can enable distributed and scalable searching. In more primitive and simpler variety of P2PNs each node may simply broadcast the query to all other neighboring nodes that are directly connected to it. This flooding of the entire network with copies of a given query ensures that the query reaches all nodes. This strategy avoids need for sophisticated query routing mechanisms but it causes severe load on the network. Based on their design sophistication, these P2PNs can be categorized under three different generations as presented below. The P2PNs which does not implement any particular topology has a random network topology [117].

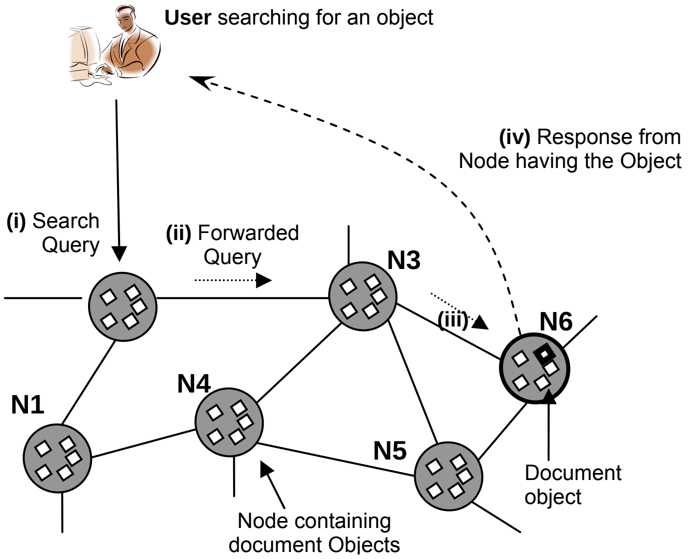

User searching for an object
(iv) Response from Node having the Object
(i) Search Query
(ii) Forwarded Query
N3
(iii)
N6
N4
N1
N5
Document object
Node containing document Objects

Fig. 2.9 Distributed search operation in a P2P network

2.9.4.2 First generation P2P network

This generation of network implemented searching by flooding the network with the query. Gnutella version 0.4 [109] is an example of this class where all nodes have equal status and capability in a truly peer-to-peer configuration. These schemes generate high volume of message traffic due to naïve searching by query flooding. They also do not bound the end-to-end query routing response because in any random network, this time can be arbitrarily large. The rational for this assertion will be discussed in sections 2.10.4.3 and 2.10.4.6.

2.9.4.3 Second generation P2P network

This class of networks selectively routes query messages and uses mature networking organization mechanisms to avoid network flooding and yet carryout searching in all nodes. Here query routing tables are used to progressively route search queries towards nodes that are likely to have matching content. The query routing tables are maps between search keywords/terms (key) and next hop node addresses (destination). Various routing table updating schemes are used that either enables permanent table entries or temporary caching depending on use and age of the entry. Examples include: SemAnt [100], Range Addressable Network [114], NeuroGrid [98], Intentional Name System [115], [111], [112] and Remindin [113].

In these proposals, the routing tables are implemented as distributed hash tables (DHT) where a single keyword/term serves as the routing table key. This is a severe limitation because this kind of query routing is based on strict keyword matching and not meaning based. For example, if the search key (intention) is “P2P networks software products” then the query might be routed towards all nodes that stores documents that has the following standard keywords/term/phrases: “P2P networks”; “software”; “products”; “software products”, in addition to the nodes that have the required document having conjunction of all the keywords, but will not forward messages to nodes having relevant objects having description keywords “Gnutella” or “Neurogrid”, etc. This kind of query forwarding without sophisticated meaning based comparison at the routers will cause more message traffic and yet yield lower search recall.

2.9.4.4 Third generation P2P network

This group of networks used premeditated topology to manage search response time. These networks actually propose to organize their network according to a specific topology. However neither formal justifications were given nor evaluation of the used topologies were carried out to explain why those specific topologies were chosen. These designs also lacked a suitable design for the keys used in the routing table. Examples include: Seers Search Protocol [99], Schema Based P2P [101], Content Based Addressing and Routing [102], SenPeer P2P Data Management System [103], H-Link semantic routing [104], D2B [105], The Socialized.Net [106], PlanetP [107], Makalu [108], CAN [110], and [116].

2.9.4.5 Limitations of existing P2P networks

The aforementioned three generations of P2P networks have several weaknesses, hence they are not readily useful to materialize a distributed index system for mainstream usage. These weaknesses are follows:

1. *Sophisticated, meaning based search not yet possible*: Most designs use distributed hash tables which are based on exact matching of keys, so meaning based searches are not possible.
2. *P2P network designs do not attempt to adequately improve retrieval performance metrics*: In a P2P network, the evaluation criteria should encompass all four metrics: precision, recall, search response and complete recall time, instead of only open ended precision and recall metrics. However this is not a common P2P network design practice, therefore many aforementioned designs often do not address key questions like: (1) how to reduce the end-to-end query routing response time; and (2) how to increase query routing success rate. Hence these proposed designs ignore the fundamental requirements and have not been able

to apply the right solutions. Managing routing success, response time and messaging overheads requires applying network science fundamentals as available in [117], [118] and [119] to choose an optimum topology. Only some of the third generation P2PNs have started to leverage topology to manage routing success and response time.

2.9.4.6 Useful lessons from P2P network designs

It is important to note that even though the P2P networks are not be readily suitable, however there are relevant lessons to be learnt from their weaknesses. These would be useful to guide the design of the required SRN. Some of these lessons are as follows:

1. Peer-to-peer networking can support distributed searching and reduce number of comparisons.
2. Search query should be selectively routed towards resources that can best service the query.
3. An optimal overlay network topology is needed for high query routing success rate and low response time.

2.10 Networking science (graph theory) fundamentals

2.10.1 Role of network science (graph) theories

Traditional numeric key based indexes like binary or k-ary trees are based on tree topologies (as illustrated earlier in Fig. 2.2). However such tree topologies can not be constructed when the address keys are non-numeric entities, for reasons that were discussed earlier in Chapter I. Therefore alternative network topologies have to be adopted which can work with non-numeric keys. On the other hand, P2P networking solutions could have been used as meaning based distributed index infrastructure, provided they supported meaning based operation and adopted optimal topologies. Network science provides us with the understandings that are necessary to choose and adopt appropriate network topologies for building meaning based index systems based on the P2P network paradigm. To gain useful insights, in the following sections, we discuss some fundamental notions from network science domain.

2.10.2 Definitions: Node degree, clustering coefficient and path length

2.10.2.1 Node degree

The "degree of a node" in a given network, is defined as the number of connecting links (edges) that a node has. In Fig. 2.10, the node "A" has a degree of 4.

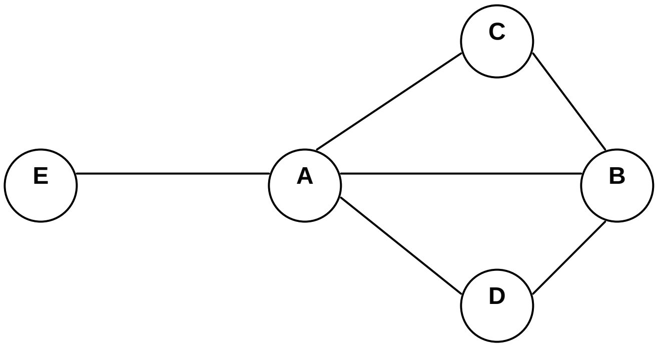


Fig. 2.10 Network science concepts

2.10.2.2 Clustering coefficient

Clustering coefficient is a characteristic of a network topology. We use the term "clustering coefficient" to indicate the measure of the tendency of the nodes in the network (graph) to cluster together. A number of clustering coefficient metrics has been defined in the literature. We will consider the "average clustering coefficient" metric as defined by [119] and explained in [120]. From henceforth, by the term clustering coefficient, we will mean average clustering coefficient.

This clustering coefficient is defined as the average of all the "local clustering coefficients" for the entire given network. The local clustering coefficient is the quantification of how close the neighbors of this given node are to being a complete graph (i.e. where all nodes are directly connected to each other). Local clustering coefficient is defined as follows.

Using formal graph theory notations, we say that a graph/network G is defined as a set $G = (V,E)$, which consists of a set of vertices/nodes V and a set of edges/links E connecting them, and a particular edge e_{ij} connects vertex/node i with vertex/node j. For a directed graph, the edges e_{ij} and e_{ji} are distinct from each other and for an undirected graph, they are identical. In all our examples, for simplicity we will only consider undirected graphs/networks. Suppose the set N_i denotes all the nodes that are directly connected to the node i and E_i is the set of all edges between these nodes in set N_i. The total number of nodes in the set N_i is denoted as $|N_i|$ and $|N_i| = k_i$, where k_i is the degree of the node i. Maximum number of edges that are necessary to connect all these $|N_i|$ nodes together to form a complete graph is $k_i(k_i\text{-}1)$. We denote the set of all possible edges as the set E_i and $|E_i| = k_i(k_i\text{-}1)$. Hence for a directed graph/network the local clustering coefficient C_i of node i, is defined as follows:

$$C_i = \frac{|\{e_{jk}\}|}{k_i(k_i - 1)} : j,k \in N_i,\ e_{jk} \in E_i \tag{2.1}$$

For an undirected graph/network the local clustering coefficient C_i of node i, is defined as:

$$C_i = \frac{|\{e_{jk}\}|}{\frac{k_i(k_i - 1)}{2}} : j,k \in N_i,\ e_{jk} \in E_i \tag{2.2}$$

For example, the local clustering coefficient C_A of node A in Fig. 2.10 is $2/({}^4C_2) = 2/6 = 0.33$.

Using the above definitions, the average clustering coefficient C of a network/graph is defined as:

$$C = \frac{1}{n}\sum_{i=0}^{n} C_i : i \in V \qquad (2.3)$$

2.10.2.3 Path length

The distance between two nodes is defined as the number of edges in the shortest path between these two nodes. In Fig. 2.10, the path length between node B and node E is 2. The average path length in a network/graph is defined as the average of all these distances for all possible pairs of nodes. Average path length is another characteristic of a network topology.

2.10.3 Watt and Strogatz topology generation model

A part of the research reported in this dissertation is based on the Watt and Strogatz topology generation model [119]. Hence we discuss this model and all the related and relevant insights. This model is based on the notion that a network can be represented in a *D*-dimensional space having *D* degrees of freedom. This model is explained below. “Degrees of freedom” is a different notion than node degree as used in graph theory, but we can say that for every two node degrees there is a single degree of freedom, i.e., $D \equiv \lceil k/2 \rceil$, where k is the node degree.

2.10.3.1 Lattice network in D-dimensional space

Fig. 2.11 shows a representation of 2-dimensional lattice network in a 2-dimensional space. In a lattice network, the nodes are connected to immediate k neighboring nodes, where k is the degree of the nodes (in Fig. 2.11 we have shown uniform degree, k = 4) and neighbors are those nodes that are closest to a given node. This means that the neighbors are those k nodes for which the distance between the given node and all other nodes are minimum. We can see that, a lattice network is an ordered network. This ordering does not mean that nodes should have uniform distance between them, but it means that the links only connect the closest pair of nodes. Fig. 2.11 should not be taken as the general case. This figure actually showed a special case where links not only connected to closest nodes, but all nodes also had uniform distance with all 4 neighbors.

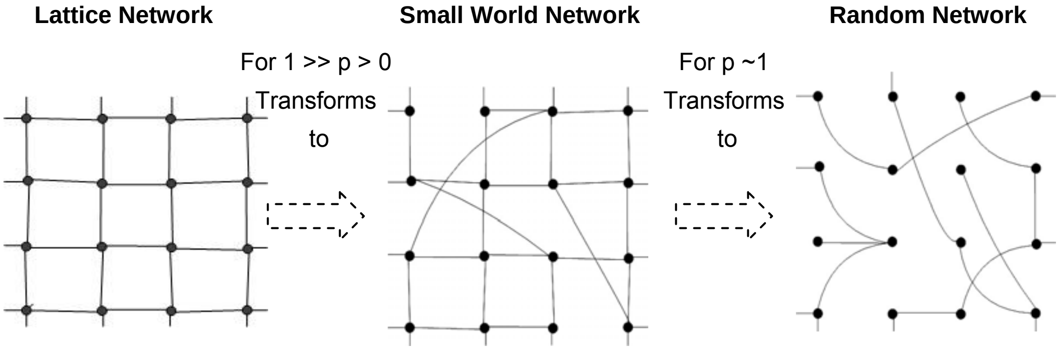


Fig. 2.11 Lattice, small world and random network topologies

In Fig. 2.11, the distance metric is the geometric distance between nodes based on their two dimensional co-ordinate positions in the 2-dimensional space. Similarly, we can consider a generalized *D*-dimensional space with *D* degrees of freedom. We can further extend this to a more general model (and more complicated space), where *k* is not equal for all nodes, but indeterministic having a given probability distribution. That kind of topology in a complex geometric space is difficult to visualize but possible to conceive and analyze using graph theoretic methods.

2.10.3.2 Generation process

This Watt and Strogatz model generates three kinds of network topologies: lattice, small world and random networks, based on a single parameter *p*. The generation begins with a lattice network. Watt and Strogatz [119] had explained their model for a *D*-dimensional lattice network connected in torus fashion (i.e. the two ends of a lattice are connected back to each other and this is done for all dimension). However, in Fig. 2.11, we show the process for a 2-dimensional lattice network, which is not connected as a torus. In the generation process, a certain *p* fraction of links of all the nodes in the lattice network are disconnected from their immediate neighbors and connected to a distant node chosen at random. So this means a *p* fraction of short links/edges between neighbors are replaced by long links/edges between distant nodes. By varying the generation parameter *p* we can create a wide variety of network topologies. When *p* is very small value <<1 or ~0, the topology that is generated is still a lattice network because not many ordered links are disturbed. Whereas for $p = 1$ or a value near 1, the topology that is generated is a random network [117]. This is because, each node is connected to any random node irrespective to the distance between them.

What is interesting here is the clustering coefficient and average path length properties of all these networks that are generated by this process. When we plot the normalized clustering coefficient $C(p)$ and average path length $L(p)$ against the generation parameter *p*, we get a trajectory as shown in Fig. 2.12. This plot was obtained from [119]. In this plot the clustering coefficient and average path lengths are normalized by clustering coefficient $C(0)$ and average path length $L(0)$ at $p = 0$, respectively and was used as the vertical axis values. The zones for lattice and random networks are also shown on this plot.

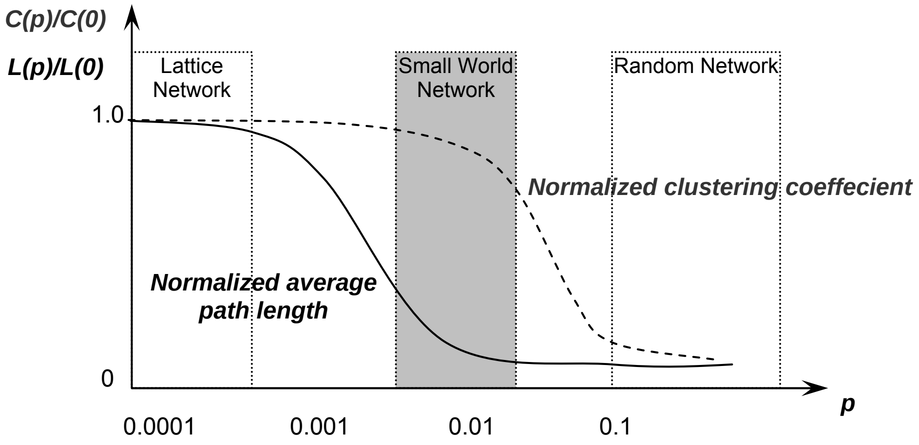


Fig. 2.12 Normalized clustering coefficient and average path lengths for different *p*

In between the two extremes of *p* = 0 and 1, there lies a region where the network topologies have high clustering coefficient, which is similar to that of the lattice network, and yet the network has a small average path length similar to that of a random network. This class of networks that have this property is called as "Small World Network" by [119].

2.10.4 Greedy routing

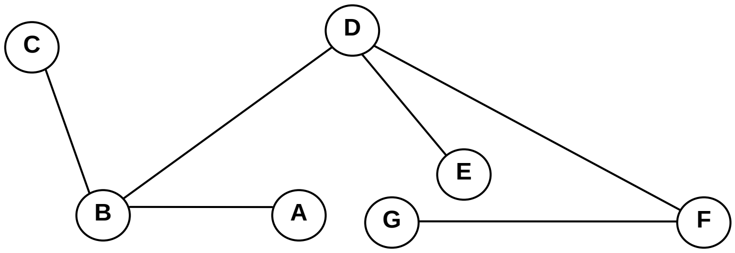
2.10.4.1 Motivation

To deliver a message to a particular node, whose address is unknown to the sender, the sender would prefer to hand it over to any arbitrary node, which will ensure its delivery to the intended destination. This mode of operation is convenient because a node can be selected at random to become the entry point in the network. In addition, this can spread the message processing load to multiple nodes if a proper network topology is adopted to implement the network.

Messages could be delivered in a single hop in a network topology where all nodes are connected to each other. However with large number of nodes, this kind of topology become impractical as it is not possible for a single node to maintain connections to all other nodes. This is specifically difficult when nodes get added to the network in an organic fashion. So we should choose topologies where all nodes are not connected with each other.

A network where all nodes are not connected to each other, a message can be delivered by flooding the network with copies of a message. In this method, each node broadcasts a message that it receives to all its immediate neighbors. However this strategy overloads the network with unnecessary copies of the same message. A better alternative is where each node selectively routes a message through a path that takes the message to the intended destination. However there are challenges in this selective routing.

Selective routing in an arbitrary random network requires a pre-determined map of the shortest path which a message should take, so that the search time is minimal. To allow injection of the query/message in any arbitrary node and yet successfully route the query to the destination, we need to construct and maintain a large number of explicit route maps at every node. For example, in the random network as shown in Fig. 2.13, for a node B, the required list of route maps are also shown in the same figure.



Route maps at node B:

To A: A
To C: C
To D: D
To E: D→E
To F: D→F
To G: D→F→G

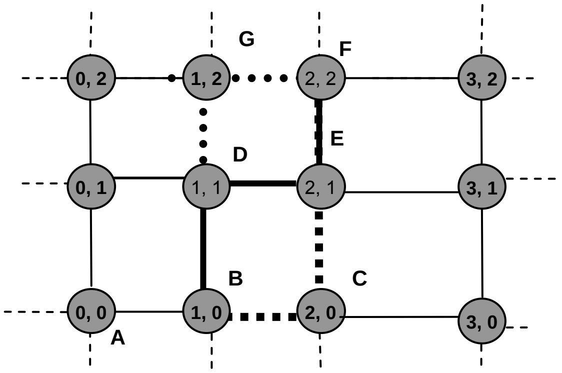
Fig. 2.13 Need for explicit route maps in a random network

Thus for every node, there has to be a list of *N*-1 route map entries when there are *N* nodes in the network. Identification of all these routes will require that each node has the global knowledge of the entire network topology (i.e. how all other nodes are connected to each other). Identifying and maintaining such elaborate route maps is expensive. If we use ordered lattice network and adopt greedy routing method, we can avoid this need for explicit maps [118], [121], [122]. In our proposed semantic routed network we adopt a modified form of greedy routing because it is simple and elegant.

2.10.4.2 Basic mechanism

In this greedy routing method, the address of each node in the lattice network is expressed using a *D*-dimensional co-ordinate system [123]. In this co-ordinate system, a numeric identifier is assigned to each node, based on its position on a particular dimension. Here a node's position is considered as its address. The address (or position) assignment of the nodes in a 2-dimensional lattice network, is shown in Fig. 2.14. Identifiers are assigned in ascending order starting from one node and traveling in one direction on a particular dimension. The lattice network may be connected in a torus fashion (though not shown in this figure).

In the greedy routing method, a node decides only the next immediate destination of the message in the current hop, and the next hop is independently decided by the next node and so on. By this manner a message is progressively routed to the final destination. In Fig. 2.14, message routing from node B to F is shown in a 2-dimensional lattice network. Here three alternate shortest paths are possible all of which takes 3 routing hops to reach the final destination. These alternate paths are: (1) B→D→E→F, (2) B→D→G→F, (3) B→C→E→F. Here each node decides the next hop node depending on which neighboring node is closest to the destination's position/address. To determine the distance between all the neighboring nodes and the destination node, the Euclidean distance using the *D*-dimensional co-ordinate system is used. Each node maintains a routing table that has position (address) of the neighboring nodes in terms of the *D*-dimensional co-ordinate system and the corresponding outgoing link. This positions/addresses act as the routing table lookup key and outgoing link identifiers act the next hop route direction. Here the link identifiers act as the substrate mechanism over which the network is overlaid.



Message containing destination's address

(2,2)

Routing Table of node **D**

Key	Link	Dist	Dot prod
(0, 1)	Left	2.23	0.71
(1, 0)	Down	2.23	0.71
(2, 1)	Right	1	0.47
(1, 2)	Up	1	0.47

Fig. 2.14 Greedy routing in lattice networks

When a message arrives, the Euclidean distance between the message's destination address and all the key addresses in the routing table are computed. Based on these distances one particular routing table row is chosen whose key address is closest to the message's destination. The position in the *D*-dimensional space is also a position vector and therefore instead of Euclidean distance, the dot product of the two position vectors may be used to decide the route. For the above example in Fig. 2.14, the routing table of node D and the message's destination (node F) is also shown in the figure. For the given message and destination, the Euclidean distances and vector dot products of the message's destination and node D's neighboring node's address/position is also shown in Fig. 2.14, at the right side of the routing table. The destinations addresses are next hop are mentioned in terms of the link directions (e.g. "Up", "Down", etc.). They can be also mentioned in terms of address of the nodes they connect to from the current node D. This addressing should use the underlying network's addressing scheme.

For the message routing example, as shown in Fig. 2.14, both proximity schemes- the Euclidean distance (shown as “Dist” in figure) or vector dot product (shown as “Dot prod”) measure indicated that, among all neighboring nodes of node D, only the node G or E is the closest to the final destination node F. Hence the final destination next hop destination from node D should be either node E or G.

This illustrates that though greedy routing does not require elaborate pre-determined route maps but it uses a simple rule to take routing decision at each node in the traversal path based on the very limited knowledge of only the immediate local network (i.e. the given node is connected to which all other immediate nodes). The routing table lookup mechanism, as presented above, constitutes the routing rule. This rule is simple and can be executed in a distributed fashion. This obviates the need for voluminous storage of all the shortest routes between all possible pair of nodes.

2.10.4.3 Greedy routing problems in random network

Greedy routing can not successfully deliver messages in a random network, even though routes may exist in the network. This notion is illustrated using a simple example in Fig. 2.15. Here the positions of the nodes in the given space represent their addresses. In this example, an attempt is made to route a message from node A to the final destination node G. Using greedy routing, the node A will determine that the next neighboring destination should be node B. This is because there is only one entry in node A’s routing table, which is for node B. Therefore a routing table lookup using any message destination address will yield only one route, i.e. to node B. Once the message reaches node B, that node will determine that the next destination node should be node A. This is because, node B’s routing table has three entries, belonging to node C, D and A, out of which the nearest node to the message destination’s address G is node A (refer Fig. 2.15). Hence this message will be forwarded to node A. This returning to node A, creates a loop and the message would keep on circulating within this loop and would be unable to reach the destination.

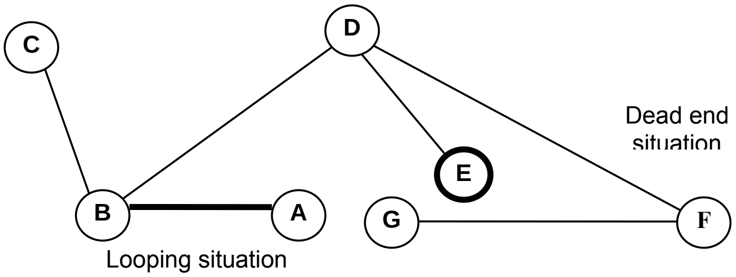


Fig. 2.15 Greedy routing is unable to route messages in a random network

Suppose the greedy routing method is fitted with a loop detection mechanism, where a node, which the message has previously visited, is never chosen as the next hop destination, then the message is routed from node B to node D. This is because, as node A is excluded by the loop detection rule, so the next nearest destination in node B's routing table is node D. Once the message reaches node D, the greedy routing mechanism will route it to node E, because node E is closer to node G, compared to node F. There is no other egress route from node E, except the one through node D, and the loop detection mechanism will not allow back tracing from node E to node D. Therefore the message effectively gets trapped in the dead end at node E and fails to reach the destination. Even though a route from node D to node G existed through node F, but the greedy routing method at node D did not use that route because the simple greedy routing rules did not enable identification of that route. This example shows that why greedy routing is not useful in random networks. On the other hand, greedy routing works well in lattice networks because it is an ordered network where by using the general sense of direction a node is able to identify the viable routes. This is also true for small world network topology. We explain this in the next section.

2.10.4.4 Role of short links in greedy routing

Sufficient number of short links between neighboring nodes will avoid the problems related to greedy routing in a random network. This insight is explained in Fig. 2.16.

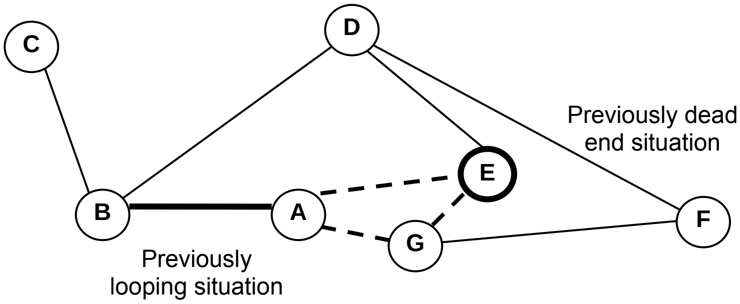


Fig. 2.16 Role of short links in greedy routing

This figure is similar to the previous one in Fig. 2.15, but there are three additional short links between neighboring nodes A, G and E. These short links are shown by broken lines. Presence of any one of these three short links could have solved the above mentioned routing problem. A random network which has sufficient number of short links connecting immediate neighbors is a lattice network having random long distance links. This network is a small world network because it has high clustering coefficient as lattice network and low average path length as a random network. This shows that greedy routing will perform better in a small world network, where the average path lengths will be small similar to random network, thus the routing (search) will successfully terminate within small number of steps (routing hops). As short links leads to high average clustering coefficient, hence high clustering coefficient in small world network is an indication of its ability to successfully route a message, whereas low path length indicates that a small number of routing (search) steps will be needed.

2.10.4.5 Greedy routing in small world network

Greedy routing works satisfactorily in a small world network, only under some specific conditions. Performance of greedy routing in small world networks has been investigated by [118], [121] and [122]. It was found that greedy routing works fine only when the random links in the small world network follows a specific pattern of connections. It is required that random long distance links should have a certain kind of probability distribution as a function of the distance between the nodes that the random link connects. Greedy routing works only when this probability distribution $P(x,y)$ of a link between two nodes x and y, is a D-harmonic distribution [121]. That is when, probability $P(x,y)$ is proportional to $1/H{\cdot}s(x,y)^D$, where $s(x,y)$ is the Manhattan distance [124] between node x and y in the D-dimensional space and H is the normalizing factor. Generating topology using this rule is difficult as the generation process will require a global knowledge of all nodes and their positions (addresses) in a network to determine the required D-harmonic probability distribution function. Networks which grow organically with time and have no known growth pattern, will pose a problem as the necessary global knowledge can never be predetermined. So we need to explore alternative avenues on how to make greedy routing work in any general small world network, which may not satisfy the D-harmonic distribution criterion.

Analysis in [122] implies that if we retrofit greedy routing algorithm with an explicit mechanism that identifies the shortest paths and explicitly program these routes in the routing tables, then we can still enjoy the benefits of greedy routing and yet successfully route a message in a small world network. This modified greedy routing is called as “indirect-greedy routing” in [122]. The benefit of greedy routing is that no route maps need to be generated and maintained, whereas in random network we have to generate and maintain all possible shortest paths. Indirect-greedy routing is a compromise between these two extremes, where only a very few shortest path routes need to be identified and maintained. The extent of the need to maintain shortest paths will depend on the ordered property of the small world network. In other words, we can say that the indirect-greedy routing method will need to identify and program a small number of pre-determined shortest path routes, as long as the small world network has high amount of ordering and have high clustering coefficient which is similar to the pure lattice network. The ordering and clustering coefficient can be preserved when the topology generation parameter *p* in the Watts and Strogatz model is kept very small.

2.10.4.6 Notion of search path length

We have seen that greedy routing does not always lead to a viable and shortest route in a general network. During search, the length of the actual path taken may be greater than the shortest path, and the average of all such search paths is defined as the "average search path length" in a given network for a given routing method. Only for lattice networks with greedy routing, the average search path length is same as the average path length as defined in section 2.10.2. But for any other kind of networks, the average search path length is greater than the average path length. This means that average search length is either equal or greater than average path length for any network topology for any kind of routing.

The average search path length in a lattice network has an order of $O(N^{1/D})$, where N = number of nodes, and D = dimension of the lattice network which is $\equiv \lceil k/2 \rceil$, where k is the uniform deterministic node degree of the lattice. The average search path length for a small world network with D-harmonic link distribution is in order of $O((1/l)(\log_k N)^2)$ [122], where l is the number of long distance links per node. Whereas, the search path length for an arbitrary small world network which does not satisfy the D-harmonic link distribution criterion, is in order of

$O(N^{\alpha})$, where α is constant. On the other hand, the average search path length for a small world network with indirect-greedy routing is in order of $O((1/l^{1/D})(\log_k N)^{1+1/D})$ [122].

2.10.4.7 Notion of routing table size

The notion of routing table size is also important in this context of routing. To enable successful routing in a random network, at every node we need to maintain the route map to every other node. This means all nodes have to keep route information about rest N-1 nodes (refer Fig. 2.13 in section 2.10.4.1). This constitutes the database which helps routing and can be thought as the routing table mechanism for a random network. Thus the order of this routing table size is $O(N)$.

For lattice network using greedy routing method, each node have to maintain only the table of k immediate neighboring nodes, therefore the order of routing table size is $O(k)$. For small world network using indirect-greedy routing, according to [122], each node need to have know $O((\log N)^2)$ other long distance links in its vicinity in addition to short link related information about k immediate neighbors. Therefore we expect that the nodes need to maintain $O(k+((\log N)^2)$ entries in their routing tables.

2.10.4.8 Lessons

The following lessons are useful in regards to adoption of small world network for constructing a meaning based message routing network and index system:

1. Greedy routing is a simple and attractive method because it reduces the routing table size at each node. But greedy routing requires that the network should have lattice like ordering.

2. Indirect-greedy routing can lead to much shorter search path lengths (and smaller search response time) in small world network compared to lattice networks. To implement indirect-greedy routing, there is a need for additional mechanisms to identify the shortest paths and explicitly program these routes in the routing tables.
3. Indirect-greedy routing is viable and attractive only when the small world network has lattice like properties, i.e., when number of long distance routes is small compared to shortest distance links between immediate neighbors. In this case only a few route maps have to be maintained.
4. Small world network has the lattice like properties when the Watt and Strogatz generation parameter *p* is small enough to maintain the clustering coefficient to a high value similar to that of the lattice network.

2.11 Queuing theory

The load sharing technique of deploying multiple replicas of index servers is a standard solution pattern to scale up the capacity of a system. Here we examine the theory behind this technique.

2.11.1 Definitions: Execution time, waiting time, response time

Concepts from two domains: real time systems and queuing theory (operations research), are useful to analyze and design the index system. Here we explain the equivalence between the concepts from these two domains. The notion of “task” which is waiting to be executed, as portrayed in real time system domain, is equivalent to the notion of an “entity” waiting to be serviced, in queuing theory.

Execution time is the time needed to execute a task. This is also known as service time in queuing theory terminology. Waiting time is the time a task (or an entity) waits before its execution (or servicing) begins. Response time is the difference of time when a task was released and the time when its execution was completed, i.e. total time a task spent in a queue plus the time spent in execution. Hence we can say that the response time is the sum of waiting time and execution time. This response time is also called as the "flow time" in operations research.

2.11.2 Basic queuing model and response time statistics

In any network device (say router), messages arrive at a rate r_{arr} per unit time, get queued up and wait for processing. The queue gets serviced by *n* concurrent message processors, each of which completes processing a message in time t_{serv} (Fig. 2.17). When a large number of messages arrive within a short time, the queue momentarily builds up. If the long term average service rate is greater than the long term average arrival rate, then the queue will eventually get serviced and the queue length will drop. Instantaneous arrival rate $r_{arr,}$ servicing time t_{serv}, response time t_{resp}, and queue length *l* are random variables.

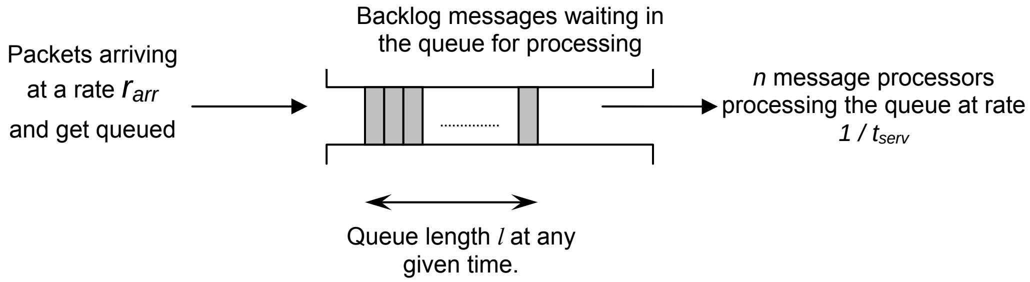

Backlog messages waiting in the queue for processing
Packets arriving at a rate r_{arr} and get queued
n message processors processing the queue at rate $1 / t_{serv}$
Queue length *l* at any given time.

Fig. 2.17 Queuing model

This kind of queue system is denoted by the "A/B/*n*" queue system notation, where A denotes the arrival model, B the service model and *n* denotes number of concurrent servers/processors.

Generally in networks, r_{arr} has been observed to have Poisson distribution [125] with long term average arrival rate λ and t_{serv} is known to have Exponential distribution [126] with long term average serviced time t (i.e. the variable $1/t_{serv}$ have Poisson distribution with long term average service rate $1/t$). This type of system is called M/M/n queue according to Kendall's notation [127], where A = M denotes the Poisson arrival process, B = M denotes the Exponential distribution of the service time.

The approximate average waiting times for this kind of system have been analytically computed and available as standard waiting time tables [128] for a given value of n and the product λ^*t, which is the long term utilization ratio for the processors. For example, in Table 2.1, when the utilization value is 0.3 and there are 4 processors, the waiting time is 0.0132 times of the average execution time t. Therefore, in this example, the average response time = $(1 + 0.0132)^*t$. This also means only a small fraction of messages will have to wait in the queue and most of them (~96.29%) will be immediately processed upon arrival.

Table 2.1 Waiting times for a M/M/n queue [128]

Processor utilization λ*t	**Number of processors (servers)**							
	1		**2**		**4**		**8**	
	W	***P***	***W***	***P***	***W***	***P***	***W***	***P***
0.1	0.1111	0.9	0.0101	0.9818	0.0002	0.9992	0	0.9999
0.3	0.4286	0.7	0.0989	0.8615	**0.0132**	**0.9629**	0.0006	0.9964
0.40	0.6666	0.6	0.1905	0.7714	0.0377	0.9093	0.0039	0.9815

2.11.3 Useful lessons and application note

From the queuing model and wait time statistics tables we assimilate the following insights:

1. The lower bound of the average response time is the average execution time t.

2. To get smaller average response time, we have to use larger number of concurrent processors, and/or processor with larger average service rate 1/*t* so that the utilization rate is smaller than 1. To ensure a small average response time, we choose appropriate *n* for a given λ and available processor with a given *t*. Therefore, how many processors should be employed is to be decided based upon the message arrival rate statistics and service rate statistics of the available processors.

3. The queuing model incorporates both response time and throughput capacity factors. When an appropriate number of processors *n* is chosen, the average response time is in order of the average execution time. This indicates that the system will have sufficient capacity to service tasks that arrive with the given arrival rate statistics. In other words, the throughput capacity and the response time of the designed system will be sufficient if the single parameter *n* is chosen properly.

4. The two steps of designing the M/M/n system are as follows. First we design a processor to minimize the average service time *t*. Then for a given arrival rate λ, we choose the minimum number of processor *n*, that is necessary to keep the utilization rate λ**t* of each processor < 1 and the average wait time *W* within the acceptable limit. As $W = k_n * t$, where k_n is a factor given by the M/M/n wait time table, so we choose *n* such that k_n is small. In the example shown above this $k_n = 0.0132$ for a $n = 4$. As response time = $(W+t) = (1+ k_n)*t$, therefore the processor design should be carried out properly to ensure that *t* is far less than the acceptable response time, so that a feasible M/M/n system can be designed to achieve the desired system response time and throughput capacity. This also means that being able to design processors with very small service time *t* leads to savings in number of processors *n*.

5. For a well provisioned system, the execution time and response times are very similar, i.e. response time = (1 + k_{n})*$t \approx t$, because the number of processors n is chosen such way that $k_{\mathrm{n}} << 1$. In many place in this thesis we have considered this assumption.

2.12 Bloom Filter basics

Since we will use a Bloom Filter in this research, hence its fundamental principles are discussed here. A Bloom Filter (BF) is a compact representation of a set [129]. It consists of a large single dimensional array of m bits generated using k hash functions (Fig. 2.18). To insert an element (a number, say "id_{i}") into this set, we hash this element using k different hash functions to generate k different index values having a range 0 to (m – 1) = (2^q-1), where q = number of bits in the k index values. These values decide which bits in the m-bit array should be set to 1. To test whether an arbitrary element is in the BF, we generate its k bit indices (using the same hash functions), and check whether all of those bits are 1. The condition that all bits are 1, indicate that the element is present in the set.

This kind of membership testing does not yield any false negatives (i.e., element is present but test will show negative), but will yield some false positives (i.e., element is absent and yet test will indicate its presence). Probability of false positives can be made smaller by proper choice of a large m and optimum k, for a given maximum element holding capacity n, where n is number of element that will be accommodated in the BF. The m bit array is implemented by a bit addressable memory. Membership test is performed by checking whether all k given bit locations (obtained by k hashing elements say "e_{i}"), contains 1 or not. This test can be implemented by testing the conjunction of all the bits using an AND gate or passing all the bits through a sequence detector to detect k consecutive ones.

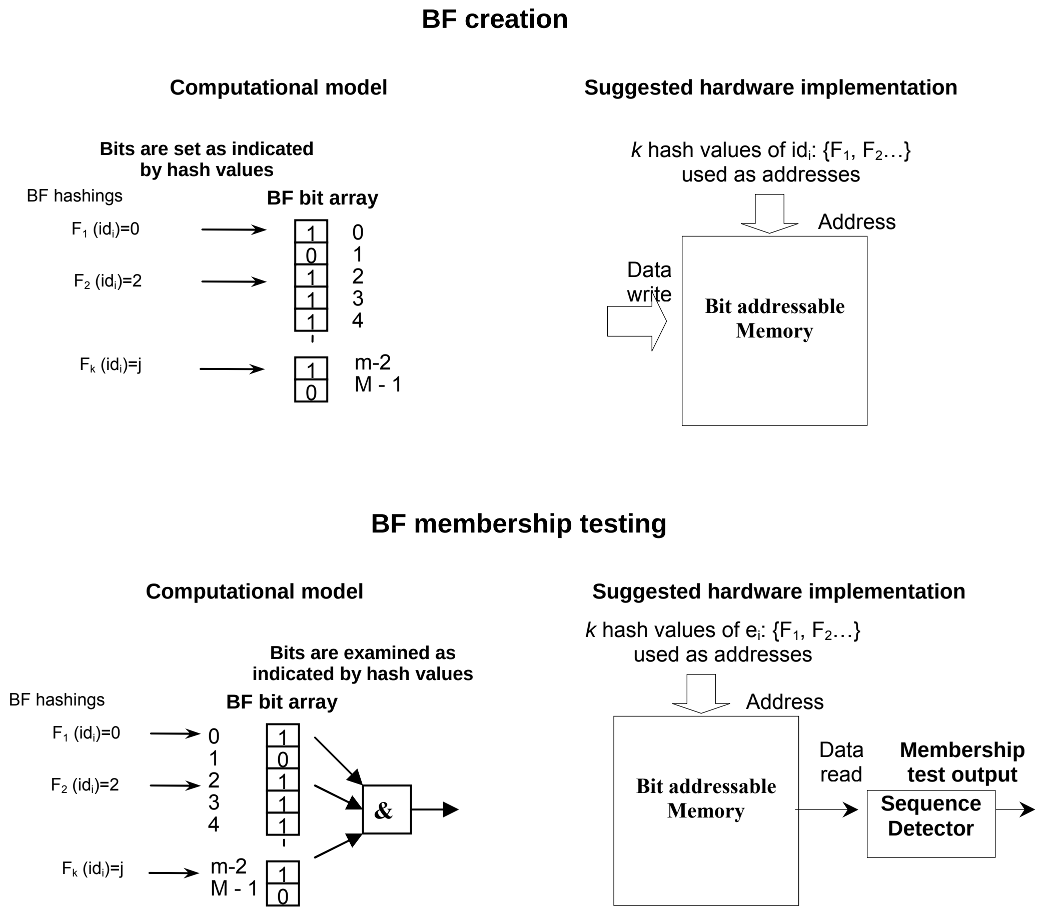


Fig. 2.18 Bloom Filter basics

Two different designs of this basis BF are possible. In one form, the *m* bit array is considered as one single array, while in the other one, the *m* bit array is partitioned in equal *k* partitions and considered as *k* separate arrays. In the partitioned version of the BF, each of the *k* bit address generated from hashing is mapped to separate partitions. For the un-partitioned case the membership testing requires *k* memory accesses. So it takes O(*k*) order of time assuming use of a single port memory. Whereas for partitioned BF, the *k* memory partitions can be implemented on separate memory banks that can be accessed in parallel, reducing the testing time to O(1).

For a given BF bit array size *m*, and number of elements *n*, there exists an optimum number of hash functions *k,* that should be used. This optimum *k* minimizes the false positive probability during membership testing. This optimum *k* is given as:

$$k = \frac{m}{n}\ln 2 \tag{2.4}$$

With this optimum *k*, the probability of false positive for a BF for a given *m* bits and *n* elements, is approximately given by a simple function [129] as follows:

When unpartitioned BF is used, then

$$P_{false+ve} = \left(1-\left(1-{}^{1}\!/_{m}\right)^{kn}\right)^{k} \tag{2.5}$$

When partitioned BF is used, then this is

$$P_{false+ve} = \left(1-\left(1-{}^{k}\!/_{m}\right)^{n}\right)^{k} \tag{2.6}$$

For example, for m = 131,072, k = 7, and $n = 10^4$, the probability of false positives are: 0.002076 for the unpartitioned BF, and 0.002077 for the partitioned BF. This shows that the probabilities are quite similar when they are small. The above equations are good approximations as long as $k{\cdot}n << m$ and the hash functions are collision resistant.

2.13 Theory of hash functions

For proper operation of the Bloom Filter, we need high quality, minimal collision (i.e. collision resistant) hash functions that satisfy the “strict avalanche criterion” [130]. This criterion tells that each bit in the hash value should change with a probability of 0.5 whenever a single bit of the input bit string changes. This quality ensures that the hash values are equally distributed across the entire range, hence minimize collisions.

2.14 Summary

We need to understand how human mind actually comprehend meanings. This is necessary to design an effective meaning representation data structure and comparison technique. Similarly to devise a network which can forward messages based on meaning, we need to know network science theories as well as the pros and cons of various networks which had been employed in past for distributed searching. In this chapter we analyzed relevant literatures to get a better understanding on: the cognitive processes that are related to meaning interpretation; strengths and weaknesses of the existing meaning representations; useful natural language processing (NLP) techniques; index organization; index scaling techniques; distributed search using P2P networks and useful network science insights.

Existing meaning representation techniques used in the information retrieval domain do not support meaning composition, thus they can not adequately represent composite meanings for the purpose of comparison. Therefore we need a new meaning representation design, which is psychologically realistic. We realized that hierarchical unordered collections of elementary meanings may be a good candidate to represent composite meanings, because it has support from multiple cognitive science and linguistic theories. However to build such hierarchical collection structure for a given natural language text, we need to first interpret the text. Fortunately, several useful NLP technologies are available which can be used to extract the most essential meaning from a text document and help building the structure.

We also realized that the P2P networks can be used as distributed indexes, as they distribute the search (or index lookup) task load across multiple nodes, thereby improving scalability. However P2P networking schemes have to be enhanced using network science insights to improve its search success rate and response time. This requires adoption of appropriate query routing methods. Load sharing technique, which is based on queuing theory is used in existing distributed index designs. This is a good and proven mechanism, which should be adopted and adapted to further improve scalability of a P2P network based index system.

CHAPTER III

MEANING REPRESENTATION AND COMPARISON MODEL

Systematic organization of the index entries in an index system can save number of servers necessary to build the index system. However such systematic organization of the index system requires an overlay network appliance called semantic routers. These overlay networking routers forward search queries to the particular index server which is most likely to contain matching index entries. This forwarding of destination is decided by ascertaining how closely the meaning of query matches the meaning of the document whose index entry is stored in a particular index server. To support such decisions we need a design of a descriptor that will represent meanings of documents, queries and enable their comparisons. In this chapter we present the design of such descriptor and a comparison technique. This descriptor design supports generative meaning composition and thus can represent and compare composite meanings as explained in Chapter I. We begin by specifying the requirements of a desired meaning representation scheme followed by a description of role of the meaning comparison in the proposed index system and the proposed meaning comparison scheme.

3.1 Requirements for the descriptor

Based on the understanding gained in Chapter II, we can assert that the proposed semantic descriptor design and the comparison technique should satisfy the following requirements:

(A) *The descriptor should be able to express complex concepts (or meanings).* This is a key requirement because this compositionality aspect is fundamental in meaning processing and meaning comprehension. The relevance, rationale and evidences to support this requirement had been presented and discussed in the previous chapter.

(B) *The descriptor design should be coherent with human cognitive processes* and supported by understandings from cognitive sciences. This requirement is needed to ensure realistic data structures to serve the needs of practical applications.

(C) *The meaning in a text should be reasonable expressed by the proposed descriptor*.

(D) *The semantic similarity comparison should be a self contained process* and the descriptor should be sufficiently descriptive to preclude the need of additional information to disambiguate meaning or to enable recognition of meaning subsumption, during the meaning comparison operation.

(E) *Descriptor data structure should be compact enough for efficient storage in the index and transmission as message payloads*.

(F) *The similarity computation technique should be computationally efficient* for faster comparison during index operations (lookups, additions/deletions, etc.).

3.2 Role of SRN in the proposed distributed index

To appreciate the design of the meaning comparison scheme, we need to first understand its role in the index system. Therefore, in this section we present an overview of the proposed index system which will use this meaning comparison scheme. We begin by explaining the data entities that are used in this index system, then describe the role of the key system components and finally identify the components which will carry out the meaning comparison operation.

3.2.1 Documents, URLs, index entries, document ids and storage

For basic search operation, instead of the web page or documents, only their URLs need to be stored in the search engine. When users search for a web page or a document, the URLs would be returned from the index. This index system which maps meanings to URLs, has a two-tier mapping system very similar to the one proposed by [4]. In the first tier, the meaning is mapped to a unique document identifier, in the second tier the document id is mapped to its URL.

The first tier mapping entries are called index entries, each of which consists of two entities: (1) the document identifier (called "document id") of the object being searched; and (2) the meaning representation (semantic descriptor) of the web page or the document. These index entries are stored in the index servers in form of inverted index as described in section 2.9.1. Whereas, the second tier mappings between document ids and URLs are stored in the document servers. Each second tier mapping entry consists of document id and document URL, and all these mapping entries are stored in the document server.

In this system, when a user makes a search request for an object (URL of the web page or document), actually a search is made for the object's index entry in the index server. When the index entry is found, its document id is retrieved and sent to the document server. For a given document id, the document server retrieves the corresponding URL and presents it to the user.

3.2.2 Distributed search engine components

We proposed that, to achieve scalability, the index should be deployed as a distributed system of large number of small index server nodes, each specializing in specific domain areas. For example, index entries related to biological sciences can be stored in one index server node, whereas entries related to sports news can be in another.

In this proposed system, all these index servers will be interconnected by a special network called Semantic Routed Network (SRN) (as in Fig. 3.1). To search and retrieve index entries, these specialized index nodes may apply custom search strategies that are suited to the domain, search context, type of data objects, etc., to yield superior search performance. Each index node may be deployed as a single computer/server or a pool of servers to achieve scalability. A single pool or index node may host index entries on multiple subject categories to utilize available spare storage capacity. In that case, this single physical location will be represented as multiple destinations from SRN's viewpoint. This allocation of stored physical space to multiple subject matter categories is premeditated, not random and unaccounted as in the case shown in Fig. 1.2, Chapter I. In that system, the index entries are randomly distributed across all index server pools.

When number of index server pools is small say ~1000, one single semantic router (as in Fig. 3.1) may be used to implement the entire SRN. The other alternative is to use multiple smaller routers to implement the SRN. These alternative methods will be discussed in Chapter VI.

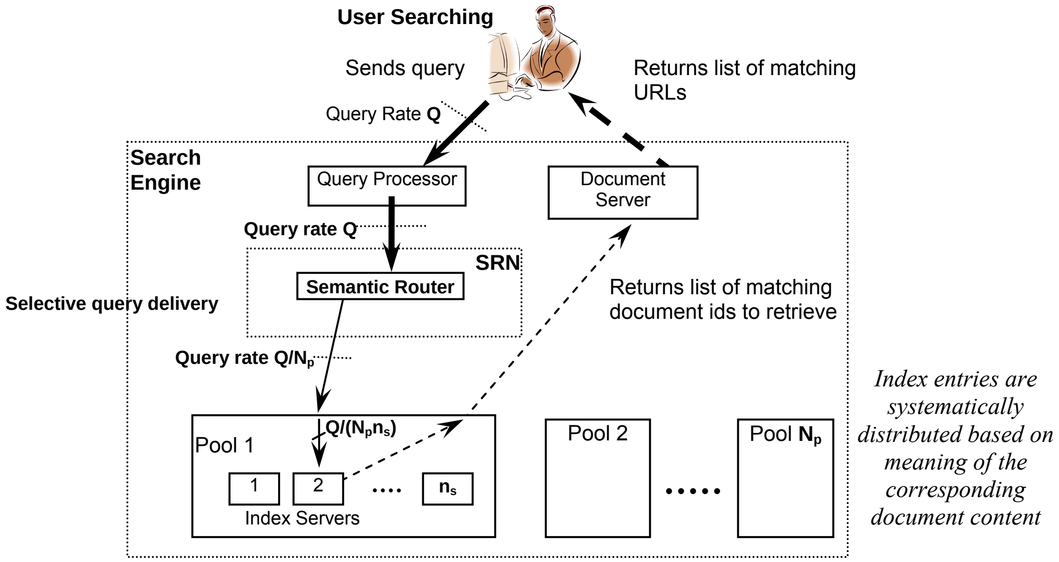


Fig. 3.1 Role of Semantic Routed Network and semantic routers in query delivery

3.2.3 Components that carry out meaning comparison

Meaning comparison operations are carried out in the semantic routers. The semantic routers use this comparison operation to decide message forwarding routes. In addition, the index servers may also use this comparison operation. In a typical search engine, the inverted index operation, which is carried out inside an index server, is based on a vector model (either term based or latent semantic indexing models). Therefore inverted index operation alone does not give good meaning comparisons and discerning ability. If index servers also use the proposed meaning comparison operation in addition to the inverted index operation, then search performance can be improved. This hybrid scheme of meaning comparison will operate as follows.

First a consideration set of index entries are identified using the inverted index system, then the semantic descriptors of all the considered index entries are compared against the query descriptor using the proposed meaning comparison scheme. Only the top few index entries whose comparison values are high enough will be sent to the document server for second tier mapping resolution. Inverted index mechanism is best suited to pinpoint suspected matches from a large collection, whereas the proposed mechanism can do a better job in identifying relevant objects from the set suspects generated by inverted index mechanism.

3.3 Overview of the proposed meaning comparison process

Here we present an overview of the processes in a distributed search engine that are necessary for operation of the proposed meaning representation and comparison scheme. This will help appreciating the design of this proposed scheme. This scheme can be retrofitted to the existing inverted index based distributed search engine.

Fig. 3.2 presents an overview of the information processing flows involved in the search process using this scheme.

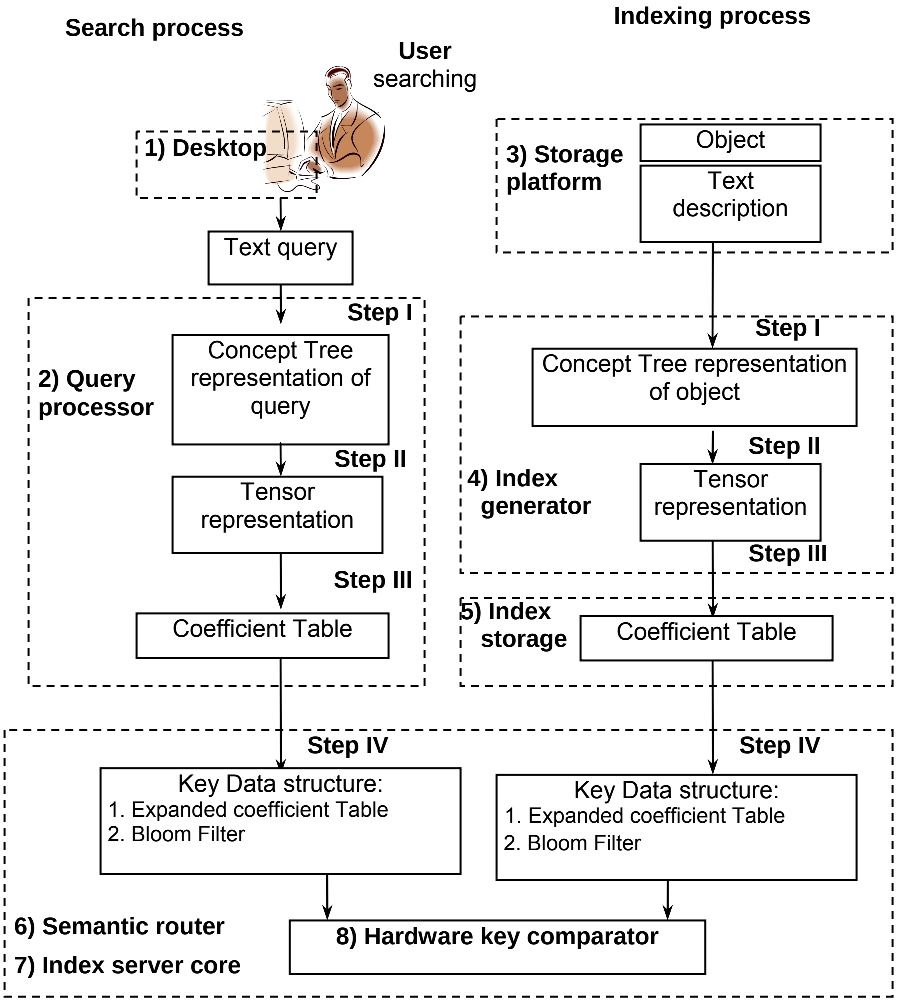


Fig. 3.2 Overview of the search process

The search process involves seven systems (shown by boxes with broken boundaries): (1) *user's desktop terminal*; (2) *query processor*; (3) *object storage platform*; (4) *index generator* system that generates semantic descriptors (keys) for the stored object; (5) *index storage* sub-system which stores all key-docId pair mappings inside the index server; (6) *semantic routers* which takes query/message routing decision by comparing a query/message's key against destination address keys; and (7) *index server core* which carries out the 2nd phase of index lookup for a given query by comparing it against object keys inside the index servers, after the 1st inverted index based lookup phase has been completed; and (8) the *hardware key comparator* located inside the semantic router and the index server core. As the comparison processes inside the semantic router core and the index server cores are similar, hence both processes are shown by a single common representation in Fig. 3.2 [13]. The key comparator will be located inside the semantic routers and also in the index servers where meaning comparisons will be carried out for query routing and index lookup. We propose that the key comparator be implemented by using a specialized hardware accelerator (e.g. ASIC based coprocessor on PCI-E card inside a server) for: speed, smaller hardware investment and energy efficiency. Based on the approaches proposed in this dissertation, it is possible to design such hardware [131]. In Fig. 3.2, the arrows indicate flow of data entities (shown as boxes with continuous lines).

Here we assume that an object is either a text or has a text description provided by the object owner/publisher. This text conveys the meaning of the object. Using the processing steps I to IV, as shown in Fig. 3.2, the final forms of the object and query keys are generated from the text. These final forms are used for key comparison in the hardware comparator.

In step I, the meaning of the text (a complex concept) is captured and represented as a *concept tree* representation (explained in details in next section) by a manual or automated process. In step II, an *algebraic (tensor) representation* of this concept tree is generated to enable *cosine similarity comparison* of two trees. In step III, this derived tensor is encoded in the *Coefficient table* data structure (details explained later in Chapter IV). This coefficient table is used as the format to store the key in the index (Fig. 3.2). In step IV, this table is extended to a Bloom Filter based data structure to enable fast parallel computation of cosine similarity using a simple hardware accelerator. During the query key generation (left side of Fig. 3.2), steps I to III take place within the search engine query processor. For object key generation (right side of Fig. 3.2), steps I to III takes place within the indexer system. Step IV takes place within the search engine/index core. Step III uses hash functions and bloom filters [129] that are specific to a search engine and indexer implementation. Sometimes there is also a need to find objects that are similar to a given object (e.g. as in recommender or search systems in Amazon [132], Pubmed [69]). In such cases, the given object's coefficient table is used as the query coefficient table. The steps III, IV and the descriptor data structure are necessary to satisfy requirements (E) and (F) as presented in the earlier section. Whereas step I will incorporate techniques that would satisfy requirements (A) to (D).

The cosine comparison for all the three the descriptor components are carried out in the hardware comparator based on strict string (or vector label) matching. There is no need to check for the semantic similarity of different keywords from ontology or keyword co-occurrence matrix during the dot product computation hence the process is straightforward and speedy. However step I & II will involve natural language processing and semantic relationships between words and phrases (this is not in the scope of the proposed dissertation). Even though step I & II are outside the scope of this dissertation, for sake of continuity, in the next section, we suggest how these steps may be implemented, along with other details of this meaning representation and comparison model. These are discussed to demonstrate the fact that already several techniques are available to support steps I and II and these steps are achievable provided some additional research work is carried out. These are identified in Chapter VIII.

3.4 Concept tree representation and its generation

Concept tree is an abstract representation of a composite meaning. From this concept tree representation a more concrete from of the descriptor data structure is generated. This section introduces the notion of concept tree followed by a description of suggested methods and techniques to generate a concept tree for a given text.

3.4.1 Notions and rationale

The concept tree construct is based on the notion of hierarchical collections as presented in Fig. 2.4 in section 2.4.6, Chapter II. We represent a hierarchical collection (or a nested set) of concepts by an equivalent tree structure as shown in Fig. 3.3. A collection of concepts are represented by the set notation where the curly brackets "{…}" denote the set or collection. In a concept tree, the nodes at the intermediate levels represent complex meanings which are hierarchical collection of elementary meanings (concepts). Thus concept tree representation supports generative composition of meanings to represent a complex concepts (meaning). This structure has backing from cognitive science and linguistic domains, as explained earlier in section 2.4.7 of Chapter II. Therefore concept tree representation satisfies requirement (A) and (B).

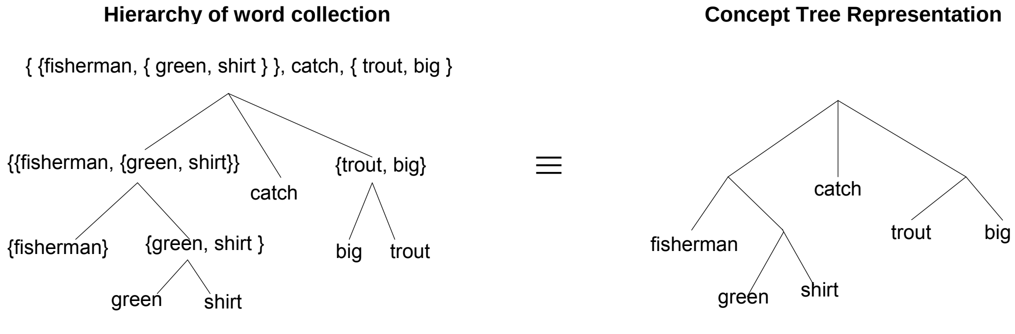

Hierarchy of word collection
{ {fisherman, { green, shirt } }, catch, { trout, big }
{{fisherman, {green, shirt}}
catch
{trout, big}
{fisherman}
{green, shirt }
big
trout
green
shirt
≡
Concept Tree Representation
catch
fisherman
trout
big
green
shirt

Fig. 3.3 Equivalence between nested set of elements and concept tree

3.4.2 Representation of standard concepts and contextual meanings

We consider two kinds of complex concepts. Both kinds are defined as a collection of its attributes. The first kind constitutes standard concepts which have standard names in the natural language vocabularies. For example the concept which is called "car" is expressed by its collection attributes as { car, wheel, engine, transportation, etc….}, where the name "car" is also an attribute. These are also called "named entities" in natural language processing domain.

The second kind includes complex concepts (composite meanings) which are contextual and transitory in nature. This kind of concept is only defined for the given context that arose during the discourse and the concepts of this kind are used to communicate a specific idea. These contextual concepts do not have standard names (or identifiers) in vocabularies and only exist for conveying an idea in the given context. This kind of concepts is expressed by a collection of the attributes that define the context. For example, in Fig. 2.4 and Fig. 3.3, the concept conveyed by the natural language sentence is the entire idea or story that is being conveyed.

3.4.3 Role of composition templates and ontology artifacts

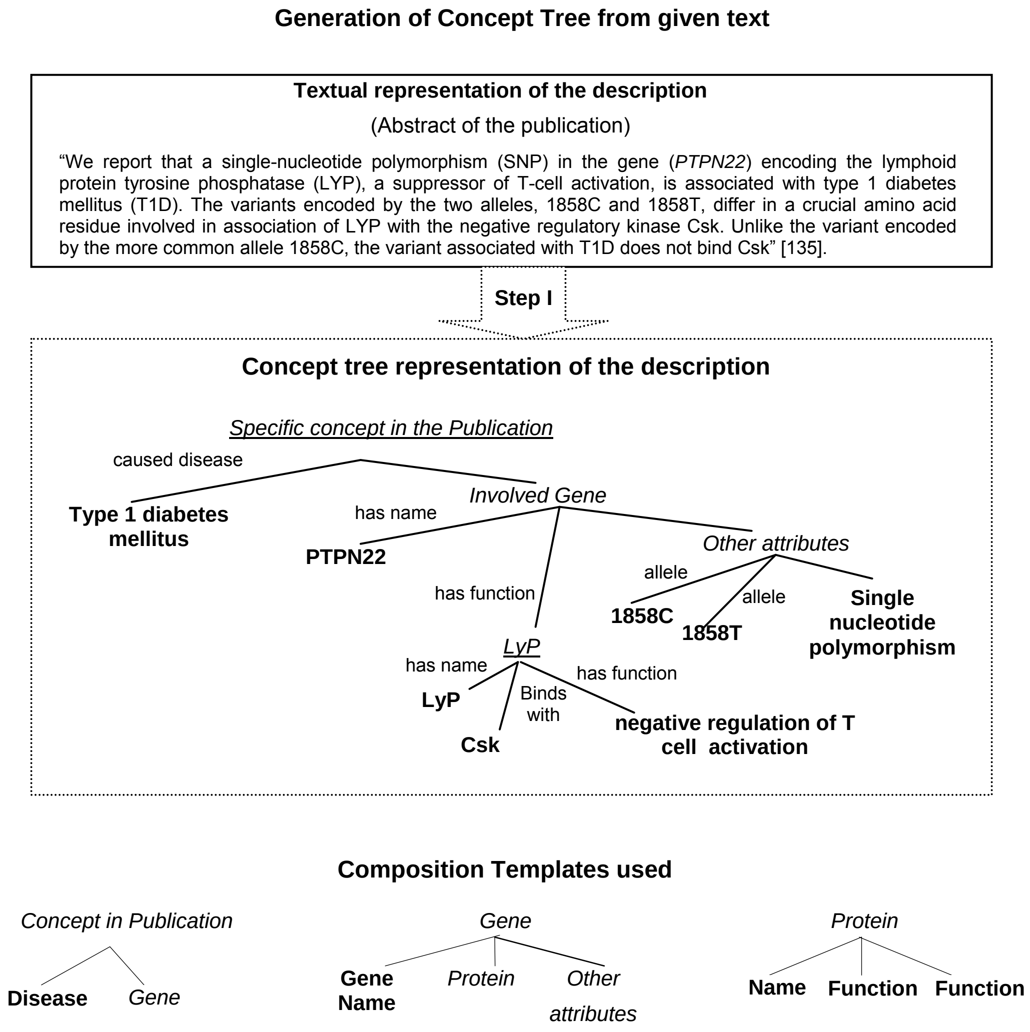
A key question is how does one decide what attributes to use to define a contextual concept. For some cases, it is possible to find common patterns in some of these contextual complex concepts. For example a concept of gene, as in bioscience, can be described by its name, by function, which is protein encoding, and also by its variant names (allele names). This choice of attributes, i.e., name, function (encoded protein), etc., is based on a fundamental knowledge model in bioscience (biochemistry) domain. In bioscience domain we recognize a gene in terms functions. Therefore, here we represent the specific gene "PTPN22" as a collection of its attributes as follows: {PTPN22, LyP, single nucleotide polymorphism, 1858T, 1858C,…….}, where LyP is the protein that is encoded by this gene PTPN22 and this gene demonstrates the "single nucleotide polymorphism" phenomenon and have polymorphic forms 1858T, 1858C, etc. This pattern to describe a gene is actually a composition template, which can be used to describe multiple other genes as well. This template is derived from a standard knowledge item from bioscience knowledge domain, which tells that a gene is involved with encoding of a specific protein and it can have multiple alleles. These knowledge artifacts can be formally structured as ontology artifacts and made available on demand to help formation of compositions whenever possible. Next we show how this notion may help to generate the concept tree structure for a given text document.

3.4.4 Concept tree structure for a given text

To explain the concept tree and its generation process we have considered a bioscience publication as a document object in Fig. 3.4 [13]. For such documents, the abstract of the publication can be used as the text description of the document. The advantage of using a readily available abstract is that meanings related information (the central idea) is available in the abstract in more concentrated form instead of being dispersed through the entire document text. Alternatively the entire document could be considered as the description. In this example, the abstract gives a fair idea about the meaning of the document. Even an abstract may be automatically generated using techniques mentioned in section 2.8.5, Chapter II and then used.

Fig. 3.4 presents the concept tree representation of this document. In the concept tree figure, each complex concept is defined by collection of elementary concepts. These elementary concepts are considered as the attributes of the complex concept, therefore taken together they denote the complex concept. The composed concepts are underlined and the standard concepts (called "basic concepts") are shown in bold as the leaves. Basic concepts have been defined as controlled terms in domain ontologies and lexicons like Gene Ontology [133], Disease ontology

[134], etc. Basic concepts are denoted by the corresponding terms and their ASCII character strings represent these terms inside the computer memory.



Generation of Concept Tree from given text

Textual representation of the description

(Abstract of the publication)

"We report that a single-nucleotide polymorphism (SNP) in the gene (*PTPN22*) encoding the lymphoid protein tyrosine phosphatase (LYP), a suppressor of T-cell activation, is associated with type 1 diabetes mellitus (T1D). The variants encoded by the two alleles, 1858C and 1858T, differ in a crucial amino acid residue involved in association of LYP with the negative regulatory kinase Csk. Unlike the variant encoded by the more common allele 1858C, the variant associated with T1D does not bind Csk" [135].

Fig. 3.4 Generation of concept tree from text

Here the given text is a publication which is a narrative about the gene PTPN22 whose malfunction causes diabetes and other autoimmune diseases. Therefore the publication is expressed as a contextual concept shown at the top of the tree. This concept is composed of two child concepts: the resulting disease "Type 1 diabetes mellitus" and the gene "PTPN22". Each of these child concepts are further defined by collection of elementary concepts. Thus the next two levels of the tree describe the concepts that represent the gene and its function. The concept of the “Lyp” protein is defined by multiple attributes: by its name “LyP”; by its “binds with” relationship with another gene product “Csk” and by its function “negative regulation of T cell activation”. Thus the “Lyp” concept is represented by a collection of all these elementary concepts. The rationale behind this representation is explained in the next section.

3.4.5 Tree construction rules

The specific rules of constructing this tree will depend on the domain knowledge models. These rules can be codified as composition templates which are to be used to represent concepts as and when required. Here three composition templates were used. The first one was the composition of disease name and implicated gene which served as the template to describe the publication. The second one was a composition of gene name, protein name, and other gene attributes (variations), which was used to represent the gene “PTPN22”. The third template was a composition of protein name and functions, which was used to represent the protein “Lyp”. These standard templates can be put in a library to be used during construction of the concept tree. Standard templates will enable use of common set of rules (domain knowledge) to construct concept trees and ensure proper meaning representation and similarity comparison. This will ensure that similar concepts are represented by similar concept tree representations (descriptors). This will enable proper similarity comparison between meanings.

The keywords or a phrases used at the leaves are selected from a controlled vocabulary to represent the meaning of the text for indexing purpose. When sufficient controlled vocabulary is available for a given domain, this technique can be applied to generate concept trees from text. The actual nature of relationships between concepts is ignored. However they are shown in the figure for the purpose of illustration. Inclusion of all relevant attributes of a context allows for disambiguation of a meaning without needing additional information, therefore satisfies requirement (D).

3.4.6 Incorporating subject category information

3.4.6.1 Motivation

Each index node specializes in storing one or more topic areas. The specialization assignment is made by choosing a model document/object and assigning its descriptor as the index node’s specialization descriptor. Here the idea is that all objects which are similar to this model object should be indexed by this index node. Therefore we have to ensure that descriptors of all similar objects should give high similarity value when compared against the descriptor of the model object.

The concept tree shown in Fig. 3.4 only includes information about the specific subject matter described in the text. If this tree is used as the object descriptor, then there is no way that this document can be found similar to other documents from bioscience domain, and thus can not be assigned to a index server that specializes in storing bioscience publications. Therefore the descriptor needs to incorporate additional information so that it can be found similar to the model document from same domain (i.e. molecular biology or biochemistry, in this example).

3.4.6.2 Suggested method

Here we will show hot to get a concept tree that defines the specific context for the given text along with the concepts that categorize the given publication. A collection of the concept tree for the specific context and all the category topics can be considered as a high level concept tree itself. This concept tree is shown in Fig. 3.5. In this simple example, we used two subject categories: “Molecular biology” and “Diseases” from Dewey decimal classification [136] (we choose this classification system in this example for sake convenience, some other taxonomy could have been also used). In real application this top level collection can include multiple subject categorization concepts to include all facets of the document.

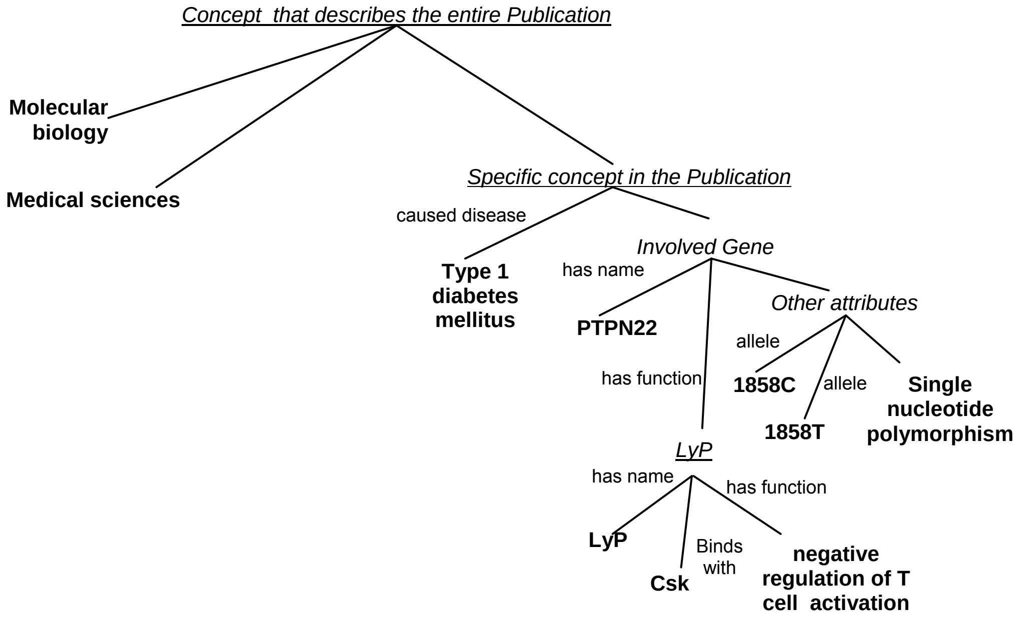


Fig. 3.5 Concept tree representation of the entire publication

3.4.7 Tree generation process

3.4.7.1 Involved artifacts

Generation of trees from a given text involves four kinds of artifacts, as follows:

1) Domain vocabularies or lexicons, based on which the next three artifacts are generated.

2) Standard concepts or named entities which are identified from the given text.

3) Domain ontologies, from which composition templates are generated.

4) Composition templates that can guide formation of contextual concepts.

Ontologies and lexicons may be available as multiple separate ontologies and lexicons or in partially integrated form.

3.4.7.2 Required processes

The tree generation involves a two phase process. In the first phase the artifacts that are needed to support the tree generation process is built. In this phase, the lexicon, ontological artifacts and templates are created and made available for use in the second phase. The second phase involves construction of a specific concept trees for a given text document or object. The first phase is common for all instances of concept tree generation (i.e., phase 2), whereas the second phase is specific to a concept tree that is being generated from a given text. A method is suggested below to carry out both these processes.

3.4.7.3 Phase 1: Generation of lexicon, ontology artifacts and templates

Lexicons can be manually created or can be semi-automatically generated under human supervision based on terms harvested from a corpus of texts from the given domains that are necessary. Similarly, ontology artifacts may be manually, partially or fully automatically created with or without human intervention. Technologies for creation of lexicons and ontologies are available and some have been mentioned in Chapter II, section 2.8.3 and 2.8.4.

Generation of templates involves identification of association patterns between standard concepts (named entities) that are common across multiple documents. These patterns may be available from the available ontologies that have been generated so far or may be manually created by human users. The creation of association patterns are part of the ontology creation process itself. For example, the strong association between genes and diseases may be observed in multiple scientific publications and may be incorporated as an ontology artifact. In this case each publication will have a specific association between a specific gene and a specific disease. Based on this association one can decide to formulate a template {gene name, disease name} to define a context that arises in a given publication.

3.4.7.4 Phase 2: Generation of concept tree for a given text

The steps for a suggested concept tree generation method is given below-

Step 1. Identify the broad topics of the given text by available topic identification techniques as mentioned in section 2.8.1, of Chapter II.

Step 2. In case abstract of the text is unavailable, create machine generated abstracts using text summarization techniques as mentioned in section 2.8.5, of Chapter II.

Step 3. Consider a moving window that puts a portion of the abstract text in the current scope. Identify all the named entities in the given text window. Techniques to identify named entities are available and have been discussed in section 2.8.3, of Chapter II.

Step 4. Check the available ontology to see whether any association exists between the named entities. Here an application of multiple techniques like: UNL deconversion, discourse analysis, machine learning techniques, etc., as mentioned in section 2.8, will be required. One way to achieve this is to parse the sentences using the UNL deconverters [95] and/or discourse analyzers [91]-[94] to generate their UNL form and/or discourse structures. Then from the UNL form and/or discourse structures, one can identify which named entity is associated with which other ones.

Once an association network between entities is identified, then we need to decide which single entities will be the primary ones, and which ones will be considered as its attributes. This can be based on the fact that the primary entities are generally the focus of the discussion and will be referred more frequently in other windows compared to other entities found in the current window. There can be other ways to identify the primary entities. Once the primary entities have been identified we can construct a small tree to represent the primary entity in terms of other entities. There may be more than one primary entity, which can not be associated based on available ontological artifacts. For each primary entity there should be a concept tree which may or may not have leaves.

Step 5. Keep on moving the window to identify more entities, construct more trees or grow the existing ones by attaching branches and leaves.

Step 6. Finally consider a collection of all disjointed trees as the top level collection that represents the contextual concept conveyed by the text document.

In the search process, as illustrated in Fig. 3.2, the ontology and templates are specific to the search engine query processor and indexer. Using the available ontology and knowledge base (construction templates, etc.) the search query processor and indexer system will make the best effort to iron out variations and ambiguities that might be present in user provided terms and construct the right kind of concept tree from the given text. For example, all synonyms will be replaced by similar trees, thereby addressing the synonymy problem. The named entity recognition techniques as mentioned in section 2.8.3, Chapter II, provides such capabilities.

3.4.8 Practical considerations

There is a limit how many basis vectors can be compared inexpensively, so the number of basis vectors should be limited by considering limited number of attributes in the lower level compositions and limiting the number of hierarchical levels in a concept tree. Only those attribute which are most important should be considered. When abstract is considered for generation of the concept tree, then only the most important named entities and their relationships are considered. This keeps the number of leaves in the concept tree under check.

3.4.9 Evidence that support the rationale behind the tree construct

Neuro-scientific findings reported in [23] indicate that human brain uses similar neurological processes to comprehend meaning from text and visual imagery. Other evidences [31],[32] suggested that semantic processing takes place at a common site irrespective of the kind of language that is being processed. This corroborates well with the argument that semantic processing and composition has its own set of rules which are common across different languages including visual imagery.

On the other hand, meaning representation techniques like Universal Networking Language [95] gives a confidence that using a small formal set of linguistic and meaning representation rules, it is possible to represent meanings of multiple languages. The concept tree is based on even more minimal set of semantic composition rules and these same set of rules are also the basis behind all these languages [23], [24]. Therefore concept tree model is also likely to be compatible with all these languages and meanings from visual imagery. Hence we expect that concept tree representation will be able to express meanings of natural language texts and other sources (e.g. images). Therefore concept tree representation is likely to satisfy requirement (C). As concept tree model is a good candidate for meaning representation, therefore it is worthwhile to explore and develop techniques for fast comparison of concept trees.

3.5 Tensor representation of a concept tree

3.5.1 Motivation

To compare two composite meanings we have to compare the concept trees that represent these meanings. Traditional tree comparison algorithms are elaborate and expensive [65], [66], [67], hence a faster alternative is required. We propose to generate the tensor representation of a concept tree then compare two tensors of two given concept trees by taking their dot (cosine) products. To compute tensor dot products at high speeds, we have developed a computational method and an associated hardware based processing architecture. Therefore to exploit this technique we need to generate tensor representation of a concept tree.

3.5.2 Overview of the technique

In a concept tree, each composition is a set, and the entire concept tree is actually a nested set (set of sets) having hierarchical structure (Fig. 3.6). Such a nested set or a tree structure can be expressed as a tensor which is sum of scalar weighted polyadic products (polyads) [137] of the basic basis vectors that represent the elements at the lowest/innermost levels of the nested set (leaves of the tree). Fig. 3.6 illustrates this process of converting a concept tree of a document (only publication title is shown) to its tensor representation. Only the tensor representation of the shaded portion of the tree is shown for sake of simplicity. Any particular composition (or a set) in the hierarchical structure is represented as a tensor, which is a function of all the elementary meanings contained in the set. Here the elements (say A, B, C…) are considered as tensors themselves, and their composition (or the set) is expressed as {A, B, C, ….}, where the curly brackets "{…}" denote a certain algebraic function of: A, B, C,…. . This function can express the set as a mix of conjunction and disjunction of the constituent elements. An algebra is developed to formally represent all the required rules. This algebra, the explanation of the delimiter vectors “▷” and “◁” and the generative method to produce the scalar coefficients and the polyads, are presented below.

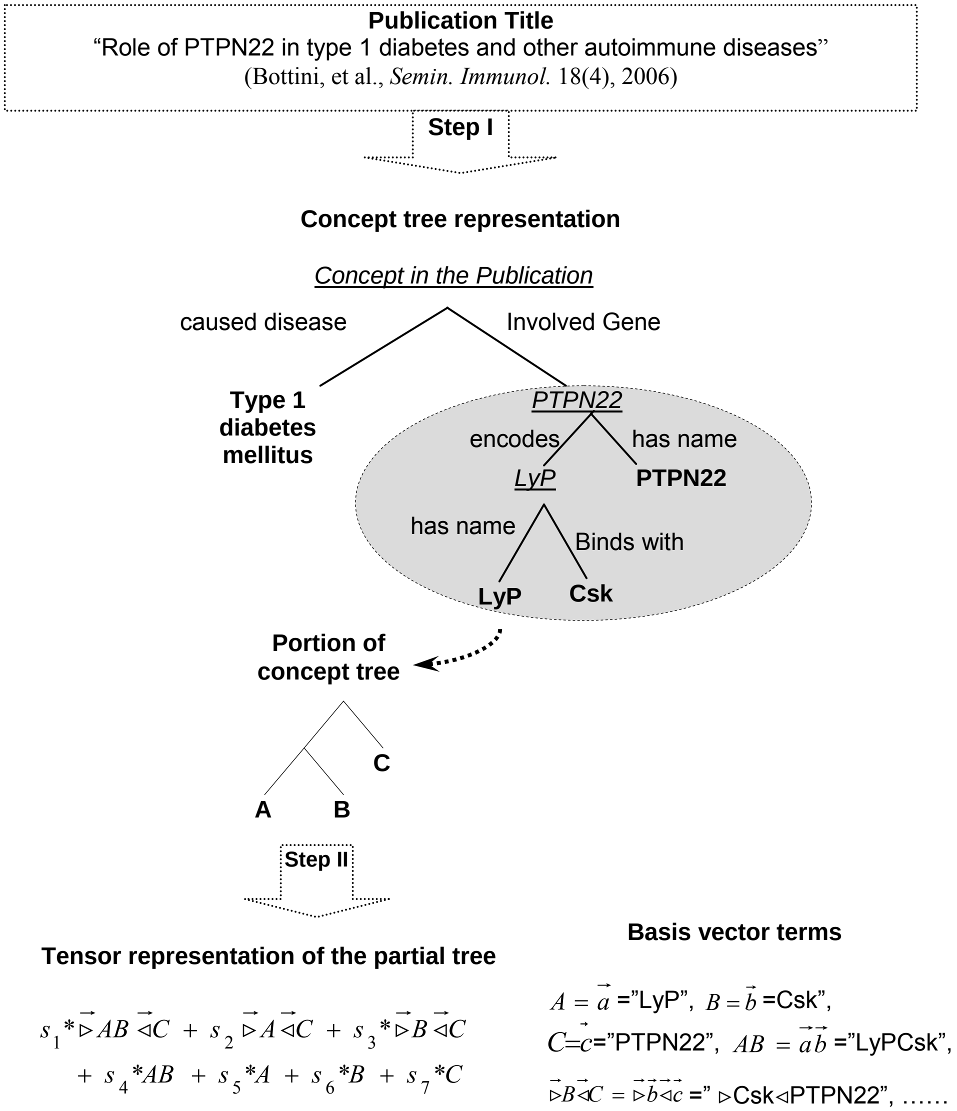

Publication Title
“Role of PTPN22 in type 1 diabetes and other autoimmune diseases”
(Bottini, et al., *Semin. Immunol.* 18(4), 2006)
Step I
Concept tree representation
Concept in the Publication
caused disease
Involved Gene
Type 1 diabetes mellitus
PTPN22
encodes
has name
LyP
PTPN22
has name
Binds with
LyP
Csk
Portion of concept tree
C
A
B
Step II
Tensor representation of the partial tree
$s_1 * \vec{\triangleright} AB \vec{\triangleleft} C + s_2 \vec{\triangleright} A \vec{\triangleleft} C + s_3 * \vec{\triangleright} B \vec{\triangleleft} C$
$+ s_4 * AB + s_5 * A + s_6 * B + s_7 * C$
Basis vector terms
$A = \vec{a}$ =”LyP”, $B = \vec{b}$ =Csk”,
$C = \vec{c}$=”PTPN22”, $AB = \vec{a}\vec{b}$ =”LyPCsk”,
$\vec{\triangleright}B\vec{\triangleleft}C = \vec{\triangleright}\vec{b}\vec{\triangleleft}\vec{c}$ =” ▷Csk◁PTPN22”, ……

Fig. 3.6 Tensor representation of concept tree

3.6 Required algebra

3.6.1 Notations and significance

Here we use four quantities:

(1) basic basis vectors, denoted by smaller case alphabets with an arrow on the top (e.g. $\vec{a}$);

(2) scalar coefficients, by smaller case alphabets (e.g. "s_i");

(3) polyadic of basic basis vectors, as an sequence of basic basis vector notations (e.g. $\vec{a}\vec{b}\vec{c}$);

(4) tensors, by upper case alphabets (e.g. "A").

Within computers, basis vectors are represented by character strings (keywords/phrases). These represent elementary meanings/concepts (e.g. "Csk", "negative regulation of T cell activation") or special purpose characters ("▷", "◁") having specific functions. Tensors represent composite meaning (complex concept). Algebraically they are represented as sum of scalar weighted basis vectors.

Polyadic combinations are represented as concatenated strings that represent individual basic basis vectors. For example, $\vec{\triangleright}\vec{a}\vec{b}\vec{\triangleleft}$ is represented by a string "▷CskLyP◁" which means conjunction of both basic basis vectors $\vec{a}$ ("Csk") and $\vec{b}$ ("LyP"). By $\mathrm{hsh}(\vec{a})$, we mean the hash (e.g. FNV hash [138]) value of the character string that is represented by a basis vector $\vec{a}$.

3.6.2 Definition of the vector space

A concept is expressed as a tensor [137] in an infinite dimensional space. This space is represented by two kinds of basis vectors. One kind comprises of a set of basic basis vectors (e.g. $\vec{a}$), each of which corresponds to a unique basic concept (e.g. "Csk", "LyP", "negative regulation of T cell activation") in the domain lexicon. The other kind includes basis vectors which are polyadic combinations represented in the form: $\vec{a}\vec{b}\vec{c}$, which represent conjunction of concepts (e.g. "LyP" & "CsK" & "negative regulation of T cell activation" &…). This second kind of basis vector is needed because elementary concepts can combine with each other to form complex concepts which are entirely different from the elementary ones. The semantic similarity between two concepts is given by the cosine product of their tensors representations. The basis vectors and their polyads represent different concepts, hence we assert that all the dot product of orthogonal basis vectors are zero. This rule can be expressed as: $\vec{a} \bullet \vec{b} = \vec{\triangleright}\vec{a}\vec{b}\vec{\triangleleft} \bullet \vec{a} = \vec{\triangleright}\vec{a}\vec{b}\vec{\triangleleft} \bullet \vec{c} = \vec{\triangleright}\vec{a}\vec{b}\vec{c}\vec{\triangleleft} \bullet \vec{\triangleright}\vec{a}\vec{b}\vec{\triangleleft} = \ldots \equiv 0$.

3.6.3 Algebraic representation of composition and binder functions

We use polyadic product of tensors to represent the composition of the elementary tensors. These polyadic tensor products [137] are distinct from tensor cross (sine) products or dot (cosine) products. Polyadic products are denoted by ordered juxtaposition of tensors, for example, AB denotes a dyadic tensor product, ABC denotes a triadic tensor and ABCD... represents a polyadic of individual tensors: A, B, C, D, etc. In general, AB ≠ BA, i.e. the product operation is not commutative. This definition can be expanded for a general case of *n* arguments [139], [13].

A triadic tensor product ABC represents a conjunction composition where three elementary tensors A, B and C are arranged in a specific order from left to right. However this tensor product is not sufficient to represent a simple single level collection, because it does not satisfy two important requirements:

1. The expression has to be commutative to avoid forced ordering of elements.

2. It has to represent both conjunction and disjunction composition at the same time.

Hence we need a different algebraic representation that can satisfy both these requirements to represent a collection, which does not force ordering of elements. To achieve this purpose we introduce two algebraic binders (functions): (1) **[**A**,** B**,**…**]** ; and (2) {A**,** B**,** …}, that bind two or more tensors (concepts) together. For example, by using the binder function **[**A**,** B**,** C**]**, three tensors A, B and C, can be bound together to represent a certain composition of A, B, and C. Using these two binder functions we synthesize an algebraic (tensor) representation that can depict a concept tree in terms of its leaves. These binder functions represent compositions which do not force ordering of arguments. This means that these binder functions are commutative with respect to their arguments. This ensures that all possible isomorphic trees (e.g., Fig. 3.7 [139]) that convey the same meaning are expressed by a single tensor.

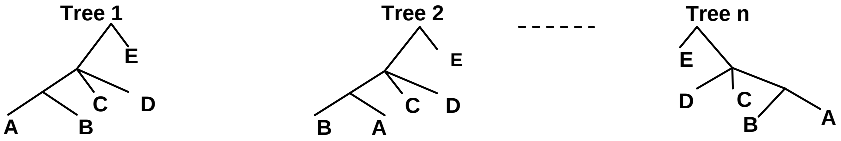


Fig. 3.7 Isomorphic concept trees which convey the same meaning

In the following sections, we define the aforementioned commutative algebraic binders (functions).

3.6.4 Definition of **[**…,… ,…**]** *binder function*

For one, two and three and *n* arguments we define the following rules [13]:

$$[\vec{a}] \equiv \vec{a} \tag{3.1}$$

$$\begin{aligned}[\vec{a},\vec{b}] &\equiv \vec{\triangleright}\vec{a}\vec{b}\vec{\triangleleft} \quad \text{if } \mathrm{hsh}(\vec{a}) > \mathrm{hsh}(\vec{b}) \\ &\equiv \vec{\triangleright}\vec{b}\vec{a}\vec{\triangleleft} \quad \text{if } \mathrm{hsh}(\vec{b}) > \mathrm{hsh}(\vec{a})\end{aligned} \tag{3.2}$$

$$\begin{aligned}[\vec{a},\vec{b},\vec{c}] &\equiv \vec{\triangleright}\vec{a}\vec{b}\vec{c}\vec{\triangleleft} \quad \text{if } \mathrm{hsh}(\vec{a}) > \mathrm{hsh}(\vec{b}) > \mathrm{hsh}(\vec{c}) \\ &\equiv \vec{\triangleright}\vec{b}\vec{a}\vec{c}\vec{\triangleleft} \quad \text{if } \mathrm{hsh}(\vec{b}) > \mathrm{hsh}(\vec{a}) > \mathrm{hsh}(\vec{c}) \\ &\vdots \\ &\equiv \vec{\triangleright}\vec{a}\vec{c}\vec{b}\vec{\triangleleft} \quad \text{if } \mathrm{hsh}(\vec{a}) > \mathrm{hsh}(\vec{c}) > \mathrm{hsh}(\vec{b})\end{aligned} \tag{3.3}$$

$$[s_a\vec{a},\ s_b\vec{b},\ s_c\vec{c}....] \equiv s_a s_b s_c.....[\vec{a},\ \vec{b},\ \vec{c}.....] \tag{3.4}$$

$$[(\vec{a}+\vec{b}+....),(\vec{c}+\vec{d}+....)] \equiv [\vec{a},\vec{c}]+[\vec{a},\vec{d}]+....+[\vec{b},\vec{c}]+[\vec{b},\vec{d}]+.... \tag{3.5}$$

$$[\ A\ ] \equiv A \tag{3.6}$$

$$\begin{aligned}&\text{If } \ A = \sum_i s_{a,i}\overrightarrow{a_i}\,,\ \ B = \sum_i s_{b,i}\vec{b}_i\,,\ \ C = \sum_i s_{c,i}\overrightarrow{c_i}\,,\ \ .... \\ &\text{then } \ [A,B,C,....] \equiv \sum_{i,j,k} s_{a,i}\ s_{b,j}\ s_{c,k}.....[\ \vec{a}_i,\vec{b}_j,\vec{c}_k,.....]\end{aligned} \tag{3.6}$$

Proof for the commutative property, for the two argument case, is given below which can be extended for *n* arguments. Based on equation (3.2), we can claim that [A, B] = [B, A], because whatever the ordering of the arguments, the outcome only depends on the ranking of their hash values. Which means the tensor representation does not depend on the argument ordering.

3.6.5 Definition of {…, …., …} *binder function*

For one, two and three argument cases, we define the followings:

$$\{\ A\ \} \equiv \frac{\sqrt{h_A}\,[\ A\ ]}{\sqrt{h_A}} = A \tag{3.7}$$

$$\{A, B\} \equiv \frac{\sqrt{h_{AB}}[A, B] + \sqrt{h_A}[A] + \sqrt{h_B}[B]}{\sqrt{h_{AB} + h_A + h_B}} \tag{3.8}$$

$$\{A,B,C\} \equiv \frac{(\sqrt{h_{ABC}}[A,B,C] + \sqrt{h_{AB}}[A,B] + \sqrt{h_{BC}}[B,C] + \sqrt{h_{AC}}[A,C] + \sqrt{h_A}[A] + \sqrt{h_B}[B] + \sqrt{h_C}[C])}{\sqrt{h_{ABC} + h_{AB} + h_{BC} + h_{AC} + h_A + h_B + h_C}} \tag{3.9}$$

This binder encompasses all possible combinations and permutations of arguments. The resultant tensor is also normalized and used as an elementary tensor to be incorporated for next higher level of composition. Each instance of this binder has a corresponding set of co-occurring coefficients "*H*", having real valued scalar elements (e.g. H = set { h_{ABC}, h_{AB}, h_{BC}, h_{AC}, h_A, h_B, h_C}), each of which indicates the importance of the corresponding polyad to represent the meaning of the composed concept. For example, when only $h_{ABC} = 1$ and all other scalars $h_{AB} = h_{BC} \ldots = h_C = 0$, then the composed concept is the one which is given by a strict conjunction of A,B and C. Whereas the set $h_A = h_B = h_C = 1$ and $h_{ABC} = h_{AB} = h_{BC} = h_{AC} = 0$ represents disjunction composition. A mix of all these extremes is possible by suitable choice of values for the co-occurring coefficients. This enables a controlled mix of conjunction and disjunction composition to suit specific situations. Rules that guide assignment of these values can be codified and made accessible along with composition templates.

Proof for the commutative property of this binder function is given below. Suppose a function "*F*" is a linear composition (linear functional) of two other functions "F_1" and "F_2", such that

$F = \lambda_1 * F_1 + \lambda_2 * F_2$, where λ_1 and λ_2 are real numbers.

If both functions F_1 and F_2 are commutative, then their linear functional F is also commutative.

The rationale of this property is explained below for two arguments, but it can be extended for *n* arguments.

$$\text{If } F_1(A, B) = F_1(B, A) \text{ and } F_2(A, B) = F_2(B, A)$$

$$\text{Therefore, } F(A, B) = \lambda_1 {*} F_1(A, B) + \lambda_2 {*} F_2(A, B) = \lambda_1 {*} F_1(B, A) + \lambda_2 {*} F_2(B, A)$$

Here F_1 and F_2 are the **[**…,… ,…**]** binder functions and F is the **{**…, …., …**}** binder function. Therefore as the **{**…, …., …**}** binder is a linear functional of commutative **[**…,… ,…**]** binders, therefore the {…, …., …} binder (i.e., "*F*") is also commutative. The commutative property of the **{**…, …., …**}** binder makes the composition tensor insensitive to ordering of the leaves.

3.6.6 An example to illustrate tensor generation

Here we explain how the tensor expression of the **{**…, …., …**}** binder also supports generative composition. In Fig. 3.8, all the intermediate composed concepts for the given concept tree are represented by the expressions shown at corresponding nodes in terms of the **{** …, …,…**}** binder. On expanding these expressions we get the tensor. Each of the two compositions: {A,B} and {{A,B},C} in the example tree, has two associated co-occurrence sets: ^{AB}H, $^{(AB)C}H$, as shown in Fig. 3.8.

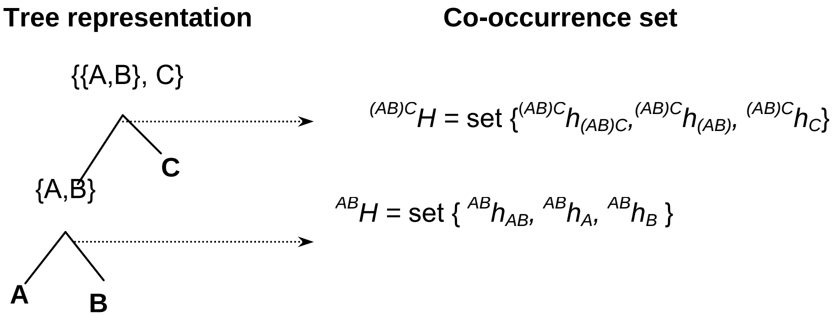

Tree representation
Co-occurrence set
{{A,B}, C}
{A,B}
C
A
B
$^{(AB)C}H$ = set {$^{(AB)C}h_{(AB)C}$, $^{(AB)C}h_{(AB)}$, $^{(AB)C}h_C$}
^{AB}H = set { $^{AB}h_{AB}$, $^{AB}h_A$, $^{AB}h_B$ }

Fig. 3.8 Concept tree tensor expression

The expression {{A,B}, C} which represents the tree in Fig. 3.8, is expanded bottom up as an example [13], using equation (3.1) to (3.9). Only a few terms are shown for a specific case when $\mathrm{hsh}(\vec{\mathrm{a}}) > \mathrm{hsh}(\vec{\mathrm{b}}), \mathrm{hsh}(\vec{\triangleright}\vec{\mathrm{a}}\vec{\mathrm{b}}\vec{\triangleleft}) > \mathrm{hsh}(\vec{\mathrm{c}})$.

$$A = \vec{a}, B = \vec{b}, C = \vec{c}, \text{ and } \{A, B\} = \frac{\sqrt{{}^{AB}h_{AB}}\,\vec{\triangleright}\vec{a}\vec{b}\vec{\triangleleft} + \sqrt{{}^{AB}h_{A}}\,\vec{a} + \sqrt{{}^{AB}h_{B}}\,\vec{b}}{\sqrt{{}^{AB}h_{AB} + {}^{AB}h_{A} + {}^{AB}h_{B}}}$$

$$\{\{A, B\}, C\} = \frac{\sqrt{{}^{(AB)C}h_{(AB)C}}\,[\{A, B\}, C] + \sqrt{{}^{(AB)C}h_{AB}}\,\{A, B\} + \sqrt{{}^{(AB)C}h_{C}}\,C}{\sqrt{{}^{(AB)C}h_{(AB)C} + {}^{(AB)C}h_{AB} + {}^{(AB)C}h_{C}}}$$

$$= \frac{\sqrt{{}^{(AB)C}h_{(AB)C}}}{\sqrt{{}^{(AB)C}h_{(AB)C} + {}^{(AB)C}h_{AB} + {}^{(AB)C}h_{C}}} * \frac{\sqrt{{}^{AB}h_{AB}}}{\sqrt{{}^{AB}h_{AB} + {}^{AB}h_{A} + {}^{AB}h_{B}}} (\vec{\triangleright}\vec{\triangleright}\vec{a}\vec{b}\vec{\triangleleft}\vec{c}\vec{\triangleleft})$$

$$+ \frac{\sqrt{{}^{(AB)C}h_{(AB)C}}}{\sqrt{{}^{(AB)C}h_{(AB)C} + {}^{(AB)C}h_{AB} + {}^{(AB)C}h_{C}}} * \frac{\sqrt{{}^{AB}h_{A}}}{\sqrt{{}^{AB}h_{AB} + {}^{AB}h_{A} + {}^{AB}h_{B}}} (\vec{\triangleright}\vec{a}\vec{c}\vec{\triangleleft})$$

+ ....

$$+ \frac{\sqrt{{}^{(AB)C}h_{AB}}}{\sqrt{{}^{(AB)C}h_{(AB)C} + {}^{(AB)C}h_{AB} + {}^{(AB)C}h_{C}}} * \frac{\sqrt{{}^{AB}h_{AB}}}{\sqrt{{}^{AB}h_{AB} + {}^{AB}h_{A} + {}^{AB}h_{B}}} (\vec{\triangleright}\vec{a}\vec{b}\vec{\triangleleft})$$

+ ....

$$+ \frac{\sqrt{{}^{(AB)C}h_{AB}}}{\sqrt{{}^{(AB)C}h_{(AB)C} + {}^{(AB)C}h_{AB} + {}^{(AB)C}h_{C}}} * \frac{\sqrt{{}^{AB}h_{B}}}{\sqrt{{}^{AB}h_{AB} + {}^{AB}h_{A} + {}^{AB}h_{B}}} \vec{b}$$

$$+ \frac{\sqrt{{}^{(AB)C}h_{C}}}{\sqrt{{}^{(AB)C}h_{(AB)C} + {}^{(AB)C}h_{AB} + {}^{(AB)C}h_{C}}} \vec{c}$$

This method presented above illustrates that is possible to use the {…, …., …} binder function recursively for two levels of composition. If this recursion is defined for two levels, then it can be logically established (proof by induction) that it is also defined, in general, for *n* multiple levels. Therefore this {…, …., …} binder function can be used to enable recursive (generative) composition.

3.6.7 Algorithm to generate tensor expression for a given concept tree

Here we suggest an algorithmic method to generate tensor expression for a given concept tree. Here the abstract concept tree is represented as a tree data structure in the computer memory. The tensor expression is represented in the memory by a suitable data structure, which maintains the information about the sum and product relationship between the individual terms and scalar coefficient values. The proposed algorithm uses this tree data structure as the input and generates the tensor data structure as the output.

In this method the algebraic representations are generated for each nodes, starting from leaves from bottom of the tree and progressed up to the top most root node. The steps of this algorithm are as follows:

Step 1. Iterate the concept tree by depth-first basis.

Step 2. If the current node is a leaf, then do nothing.

Step 3. If the current node is not a leaf then consider all the children nodes, take the basis vector terms of the children nodes that constitutes their tensor representations, then using equation (3.1) to (3.9) construct the terms of the tensor representation for the current node. Store this tensor representation at the current node. This tensor represents the composition of current node of the concept tree.

Step 4. Iterate to the next location of the concept tree (go to step 2).

Step 5. At the completion of depth-first iteration, pick up the tensor representation of the entire tree from the top most root node.

3.6.8 Rationale for using delimiter characters

The ordering and combination of the leaf tensors and the delimiter characters “▹” and “◃” in the polyads retains the information about the tree structure. For example in Fig. 3.9, the two trees have similar leaves but different compositions. The arrangement between the leaves and the delimiter vectors distinguishes the two tensors that represent the respective trees. This is because for the example in Fig. 3.9, the tensor dot product value (which is < 1), is smaller than the vector dot product value (which is = 1). Therefore the tensor based comparison can recognize the two tensors (and the trees) as being different, whereas the vector based comparison perceives the vectors to be similar.

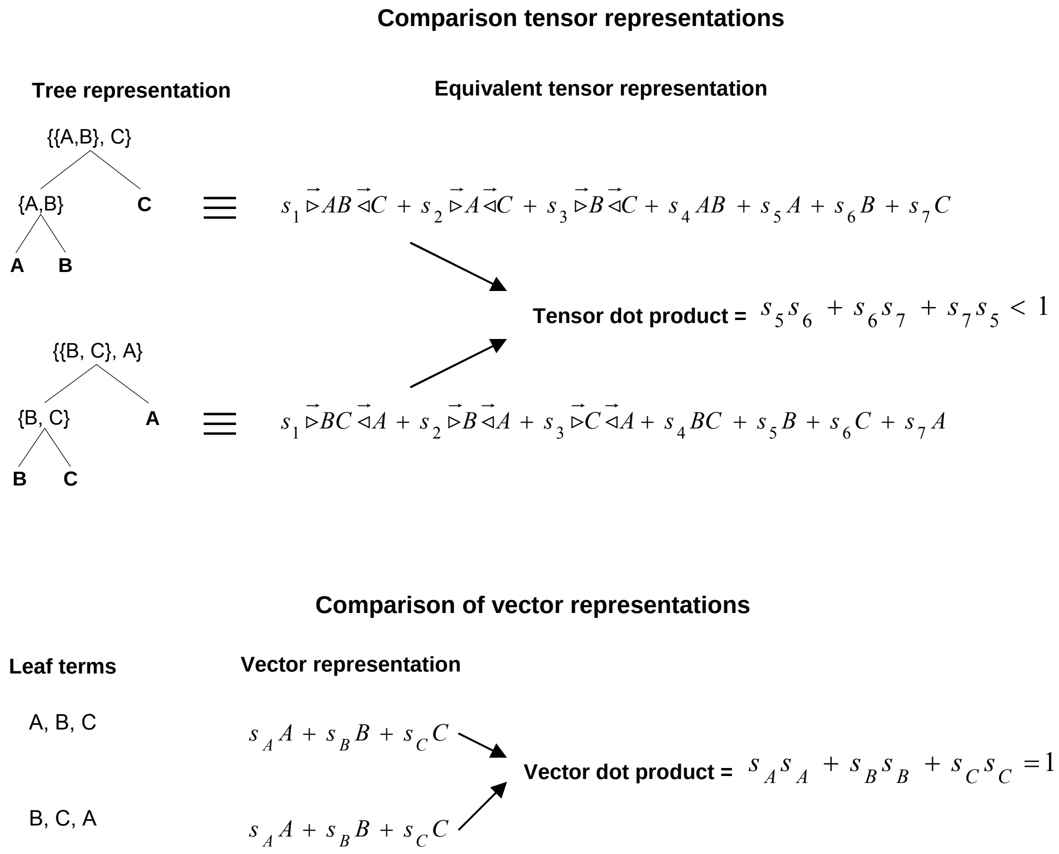

Comparison tensor representations
Tree representation
Equivalent tensor representation
{{A,B}, C}
{A,B}
C
A
B
$\equiv$
$s_1 \vec{\triangleright} AB \vec{\triangleleft} C + s_2 \vec{\triangleright} A \vec{\triangleleft} C + s_3 \vec{\triangleright} B \vec{\triangleleft} C + s_4 AB + s_5 A + s_6 B + s_7 C$
Tensor dot product = $s_5 s_6 + s_6 s_7 + s_7 s_5 < 1$
{{B, C}, A}
{B, C}
A
B
C
$\equiv$
$s_1 \vec{\triangleright} BC \vec{\triangleleft} A + s_2 \vec{\triangleright} B \vec{\triangleleft} A + s_3 \vec{\triangleright} C \vec{\triangleleft} A + s_4 BC + s_5 B + s_6 C + s_7 A$
Comparison of vector representations
Leaf terms
Vector representation
A, B, C
$s_A A + s_B B + s_C C$
Vector dot product = $s_A s_A + s_B s_B + s_C s_C = 1$
B, C, A
$s_A A + s_B B + s_C C$

Fig. 3.9 Role of delimiter vectors in distinguishing compositions

3.7 Method to incorporate hypernyms

3.7.1 Motivation

In Fig. 3.5, “Type 1 diabetes mellitus” (hyponym) is a specific instance of “Auto immune disease” (hypernym). Similarly the subject category information items “Bioscience” and “Medical sciences” have common taxonomic ancestors, i.e. hypernyms, according to the Dewey classification system [136], which we adopted for the sake of this example. We need a method to incorporate these hypernyms or taxonomic ancestors so that when a user searches for with a query concept tree that has the taxonomic ancestors, the query should be somewhat similar to the given document's descriptor. This notion is best explained by an example, which is presented below.

3.7.2 Suggested method

Any leaf term represented by basis vector $\vec{c}$, should be replaced by a tensor that represents all the taxonomic ancestors of this term that corresponds to this basis vector. This is best explained by an example. Fig. 3.10 shows a portion of the taxonomy of the basic concept "Type 1 diabetes".

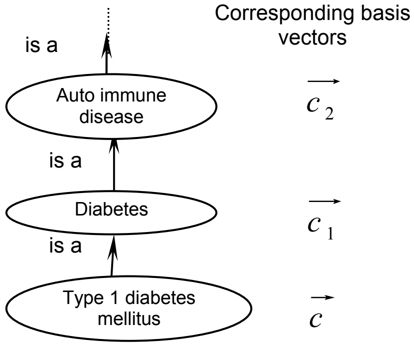


Fig. 3.10 Taxonomic ancestors of a given term

Here the term "Type 1 diabetes mellitus" corresponds to the basis vector $\vec{c}$. All the taxonomic ancestors of this terms corresponds to the basis vectors: $\vec{c}_1$, $\vec{c}_2$, $\vec{c}_3$, and so on. Here $\vec{c}_1$ corresponds to "Diabetes", $\vec{c}_2$ corresponds to "Auto immune disease", etc.

Suppose a portion of the original tensor expression is $..... + s_{ijk} \vec{i}\vec{j}\vec{k} + ... + s_{abcd} \vec{a}\vec{b}\vec{c}\vec{d} + ...$, then $\vec{c}$ in this expression is replaced by $(y\vec{c} + y_1\vec{c_1} + y_2\vec{c_2} + ...y_n\vec{c_n})$. Therefore the expanded tensor is now given as - $........ + s_{ijk} \vec{i}\vec{j}\vec{k} + s_{abcd}\, y\vec{a}\vec{b}\vec{c}\vec{d} + s_{abcd}\, y_1\vec{a}\vec{b}\vec{c_1}\vec{d} + s_{abcd}\, y_2\vec{a}\vec{b}\vec{c_2}\vec{d} + ...s_{abcd}\, y_n\vec{a}\vec{b}\vec{c_n}\vec{d} + ...$, where y_i are weighting parameters, such that $y > y_i > y_{i+1}$ and the sum of their squares is one. This choice of weights ensures that when both tensors have the same term $\vec{c}$, then the similarity value is maximum. If the term $\vec{c}$ is not the common term, but $\vec{c}_1$, $\vec{c}_2$, $\vec{c}_3$, etc., are common, then the similarity value is smaller than the previous case, but larger than the case when $\vec{c}_1$ is not common but $\vec{c}_2$, $\vec{c}_3$, etc., are common, and so on. This satisfies the subsumption related requirement (D).

3.8 Additional applications of the tensor model

3.8.1 Possible applications

In addition of using the meaning composition framework to represent composite meanings, several other applications of the tensor models are also possible. The proposed tensor framework can be used to bolster the traditional term vector models and improve its semantic performance. In this case we do not use a large concept tree and its large tensor to represent the entire object as explained earlier. Rather, we use small tensors to selectively replace some of the basis vectors in a term vector model (where the leaves may be text terms or from controlled vocabulary). This can improve the search performance of information retrieval systems by improving its power to discern composite meanings. Some of these are discussed below:

1. *Define a term by its attributes*: Compositions having form {*item*, *attributes*} can be used instead of only *item* to define the term. E.g. use the composition {“LyP”, “Csk”} to denote the function of “LyP” in terms of relationship with “Csk” instead of only “LyP”. The curly bracket notation “{ , , …}” was explained earlier.

2. *Define the specific usage context for a term*: Composition template to be used is in the form: {*item*, *context attributes*}. E.g., use the composition {“diabetes”, “PTPN22”} to denote the cause aspect of the disease instead of only “diabetes”.

3. *Aid term disambiguation*: E.g., instead of using the single term “mouse”, use the compositions: {“mouse”, “animal”} or {“mouse”, “computer device”}. This composition {“mouse”, “animal”} will clearly denote that in the given context we mean the rodent not the computer device. This addresses the polysemy problem (ambiguity in meaning).

3.8.2 Methods to materialize term disambiguation

Here we explain how one of the applications presented above (e.g. point 3 as above) can be implemented using existing technologies. This scheme, if used, can avoid the problem illustrated in Table 1.2 in Chapter I. This scheme illustrates the process of generating tensor representations from natural texts. The steps to implement this scheme are as follows:

Step 1. Generate generic tree templates that can express terms as presented in point 3 above. Use these templates to generate concept trees for various named entities (e.g. animal mouse and/or computer device mouse). The simplest template with a single attribute will have the form: { *item name*, *attribute* }. Maintain a library of trees for named entities that require disambiguation.

Step 2. Process a given text object to recognize the various named entities [72] using state-of-art named entity recognition (NER) techniques.

Step 3. Convert the concept trees to respective algebraic forms and replace the entity terms (e.g. “mouse”) by their corresponding tensor representations, then insert these in the meaning vector for that document (or text fragment under consideration). This process is to be done during indexing when the TF-IDF vectors [9] are generated. This process will be carried out at the query processor to generate the query vector, as shown in Fig. 3.1.

Step 4. Use the meaning vector which is now retrofitted with the tensors, for indexing or searching purpose.

An alternate method can be also adopted, where small concept trees can be embedded in the XML or newer version of HTML documents to disambiguate certain named entities. Using step 1 and 2, these concept trees can be generated, and then they can be embedded in the document itself using XML based microdata [62], microformat [63], and newer HTML standard [64], as suitable. These steps, i.e. step 1 & 2 as presented above, can be carried by the object owner during generation of the document. Thus this can obviate step 1 & 2 during the searching or indexing operation (making it faster) because, in this case the concept tree structures will be readily available from the document itself.

3.9 Integrating tensor, latent semantic indexing and term vector models

3.9.1 Motivation

No one single meaning representation method alone will be sufficient. The proposed tensor model has certain strengths that can complement the existing term vector or latent semantic indexing (LSI) models, when used along with them. Hence we suggest that the tensor model should be used in conjunction with existing vector models like LSI and term vector models to complement them. This is because LSI and term vector models have following limitations.

The term vector model [9] can not address the synonymy and polysemy problems [8], [50]. Different flavors of LSI models can address some of these limitations, however a LSI model, may disregard an important meaning bearing term if documents bearing that term are not incorporated in the training corpus. This will lead to wrong routing and object placement decisions in our proposed distributed index system. In fact, it has been argued [8] that it is still inconclusive which one – the term vector model or the LSI model, has better performance. Therefore the traditional term vector model may still have a role to play because it does not have this limitation of the LSI model.

Both term vector and LSI models suffer from the "bag of words" limitations [11], hence, these vector models fail to discriminate meaning of texts which have different meanings but have similar set of keywords. Our proposed tensor model addresses this limitation, however the tensor representation can become very large and un-wieldy if used indiscriminately (refer discussion in earlier section 3.4.8). It appears that a better choice is to use a weighted or rule based mix of multiple models. The tensor model should preferably be used to represent the overall meaning of the object, whereas the LSI and term vector models can be based on the terms found in the text object. Even though other models are possible, here we limit ourselves to only vector models as they are established ones.

The method to compose a single similarity values from these three models can be derived or synthesized. However this will require data from large scale user based experimental studies. Such investigation is outside the scope of this dissertation. Here, we will only assume that there are three separate approaches, i.e. term vector, LSI and tensor, to compute the similarity and once a similarity value composition method has been decided, then the three values computed by these approaches can be composed together to compute a single similarity value.

3.9.2 Design of the integrated semantic descriptor data structure

To enable use of all the three models together as mentioned above, we propose that the semantic data structure should have three components. The first component, called latent semantic component, is a limited dimension (<1000) latent semantic vector generated according to the latent semantic indexing technique [8]. Generally this latent semantic vector is not a sparse vector. Whereas, the term vector and tensor component, that are generated according to the term vector [9] and the proposed tensor based techniques. These two components are similar in structure and are sparse vectors.

3.10 Criticality of execution time of tensor comparison

When two descriptors need to be compared, then their three components (LSI vector, term vector & tensor) are separately compared with their respective counter parts to generate three separate similarity values. We shall prefer to execute the similarity computation of all the three components in parallel and the longest execution time of the three will determine the similarity comparison execution time. In this scheme, the computation of dot product of latent semantic vectors can be carried in shortest time. This is because the number of basis vectors is very limited (< 1000), and the vector being non-sparse one, there is no need to search for and match the non-zero basis vectors. We can conceive a bank of 1000 fixed point multipliers, which can compute all the multiplication of 1000 coefficient pairs concurrently, followed by a 1000 input fixed point adder. Multipliers, having a latency of 5 clock cycles, and the required adders, having latency of 1 cycle, are available in the libraries of Electronic Design Automation tools. It is possible to integrate 1000 multipliers because each multiplier has small foot thermal and silicon area foot-print. The entire dot product (mult-add) computation can be completed in around 5+1 = 6 clock cycles.

On the other hand dot product of the tensor and the term vectors takes much more clock cycles than that. This means that the tensor or term vector dot product computation will determine the execution time of the three component descriptor comparison process. Therefore this dot product computation of these sparse vectors is the key problem here. Thus henceforth, we will only discuss this sparse vector dot product computation problem and this problem is the focus of the next chapter.

3.11 Summary

Systematic organization of the index entries in an index system can save number of servers necessary to build the system. This kind of index organization and operation can be facilitated by a network appliance called semantic routers which forward search queries and messages to index servers based on meanings of the messages. These routers need to compare meanings of messages and documents stored in the index system. To support such meaning comparisons we proposed a theoretical framework including a formal algebra and a design of a meaning representation and comparison technique. This design supports generative meaning composition and thus can represent and compare composite meanings. The computations needed in this comparison technique are simple and can be completed at high speed, if carried out on accelerator hardware. We will demonstrate that aspect in the next chapter.

CHAPTER IV

MEANING COMPARATOR: ARCHITECTURE

To materialize a semantic router which can route messages based on their meanings, we needed a method to represent composite meanings. A composite meaning is represented as a tensor in an infinite dimensional vector space. The similarity between two composite meanings is computed as the dot (cosine) product of the two tensors which represent these meanings. This chapter explains: the challenges in carrying out fast dot product computations at high speed, the solution approach and a high level information processing architecture for dot product computing processor and additional hardware. This chapter also presents several high level architectural alternatives for the dot product processor hardware and the associated design tradeoffs.

4.1 Equivalence between tensor and vector for comparison purpose

Once the basis vector terms of a tensor, are concatenated together they still remain as character strings, though they will now have larger lengths. These strings are indistinguishable from the basis vector term string of the term vector model, as presented in section 2.5.3 of Chapter II. Therefore the basis vector terms can be treated the same way as the ones from basic term vector model. In addition, the dot product of tensor is computed in same manner as the vector dot product computation as explained earlier. Therefore a dot product computation solution that is applicable for the tensor model is also good for the term vector model. So vector and tensors are indistinguishable for the purpose of comparison. From mathematical viewpoint a tensor is also a general form vector. Therefore, henceforth we will treat the infinite dimension space tensor and vector equally and sometimes the term “vector” or “meaning vector” will be used interchangeably with the term “tensor”. However the term “basis vectors” should not be confused with the term “vector”, because by “basis vectors” we always mean the set of orthonormal basis vectors for a given space in which a vector or tensor is defined.

4.2 The problem in dot product: Quick identification of common basis vectors

A problem arises in dot product computation when we assume an infinite dimensional vector space to represent the tensors. This infinite dimensionality of the space is required to enable tensor representation of composite meanings as proposed in last chapter. As representation of composite meaning has a very important role in semantic searching, so tackling efficient dot product computation of finite sized vectors/tensors, that has finite number of basis vectors with non-zero weights in an infinite dimensional space, is needed. This problem is also present in the term vector models. This is because there can be infinite number of terms that are possible, hence the space has to have infinite dimensions.

The dot product is computed by first identifying the basis vectors that are common to both tensors, then multiplying the coefficients of these basis vectors from their respective tensors (as explained in Fig. 2.6) and summing up all the products. Here the problem is to quickly match and identify the common basis vectors which are represented in the memory, as ASCII character strings of the corresponding terms.

To match and pair the basis vectors, it is necessary to consider each basis vector from one of the meaning vectors, and then search for its presence in the second meaning vector. Such searching is not necessary in case of small finite dimensional vector models like LSI [8]. This is because, both coefficients from the two meaning vectors can be multiplied irrespective of whether one (or both) of the coefficients are zero or not. However multiplication operation becomes superfluous when a coefficient is zero. Such superfluous operations can be carried out (without eliminating them), when their numbers are limited as in case of small finite dimensional VMs. However for infinite (or large) dimension VMs, it is necessary to eliminate such superfluous operations as their occurrences could be infinitely (or extremely) large and will be computationally very expensive. This elimination of unnecessary computations is done by identifying the matching basis vectors having non-zero coefficients, which are only worth multiplying. This requires the search operation to identify the common basis vectors, as explained above.

If the number of basis vectors in the two meaning vectors (having non zero weights) are denoted as n_1 & n_2, then when a sequential processors is used, this search task has a time complexity of $O(n_1.\log n_2)$ or $O(n_1.n_2)$ depending on whether a binary or linear search is used. However we can do this search using a Bloom Filter based algorithm, which has smaller time complexity, only ~$O(k)$, where k <20, or O(1) depending on the Bloom Filter design (details presented in section 4.5).

4.3 Hardware accelerators for dot product computation

If we can address the aforementioned “common basis vector identification” problem using a special technique then we can significantly speed-up this dot product computation. This technique will be embodied in a special purpose hardware co-processor. This co-processor can be retrofitted in existing server architectures by placing it on the PCI-Express or FSB bus. The proposed dissertation will present the information processing approach, hardware centric algorithms and high level architecture for such co-processor.

To implement the co-processor, we propose circuit level parallelization approach as opposed to coarse grained thread level parallelization one. This circuit level parallelization approach will provide several order of magnitude improvement in speed compared to the efficient software based dot product computations, even if they are executed on a multiprocessor system. We shall argue this point later in Chapter VII, with experimental data. We propose that this hardware should be used in the semantic routers and also in the index servers. When used this hardware can improve system throughput, diminish the need for load sharing by multiple servers and cut down number of servers required in the infrastructure to reduce investments, power consumption and operational costs in the data centers. This savings are in addition to the resource savings mentioned in Chapter I.

To insert the SRN in a distributed search engine and enable operations of the dot product processor hardware, we also need to make some changes in the query processor and index generator sub-systems. We mentioned in section 3.3, Chapter III, that the query processor and index generator sub-systems have to carry out an additional processing step called "step III" (refer Fig. 3.2). This step III converts the two column coefficient table with basis vector terms represented as character strings to another two column table format suitable for processing by the dot product computation in the dot product hardware placed within the semantic routers in the SRN and the index servers in the index server pools. This processing involves expensive hash value generation computation, hence it is best carried out by a co-processor hardware for speed. The co-processor hardware can be similarly retrofitted to the existing servers in the query processor and index generator sub-systems, by placing it on their PCI-Express or FSB bus. This arrangement can be viewed as if the dot product accelerator hardware has been split into two separate hardware components, and one of the components is placed in the query processor or index generator sub-system, while the other is placed in the semantic routers and index servers. The rational for this separation is explained in section 4.5.3.

4.4 Application of Bloom Filters in identifying common basis vectors

The key problem in computing dot products is identifying the common basis vectors which are present in both tensors. These common basis vectors can be identified very quickly by the following technique. The basis vectors taken from each of the two tensors are used as elements to form two separate sets. Then taking each element from the smaller set, we check whether it is a member in the other set or not. If this membership test is positive, i.e. the element is also member in the other set then that element (basis vector) is suspected to be a common basis vector.

If this membership test is perfect, which means there would be no false positives (i.e. test indicates membership even though the element is not a member) or no false negatives (i.e. the element is a member but test indicates otherwise. If the test is perfect then the number of suspected common basis vector would be also the authentic common ones. If the membership test has some false positives but no false negatives, then the aforementioned common vector identification method will yield a complete set of suspects, which includes all a complete set of authentic common basis vectors. So from this suspects a complete set of authentic common basis vectors can be filtered out. However this is not possible if the membership test gives false negatives.

When Bloom Filter are used to implement these sets, as explained in section 2.12, it is possible to carry out each membership tests inexpensively in O(1) or O(*k*) order of time depending on the Bloom Filter (BF) design (as explained in next section). With an appropriately designed BF, membership testing can have very small probability of false positives and absolutely no false negatives. Therefore this scheme using BF is a workable one. In all these cases, the value of *k* is small < 20. These techniques require *n* membership tests, where *n* is the number of elements in the smaller set. These tests can be carried out in parallel using simple hardware circuits. Thus, using this approach it is possible to identify the common basis vectors in O(*k*) or O(1) order time (refer “Bloom Filter basics”, section 2.12).

Once the suspected common basis vectors are identified, they can be used as keys to retrieve corresponding pairs of coefficient values from both the coefficient tables, which we propose to implement in a content addressable memory (CAM). Once the coefficient values are retrieved and paired, they can be multiplied and added. If the lookup fails then that means the suspect was a false positive. With a well designed BF, the probability of false positives is small, hence this scheme is workable and this reduces the search time.

4.5 Necessary data structure, algorithms and processing architecture

Here a method to compute dot product of two given tensor/vectors is explained. This computation consists of two parts. In the first part, two tensors are encoded in form two semantic descriptor data structures. This data structure is needed to enable faster dot product processing. Section 4.5.1 explains how the data structure is generated from a given tensor/vector. In the second part of the computation, two data structures of the two tensors are used to derive their dot product. This is explained in section 4.5.2. For each part of this computational model, we propose hardware centric algorithms and explain the parallelization strategy. Finally in section 4.5.4 we present how all these ideas will be consolidated in form of a processing architecture for a meaning comparator. In the following sections, we denote the abstract algorithm as “computation model”, the high level description of the hardware that carries out this computation is called as “hardware description” and the data processing work flow that is carried out in the hardware is termed as “architecture”.

4.5.1 Data structure generation

The data structure generation process for the proposed search system that was introduced earlier in Fig. 3.2, in section 3.3, Chapter III, has been presented below in Fig. 4.1.

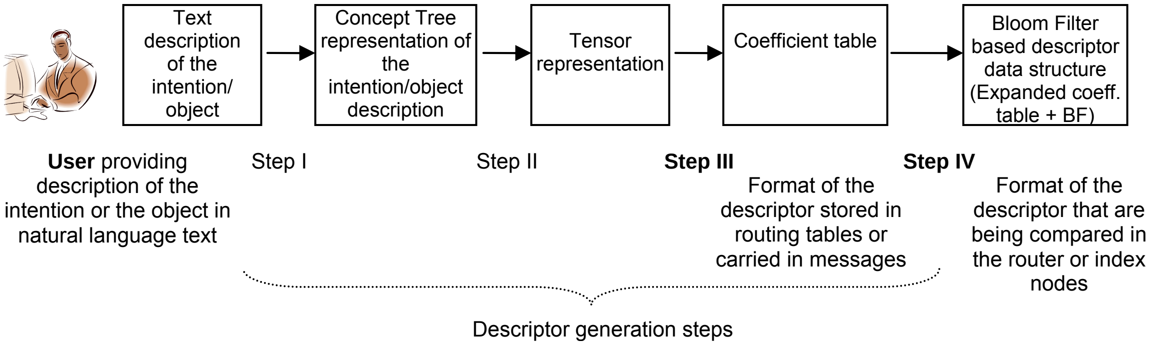

Text description of the intention/ object
Concept Tree representation of the intention/object description
Tensor representation
Coefficient table
Bloom Filter based descriptor data structure (Expanded coeff. table + BF)
User providing description of the intention or the object in natural language text
Step I
Step II
Step III
Step IV
Format of the descriptor stored in routing tables or carried in messages
Format of the descriptor that are being compared in the router or index nodes
Descriptor generation steps

Fig. 4.1 Descriptor generation process

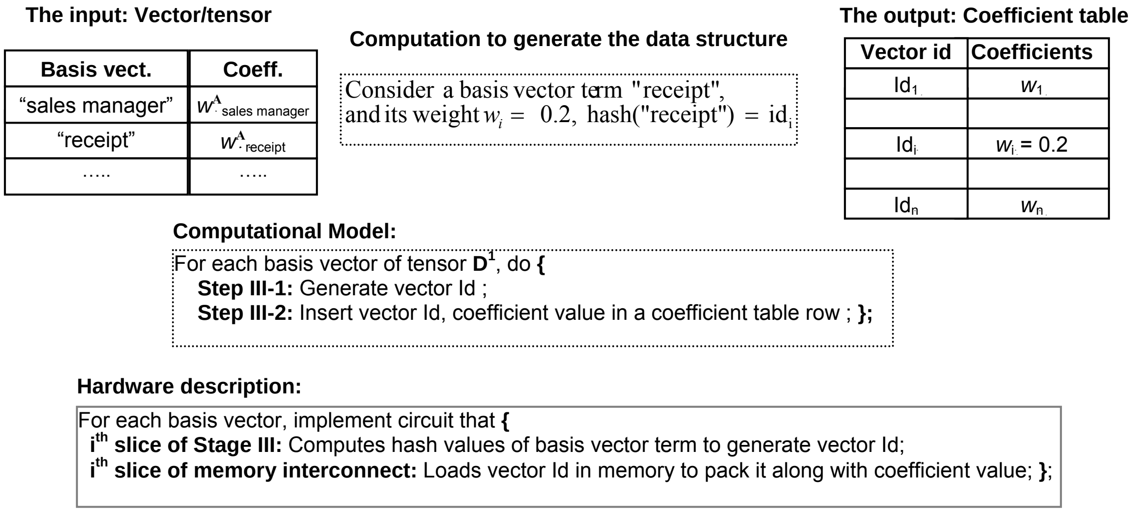
In this section we focus in step III and step IV of the process presented in the figure, where the descriptor data structure is being generated from the tensor representation. Step III generates a coefficient table having two columns, which is further processed to generate a Bloom Filter (BF) based data structure having two components: (i) an expanded coefficient table having 3 columns; and (ii) a Bloom Filter. Step III is carried out in the query processor and the index generator (refer Fig. 3.2 in section 3.3 of Chapter III), whereas step IV is carried out in the hardware comparator residing within the core of the semantic routers and index servers. Actually step IV is considered as part of the comparison process. In the next few sections we explain the algorithms and the computations in step III and IV followed by the rationale for a separate step III and IV process.

4.5.1.1 Computational model and algorithm for step III

Each row of the coefficient table consists of two columns for the following two data fields:

(1) Vector id (e.g. id_1 in Fig. 4.2); and

(2) 16 bit fixed point scalar coefficient of a basis vector (e.g. w_i);



The input: Vector/tensor

Basis vect.	Coeff.
"sales manager"	$w^{A}_{\cdot\text{sales manager}}$
"receipt"	$w^{A}_{\cdot\text{receipt}}$
…..	…..

Computation to generate the data structure

Consider a basis vector term "receipt", and its weight $w_i = 0.2$, hash("receipt") = id_i

The output: Coefficient table

Vector id	Coefficients
Id_1	w_1
Id_i	$w_i = 0.2$
Id_n	w_n

Computational Model:

```
For each basis vector of tensor D^1, do {
   Step III-1: Generate vector Id ;
   Step III-2: Insert vector Id, coefficient value in a coefficient table row ; };
```

Hardware description:

```
For each basis vector, implement circuit that {
  i^th slice of Stage III: Computes hash values of basis vector term to generate vector Id;
  i^th slice of memory interconnect: Loads vector Id in memory to pack it along with coefficient value; };
```

Fig. 4.2 Coefficient table generation

The generation of the coefficient table for the meaning vector/tensor is done in two steps (Fig. 4.2 [13], computation model). For each basis vector, the following two steps are carried out:

Step III-1: A 64 or 128 bit hash value of each basis vector term is generated. This hash value is called vector id.

Step III-2: The vector id and the coefficient value are stored in the first and second column in a row of the coefficient table.

4.5.1.2 Parallelization strategy and architecture for step III

For each basis vectors (or for each row in the coefficient table), steps III-1 and III-2 of the computation model, as shown in Fig. 4.2, can be executed in parallel, as processing of each of these basis vector terms (or rows) are independent. The parallel threads comprising of step III-1 and III-2 are short and simple computation, needing small amount of memory. This makes large scale multi-processor based parallelization unsuitable for this computation. This is because the multi-processor/core systems require significant amount of time to distribute and consolidate the threads across large number of processor cores. In addition, general purpose processors takes more time to compute simple arithmetic and logical operations compared to simple digital circuits implementing the same. For example, addition operation takes 6 clock cycles on an Intel Xeon processor, compared to 1 cycle on a hardware adder. For short computation threads, all these overheads are significant compared to the thread execution time, therefore if used, then circuit level parallelization will give a much higher order speedup compared to multiprocessor based parallelization for this step III.

To improve throughput, generally, the practice is to deploy several general processors to concurrently execute all the parallel threads. One coprocessor that computes hash values of all the basis vector terms in parallel can do this task of several general purpose processors in the same amount of time due to its higher order speedup compared to general purpose processor. Thus one single coprocessor in a single server, can do the tasks of multiple general purpose processors and can replace all of these general purpose processors and the servers that host them, each of which consumes significant power. Thus, this coprocessor can save power by several order times. Hence we prefer to use the circuit level parallelization approach. In addition, in terms of power consumption, the circuit level parallelization is also better than the general purpose processor. This is because a single co-processor using circuit level parallelization uses less power than a general purpose processor. The circuit level parallelization approach to carry out step III is described below.

A circuit is provisioned to execute each parallel thread and there are multiple such circuits. We call each of this dedicated circuit as a "slice". Therefore, each of these parallel processing instances, which can be considered as a thread, is executed by each horizontal ith slice of stage III, as shown in Fig. 4.2. There are *r* slices. For maximum parallelization we would like to use maximum number of slices in these stages as necessary, so we choose $r \approx n$, where *n* is the maximum number of rows in a coefficient table (i.e. number of basis vectors) which is estimated to be in order of 10^4 in worst case. This estimate of number of basis vectors is based on our experience with tensors for real text documents [13]. In these cases, the number of basis vectors was found to be few hundreds, which is much less than 10^4. As these circuits are simple and small, so *r* can be chosen to be in order of 10^4 as well.

4.5.1.3 Time response analysis for step III

For each basis vectors, the execution time complexity of steps III-1 and III-2 is O(*n*) because *n* hash values need to be computed and loaded on to the memory. As computation of each basis vector is independent, so each of these can be computed in parallel using *r* (~ *n*) circuits within O(*n/r*) time. For $r \approx n$, this order of time is O(1). The actual execution time of a circuit carrying out this computation would be:

$$T_{StepIII\ -exe} = {}^{n_{byte}}\!/_{u} \tag{4.1}$$

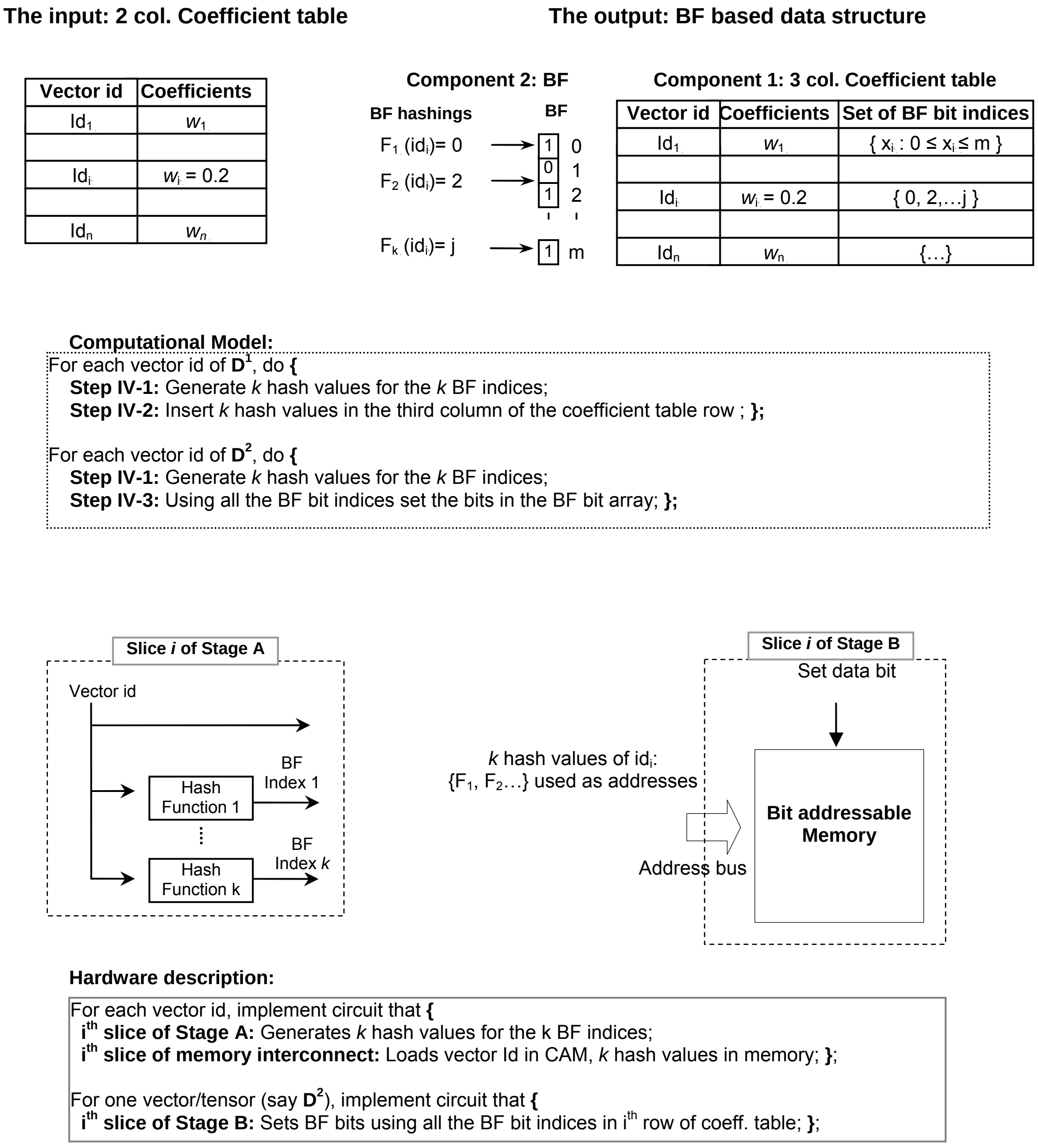
Where n_{byte} = average number of bytes in each basis vector term and *u* = loop unroll factor used to unroll loops in hash function that generate the vector id.

4.5.1.4 Computational model and algorithm for step IV

The proposed Bloom Filter based data structure for the descriptor has two components:

(1) An expanded coefficient table and

(2) A large *m* (~128K) bit long Bloom Filter (BF) using *k* (=7) hash functions (Fig. 4.3).



The input: 2 col. Coefficient table

Vector id	Coefficients
Id_1	w_1
Id_i	$w_i = 0.2$
Id_n	w_n

The output: BF based data structure

Component 2: BF

Component 1: 3 col. Coefficient table

Vector id	Coefficients	Set of BF bit indices
Id_1	w_1	$\{ x_i : 0 \le x_i \le m \}$
Id_i	$w_i = 0.2$	$\{ 0, 2, \ldots j \}$
Id_n	w_n	$\{\ldots\}$

Computational Model:

```
For each vector id of $D^1$, do {
    Step IV-1: Generate $k$ hash values for the $k$ BF indices;
    Step IV-2: Insert $k$ hash values in the third column of the coefficient table row ; };

For each vector id of $D^2$, do {
    Step IV-1: Generate $k$ hash values for the $k$ BF indices;
    Step IV-3: Using all the BF bit indices set the bits in the BF bit array; };
```

Hardware description:

```
For each vector id, implement circuit that {
  i^th slice of Stage A: Generates $k$ hash values for the k BF indices;
  i^th slice of memory interconnect: Loads vector Id in CAM, $k$ hash values in memory; };

For one vector/tensor (say $D^2$), implement circuit that {
  i^th slice of Stage B: Sets BF bits using all the BF bit indices in i^th row of coeff. table; };
```

Fig. 4.3 Bloom Filter based data structure generation

The combination of the expanded coefficient table and the bloom filter represents the vector/tensor that in turn represents the meaning of an object (text/non text document) or a query. Each row of the expanded coefficient table consists of three columns for the following three data fields:

(1) Vector id (e.g. id_1 in Fig. 4.3);

(2) 16 bit fixed point scalar coefficient of a basis vector (e.g. w_i); and

(3) Set of k BF indices, each of which have q bits, such that $m = 2^q$ (in Fig. 4.3 this is shown as: $\{x_1 : 0 \leq x_i \leq m \}$).

The generation of the expanded coefficient table for one of the tensor is done in two steps (Fig. 4.3, computation model). Here, for each vector id in the two column coefficient table generated in step III, the following two steps are carried out:

Step IV-1: A set of k BF indices is generated by further hashing each vector id by k hash functions.

Step IV-2: The BF indices are stored in the third column of the coefficient table.

For the other tensor, that has to be loaded in the BF, another additional step is carried out. This is as follows -

Step IV-3: The corresponding k bit locations are set in the BF.

Two processing stages, stage A and B, as denoted under “hardware description” in Fig. 4.3, carry out these three computation steps (steps IV-1 & IV-3). Step IV-2 is carried out by the memory interconnect and memory system. Step IV-1 of the computational model is materialized by the stage A, and step IV-3 is embodied by stage B. Concrete realization of these two stages and the notion of slices in the three stages are explained in the next section.

4.5.1.5 Parallelization strategy and architecture for step IV

For each basis vectors (or for each row in the coefficient table), steps IV-1 and IV-2 of the computation in Fig. 4.3 can be executed parallel, as processing of each of these basis vector (or rows) are independent. Therefore for the same reasons as mentioned in section 4.5.1.2, we prefer to employ circuit level parallelization to speedup this computation. To execute each parallel thread a circuit or "slice" is provisioned and there are multiple such slices. Each of these parallel processing instances is executed by each horizontal i^{th} slice of stage A, as shown in Fig. 4.3. Similar is true for the i^{th} slice of stage B. There are r slices in stage A and B. For maximum parallelization we choose $r \approx n$, where n is the maximum number of rows in a coefficient table (i.e. number of basis vectors) which is estimated to be in order of 10^4 in worst case. So r can be chosen to be in order of 10^4 as well. This is based on our experience [13].

In actual hardware, the BF bit array will be stored in the bit addressable memory and the coefficient table will be stored in a content addressable memory (CAM). The need for CAM will be explained in the following sections.

4.5.1.6 Time response analysis for step IV

For each basis vectors, the execution time complexity of stage A and B (steps IV-1, IV-2 and IV-3) is O(k) because k hash values need to be computed and loaded on to the memory. For total n_1 ($< 10^4$) basis vectors, the order of the entire data structure generation computation is O($n_1 \cdot k$). As computation of each basis vector is independent, so each of these can be computed in parallel using r ($\sim n_1$) circuits within O($n_1 \cdot k/r$) time. For $r \approx n_1$, this is O(k) with $k < 20$. If partitioned Bloom Filter is used, then this execution time is in order of O(1). This was explained in the "Bloom Filter basics" section 2.12, in Chapter II.

4.5.2 Data structure comparison

4.5.2.1 Computation model and algorithm for data structure comparison

Fig. 4.4 shows the 2nd part of the computation model and the description of the corresponding hardware. This model has six steps (steps IV-4 to IV-9) which are implemented as three processing stages (stage C to E). The step and stage numbering is continued from the first part of the model. To generate the cosine similarity value ($D^1 \bullet D^2$), the two data structures (of D^1 and D^2) are taken as inputs to step IV-4 and stage C.

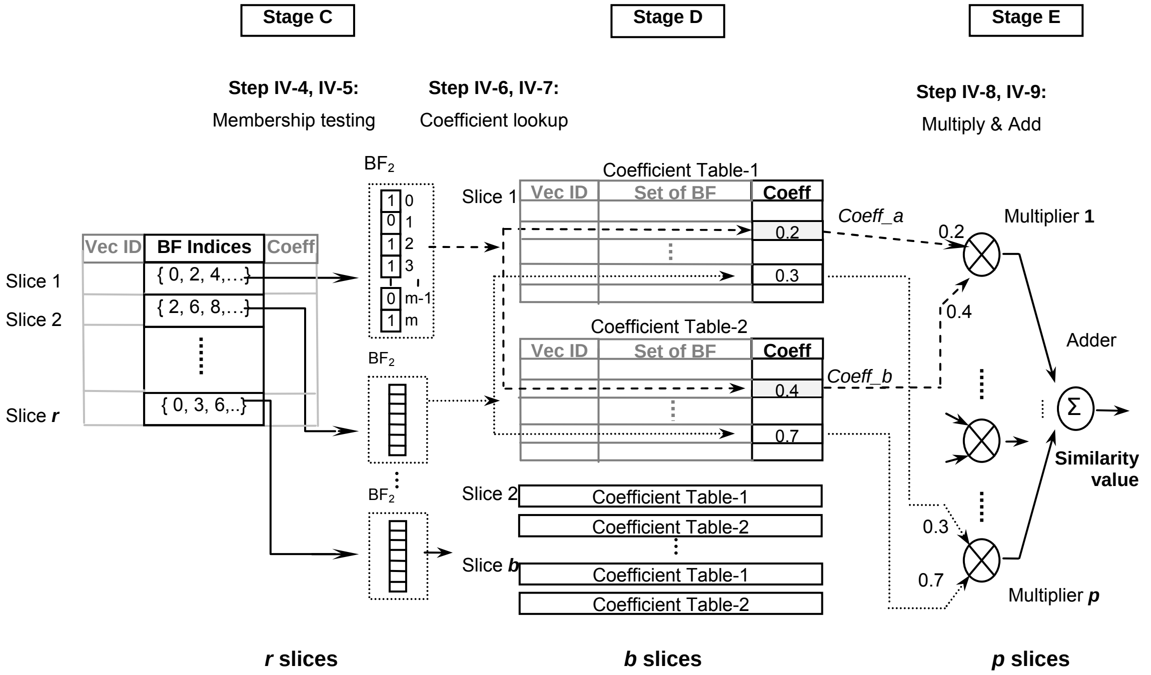
The computation steps are as follows:

Step IV-4 & IV-5 (stage C): Identify the common basis vectors from the first coefficient table (Component 1 of Fig. 4.4) by verifying which vector ids are in the second BF (BF^2, Fig. 4.4). This is carried out by Bloom Filter based membership testing technique, as explained in section 2.12 in Chapter II. The set of BF bit indices in the first coefficient table are used for this purpose.

Step IV-6 & IV-7 (stage D): If a vector is present in the BF^2 then we use that common vector id as the key to lookup, and extract the coefficient value from the coefficient lookup table of the second data structure (Coefficient Table of D^2). The coefficient table is implemented on a content addressable memory (CAM), and the coefficients are looked up in the CAM by using the vector Id as the key.

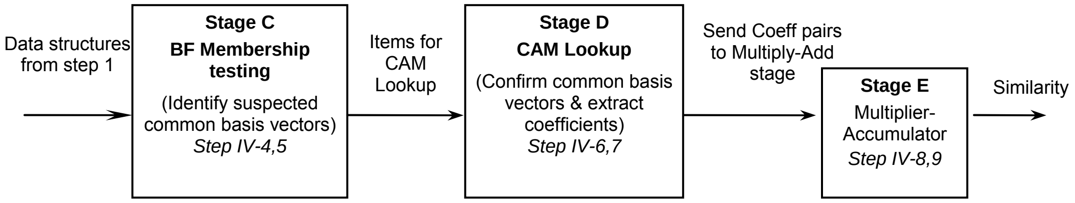
Step IV-8 & IV-9 (stage E): Multiply the pair of coefficients for each identified common basis vectors that are extracted from both coefficient tables, and then add all the products to get the similarity metric. To achieve this, the pairs of coefficients are feed to a bank of multiplier-accumulator slices.

How these stages are realized is explained in the next section.



Computational model:

Step IV-4: For each vector id in $\mathbf{D^1}$ coeff. table, test membership in $\mathbf{BF^2}$; if +ve do **{**
 Step IV-5: Note the vector in **BF** in **common vector list }**
Step IV-6: For each vector id in **common vector list do {**
 Step IV-7: Get s_i^1 & s_i^2 from $\mathbf{D^1}$ & $\mathbf{D^2}$ coeff. tables,
  Step IV-8: Add $s_i^1 \times s_i^2$ to accumulator **}**
Step IV-9: Get final cosine product $\mathbf{D^1 \bullet D^2}$ from accumulator;


Data structures from step 1
Stage C
BF Membership testing
(Identify suspected common basis vectors)
Step IV-4,5
Items for CAM Lookup
Stage D
CAM Lookup
(Confirm common basis vectors & extract coefficients)
Step IV-6,7
Send Coeff pairs to Multiply-Add stage
Stage E
Multiplier-Accumulator
Step IV-8,9
Similarity

Hardware description:

For each basis vector, implement circuit that **{**
 ith slice of Stage C: Carries out BF Membership testing ;
 ith slice of Stage D: If the testing is +ve, extracts coefficient values from CAM ;
 ith slice of Stage E: Multiplies coefficient pairs from $\mathbf{D^1}$ & $\mathbf{D^2}$ and accumulates; **}**;

Fig. 4.4 Comparison of descriptors

4.5.2.2 Parallelization strategy and architecture for data structure comparison

The roles of the three stages, that embody the second part of the computation, are as follows. Stage C identifies the common basis vectors using BF membership testing functionality, stage D extracts the matching pairs of scalar coefficients from the coefficient table; and stage E multiplies the corresponding pairs of scalar coefficients and calculates the sum to obtain the dot product.

For each row in the coefficient table, steps IV-4 and IV-9 of the computation in Fig. 4.4 can be executed parallel, as processing of each of these rows are independent. Each of these parallel processing instances is executed by each horizontal slice of each stage (C to E), as shown in Fig. 4.4. There are *r* slices in stage C, and *b* slices in stage D and *p* slices in stage E. Stage F also consolidates all the processing. The choice of *r* has been explained earlier, the choice of *b* and *p* are explained in next section.

The operations of these three stages are explained as follows. The basis vectors from the first coefficient table are membership tested against the BF of the second table in step IV-4. The BF is implemented as bit addressable memory of *m* bits. Membership test is performed by checking whether all *k* given locations contains 1 or not, as explained in section 2.12, in Chapter II. The membership testing of each vector Id (row) from the first table (step IV-4, Fig. 4.4) is carried out in parallel by dedicating one logic circuit per row of the first table in a specialized hardware (Fig. 4.4), within $\lceil n/r \rceil \cdot k$ read cycles with *r* circuits for *n* basis vectors or rows in the coefficient table. This is because each membership testing requires *k* memory read cycles and $\lceil n/r \rceil$ membership testing cycles are needed.

The second coefficient table is stored in a combined CAM-RAM unit. The vector Id column is stored in the CAM and the coefficient column is stored in the RAM in a paired manner. Whenever a vector Id for a particular coefficient table row is stored at a particular address location of the CAM, the corresponding coefficient value is stored at the same address location in the RAM. This enables extraction of the coefficient value by using the vector id as the lookup key. A vector Id is used to ascertain whether there is match in the CAM. If there is a match, then the CAM outputs the location address, in which the match happened. Then using this address, the coefficient value is extracted from the RAM.

The operation of this entire mechanism is as follows. The basis vectors that tests positive are the suspected common basis vectors. These suspects are either confirmed or rejected (step IV-6 in Fig. 4.4) by using the positive tested vector Ids as keys and performing a lookup on the second table stored in a content addressable memory (CAM). Once a match is confirmed, the corresponding coefficients from the second table, which is stored in a random access memory (RAM), are extracted (step IV-7 in Fig. 4.4). The logic signal routing by interconnects are analogous to step IV-2 & IV-3 of the pseudo-code in Fig. 4.4.

The number of positives generated by the membership testing is ($c + (n\text{-}c)\cdot P_{false+ve}$), of which "$c$" are true positives and $(n\text{-}c)\cdot P_{false+ve}$ are false ones, when c = number of basis vectors that are common (called *common basis vectors*) in both tables and $P_{false+ve}$ = probability of false positives during the BF membership testing. Therefore a total ($c + (n\text{-}c)\cdot P_{false+ve}$) suspects are identified and each of these suspects needs to be verified by using the CAM-RAM units. Therefore the number of CAM lookups (step IV-7) that are necessary is also ($c + (n\text{-}c)\cdot P_{false+ve}$). The value of $P_{false+ve}$ is kept very small << 1 (e.g., in order of 10^{-3}) by proper choice of m and k (refer section 2.12 in Chapter II).

The use of Bloom Filter (BF) reduces the number of CAM lookups and the number of the CAM banks that are necessary, from n to (c + (n-c)·$P_{false+ve}$). This value of the term (c + (n-c)·$P_{false+ve}$) is very similar to that of c, therefore by employing the BF we are saving CAM-RAM units that are necessary for parallelization and yet getting a good time savings. The CAM lookup subsequently will identify c number of common basis vectors (and reject the false positives). The corresponding coefficients will be passed to the multiplier-adder stage (Fig. 4.4). Thus this CAM lookup operation filters away the false positives and thereby avoids any incorrectness in final result and yet avoids n lookups to save time and CAM resources (less CAM can be used).

4.5.2.3 Choice of architectural parameters and rationale

To conserve hardware resources and keep the power consumption (and generation of heat in the circuit) manageable, we will choose b and p to be much smaller than n (< 10^4). We will in fact choose b to be less than 25% of n and choose p to be at most 16 as multipliers are expensive in terms of gate counts and silicon die area. For example, for $n \sim 10^4$, we chose b to be 32 and p to be 16. We show below that these choices are sufficient. Later in Chapter VII, we will also show that these smaller choices for b and p have not increased the processing time significantly. Actually for this choice of parameter b = 32, we can use a single multi-port CAM and RAM, as presented in [140], [141] to implement all the b slices in stage D.

The rationale for choosing b and p values, are as follows. If we assume that probability of a query being mapped (or routed) to a particular index server pool is equally distributed among all the N_P pools, then for large N_P (~1000), this probability is 1/N_P is small = 1/1000. For this scale of distributed system (with N_P ~1000), only one semantic router is necessary because a semantic router can very well accommodate 1000 destination entries in its semantic routing table. We will substantiate this aspect in Chapter VII. Numbers of destination lookups necessary on this routing table will require at most N_P meaning similarity comparisons.

In general, a query is expected to be matched against only one destination's key and forwarded to that destination. This means that in worst case, a query will be found similar to only one of the descriptor keys out of N_P keys in the semantic routing table. In worst possible case, for the comparison which is a match, this similarity may be a 100% match (yielding a dot product value of 1 when the c value is 100%). This means, out of N_P comparisons, only one comparison will have a c value of at most 100%. Hence the average c value for all the N_P comparisons considered together, will be 100*1/ N_P %, which is ~0.1 % (or 0.001).

The expected (average) number of CAM lookup that will take place is given $(c + (n - c)\cdot P_{false+ve})$, where $P_{false+ve}$ (~ 10^{-3}, refer section 2.12) is the probability of false positive for the large Bloom Filter being used (m ~131072, k ~ 7, refer section 2.12). In this case the expected (average) number of CAM lookups that are necessary is $(c + (n - c)\cdot P_{false+ve})$, which is in order of ~$c$ = 10. Therefore a small number of b (~32) slices in stage D and p (~16) slices in stage E will suffice.

CAM and RAM are the most power hungry components. From our hardware design experience [131], we know that the ratio of power consumption between CAM and rest of the logic circuitry is in order of 50:1, which means CAM has a disproportionate contribution in the total power draw. Thus there is a motivation to reduce the number of CAM units to cut down the total power consumption. When b, the number of CAM units, is chosen to be small, there is no point to keep a large number of multipliers, hence p, the number of multiplier circuits is also chosen to be small. However as we chose a b = 32, because a 32 port CAM design is readily available [140]. A higher value of 32 instead of 10, which equates to a utilization value of 0.32, helps in significantly reducing the average waiting time of the CAM lookup tasks compared to a utilization ~1 which causes more waiting time (refer queuing theory in section 2.11, Chapter II). Therefore this choice of b = 32 and p = 16, will reduce the CAM and multiplier circuitry, while still yielding the benefit of parallelization.

4.5.2.4 Time response analysis for data structure comparison

We have shown that the number of clock cycles needed to carry out CAM lookups is given as: $\lceil (c+(n-c)\cdot P_{false+ve})/b \rceil$, when there are b available CAM banks (copies) or a b ports in the CAM. Similarly c multiplications take time in order of $\lceil c/p \rceil$, with p available pipelined multipliers each having retiring rate of 1 (step IV-8, IV-9). The probability of false positive in Bloom Filters is approximated by a standard function as given in section 2.12 in Chapter II. A more accurate approximation will include contribution of different hash collisions. This is discussed in more details later in section 4.5.5.6.

For each basis vectors the complexity of step IV-4, IV-5 is O(k) or O(1) depending on the type of Bloom Filter used. The time complexity of step IV-6 & IV-7 is O(1) and step IV-5 & IV-6 is O(1). Therefore for n basis vectors the order of the entire algorithm is O($n_1 \cdot k$) or O(n_1) depending on the BF design. The computation of each basis vector being independent, a parallel computation using r (~n_1) circuits has time complexity of O($n_1 \cdot k/r$) or O(n_1/r), which is O(k) or O(1) for $r \approx n$. In the next section we explain the rationale for a separate step III and IV.

4.5.3 Rationale for separate step III and IV

The two column version of the coefficient table is used for storage whereas the expanded three column version is used for comparison processing. Step III generates the two column version of the coefficient table that contains the minimal information necessary for efficient storage and transmission. This coefficient table, containing 64 or 128 bit vector ids, takes much less space than a table which would have contained character strings of the basis vector terms as shown at the top left-hand side of Fig. 4.2. This is explained as follows. On average, a term in the English lexicon has 7 characters (i.e. 56 bits). The basis vector terms in a tensor are concatenated combinations of multiple English language terms. These concatenated terms have a length which is several multiples (3 or more) of 7 characters (refer section 3.6.6 in Chapter III). So an average length of a tensor basis vector term is around 210 bits or more, which greater than 64 or 128 bits. Therefore a coefficient table containing vector ids takes smaller space and thus suitable for storage and transmission. This table may be further compacted by compaction techniques.

On the other hand the expanded three column version of the coefficient table is required to carry out high speed comparison operation. This three column table does not carry any additional information compared to the two column version and this three column version is generated from the two column table only when a comparison has to be performed.

4.5.4 Comparator architecture and its execution time analysis

4.5.4.1 The architecture

The entire processing architecture, with both parts of the computation are put together, i.e., all the six stages, stage A to E, is presented in Fig. 4.5.

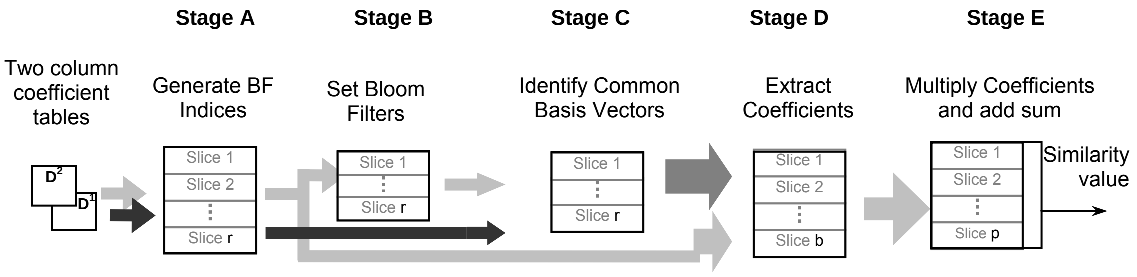


Fig. 4.5 Proposed information processing architecture

To facilitate smooth flow of data between the unequal number of slices in different stages and schedule the utilization of slices, special interconnects have to be used between the stages (Fig. 4.5). For example, the interconnect between stage D and stage E schedules processing of larger number of multiplications (*b* threads) necessary to smaller number (*p*) of available multipliers (slices). Interconnect between stage C and D plays a similar scheduling role, whereas interconnect between stage B and C distributes a single piece of data item (the Bloom Filter content) to *n* slices in stage C. Hardware design of such interconnects and schedulers are based on simple digital logic and it is outside the scope of this dissertation.

4.5.4.2 Operations

During semantic routing table lookup, the message key is compared with multiple keys belonging to the routing table rows. Therefore, we would consider a system having *n* number of comparator hardware, where *n* is the number of routing table rows. In each comparator, a particular row key is considered as first descriptor whose Bloom Filter component is loaded in the Bloom Filter (BF) RAM. The message keys will be passed through the comparator one by one and the comparison values that come out of the comparator will be noted. Therefore the row key will be loaded only once in the comparator to set it up. This will avoid multiple setup times involved in loading the BF RAM multiple times. This paradigm is explained in greater details in section 6.7.2, in Chapter VI.

4.5.4.3 Execution time analysis

For purpose of execution time analysis of the comparison operation, the critical path is through the following stages: stage A→stage C→stage D→stage E. Thus the execution time is the sum of cycles required by each stages: A, C, D, E. Whereas, the setup time is the time necessary to setup the Bloom Filter, which is done by stage B, thus the setup time is time taken to complete processing in stage B.

When real hash functions are utilized to generate the *k* BF index values, then number of clock cycles T_A taken by stage A is given by the following equation.

$$T_A = {n_{byte}} / {u} \tag{4.2}$$

Where n_{byte} = number of bytes in a vector id and u = loop unroll factor used to unroll loops in hash functions that generate the *k* BF bit indices.

The total number of clock cycles $T_{C\text{-}E}$ taken by stage C to E is given by the following equations. The delay component of each stage is also indicated below each equation.

When unpartitioned Bloom Filter (as discussed in Chapter II), is used in stage C, this is-

$$T_{C-E} = \underbrace{1}_{\text{Interconnect delay}} + \underbrace{\left\lceil n / r \right\rceil \cdot k}_{\text{Step C delay}} + \underbrace{1}_{\text{Interconnect delay}} + \underbrace{\left\lceil \left(c + (n-c)\cdot P_{false\ +ve}\right) / b \right\rceil}_{\text{Stage D delay}} + \underbrace{1}_{\text{Interconnect delay}} + \underbrace{\left\lceil c / p \right\rceil + L + A}_{\text{Stage E delay}} \tag{4.3}$$

When partitioned Bloom Filter is used in stage C, this is-

$$T_{C-E} = \underbrace{1}_{\text{Interconnect delay}} + \underbrace{\left\lceil n / r \right\rceil}_{\text{Step C delay}} + \underbrace{1}_{\text{Interconnect delay}} + \underbrace{\left\lceil \left(c + (n-c)\cdot P_{false\ +ve}\right) / b \right\rceil}_{\text{Stage D delay}} + \underbrace{1}_{\text{Interconnect delay}} + \underbrace{\left\lceil c / p \right\rceil + L + A}_{\text{Stage E delay}} \tag{4.4}$$

where L = Latency of each multiplier = 5 and A = Latency of the adder =1. The interconnect delays are based on the hardware design carried out by [131].

Thus the total execution time is the time taken by stages A, C, D and E and this is given by:

$$T_{exe} = T_A + T_{C-E} \tag{4.5}$$

whereas the setup time is given as the time needed to complete operation in stage B. When unpartitioned Bloom Filter is used in stage B, this is -

$$T_{setup} = T_B = k + D \tag{4.6}$$

where D = clock cycles required by the interconnect to load the Bloom Filter and this D ~10.

When partitioned Bloom Filter is used, the setup time is -

$$T_{setup} = T_B = 1 + D \tag{4.6}$$

4.5.5 Techniques to reduce circuit complexity and circuit power

4.5.5.1 Motivation

To reduce the need for silicon die area and lower the power consumption, we strive for simpler circuits. By choosing appropriate algorithms and computational models we can reduce the circuit complexity and thereby reduce power consumption because a simpler circuit draws less power. Simpler circuit and lower power consumption allows us to integrate a higher number of slices to achieve greater degree of parallelization. We discuss some of the approaches below to achieve this.

4.5.5.2 Approaches

Circuit complexity of the comparator hardware can be achieved by three following approaches:

1) *By using smaller number of CAM units and multipliers*: Contribution of smaller number of CAM and multiplier units had been already discussed earlier in section 4.5.2.3.

2) *By adopting a smaller Bloom Filter bit array size*: A smaller BF size decreases the amount of memory required. What should be the size of the BF in terms of number of BF bit array *m* and number of hash functions *k*, depends on many factors. We choose to work with *m* = 131072 and *k* = 7. In Chapter VII we present the evaluation whether these are good values, i.e. whether these choices gives a small execution time. However which would be a better choice is best determined based on actual performance results of the entire comparator architecture. This is requires a simulation based study, which is presented in Chapter VII.

3) *By implementing the hash value generation circuitry by cheaper alternatives*: We need *k* hash values as the *k* BF bit indexes, hence in total *k* distinct hash functions are required. Hash function implementation in hardware is expensive in terms of circuit complexity, silicon die space and power consumption. So as a solution, we propose alternative computation schemes that will use only one or two hash function implementation (circuits) to generate all the *k* hash values that are necessary to operate the Bloom Filter. Various combinations of these schemes will yield a variety of architectures. These schemes will be implemented in stage A of the proposed architecture (Fig. 4.5). Cheaper alternatives to generate hash value, is discussed in details in the next section.

The performance tradeoffs of using these approaches are analyzed in section 4.5.5.6.

4.5.5.3 Inexpensive methods to generate hash values

We need *k* different hash values to generate *k* different *q* bit Bloom Filter index values, where *m* = 2^q. This is done in two steps, first 64 bit hash values are generated, next from those 64 bit values, corresponding *q* (=17) bit hash values are derived. There are two opportunities to reduce circuit complexity here. We can choose a set of alternative circuit reduction strategies to generate 64 bit values and in addition we can also choose another set of alternative strategies to generate *q* bit values from 64 bit hash values. We can combine both these set of strategies together. This aspect is explained later once we have described these strategies.

Generation of 64 bit hash values is possible by two alternative classes of strategies. One of the class use two distinct hash functions to generate the required *k* hash values and the other class use only one distinct hash function to generate *k* values. The "two hash function" class of strategies is given by equations (4.8) to (4.11). The "one hash function" class is given by two different set of strategies. One set of strategies require simultaneous use of any one of these equations (4.8) to (4.11) along with any one of the equations (4.15) to (4.16) to generate 64 bit hash values. Whereas, the second kind of one hash function strategy will involve using only equation (4.17). Once a 64 bit hash value is generated, from it a *q* bit hash value is generated using any one of these strategies given by equations (4.12) to (4.14). All these strategies are described below.

This set of *k* values can be generated by only two distinct hash functions H_1 and H_2 by the following mechanism:

The ith hash value is given by-

$$h_i = H_1 + i * H_2 \tag{4.8}$$

where *i* is an integer : 0, 1, 2, 3,….

Using this method [142] we can generate *k* hash functions without actually implementing distinct *k* hash value generating circuitry. An integer multiplier from the hardware design compiler library is expected to be an optimized and simpler circuit than a manually designed hash function circuit, hence this strategy can reduce circuit complexity.

Another set of alternatives can be as follows. Instead of multiplying the integer *i* in equation (4.8), we can simply rotate the H_2 by *i* bit positions and then add it to H_1. In that case the ith hash value is given by-

$$h_i = H_1 + rot\,(H_2, i) \tag{4.9}$$

where *i* is an integer : 0, 1, 2, 3,….

As rotation can be materialized by re-routing bit lines so this rotations does not increase circuit complexity by adding active devices, so this strategy (equation (4.9)) further reduce circuit complexity compared to earlier method as shown by equation (4.8).

Another alternative generation function can be chosen to further reduce circuit complexity, where the i^{th} hash value is given by-

$$h_i = H_1 + 2^i * H_2 \tag{4.10}$$

where *i* is an integer: 0, 1, 2, 3,….

This avoids the need for a multiplication altogether, just bit shifting the H_2 hash value to the left would suffice. The bit shifting or rotating does not need any active circuitry, just re-routing bit logic lines can achieve the bit shifting or rotation. Therefore, the circuit complexity to implement this technique as given by equation (4.10), is even less than the case for equations (4.8) and (4.9).

Instead of adding we can even take bit wise XOR, in that case the ith hash value is given by-

$$h_i = bitwiseXOR\,(H_1, rot(H_2, i)) \qquad (4.11)$$

where *i* is an integer : 0, 1, 2, 3,….

As XOR gates are simple circuits than bit address circuitry, therefore the circuit complexity for equation (4.11) is even less than the case for equation (4.9).

There is another place where the circuit can be simplified by choosing proper computational alternative to generate the BF bit indexes. In our Bloom Filter we need 17 bit hash values, so we generate 17 bit values from 64 bit hash values by different alternative method. This can be done either by taking only 17 LSB bits, or XORing the 17 bits chunks of higher bits to the lower 17 bits, or by bit folding technique proposed by the FNV hash algorithm [138]. This folding is necessary to satisfy the strict avalanche criterion of minimal collision hash value generation (refer section 2.13 in Chapter II). These alternatives are presented below:

$$H^{17} = fold\,(H^{64}) \qquad (4.12)$$

where the *fold( )* function denotes the folding of higher 17 bit chunks to the lower 17 bits.

$$H^{17} = FNVfold\,(H^{64}) \qquad (4.13)$$

where the *FNVfold()* function denotes the folding of higher order bits as suggested by FNV hash algorithm.

The two alternatives, as presented above, appear to be similar in terms of circuit complexity, and they are expected to cause similar BF performance. Another more simpler alternative to achieve the same purpose, is presented below.

$$H^{17} = Lsb(H^{64}) \tag{4.14}$$

where function *Lsb( )* considers the lower 17 bits only.

This third alternative is the simplest of all, however one needs to examine whether deterioration of hash value property caused by discarding MSB bits causes in significant deterioration of Bloom Filter (BF) operation, like increase of false positive probability. This may happen because the hash value generated by this operation, as presented above, does not satisfy the strict avalanche criterion. In Chapter VII, from simulation results, we shall see that this deterioration is so small that it is not significant.

Additional reduction of circuit complexity is possible during 64 bit hash value generation. In fact one can avoid the use of the second hash function H_2 all together in the equations (4.8) to (4.11). The second hash value H_2 can be generated from the first hash value H_1 by either rotating the bits of H_1 by some arbitrary fixed places or by scrambling them. Scrambling or rotating does not cost any circuit space, because bit logic lines can be suitably routed to inter-change the bit positions. These two methods are represented as follows:

$$H_2^{'} = rot\ (H_1, a) \tag{4.15}$$

where *a* denotes the constant number of bit positions either rotated left or right.

$$H_2^{'} = \mathrm{s}cram\ (H_1) \tag{4.16}$$

Alternately we can directly generate the k hash values in a single step, by scrambling the bit positions of the first hash function in k different ways, without needing to generate the intermediate second hash function. In this method as the second hash function is not generated, hence it is not necessary to combine the first and second hash functions. In this case the ith hash value is given by-

$$h_i = scram_i(H_1) \tag{4.17}$$

where $scram_i$( ) denotes the scrambling of bit positions in a ith manner.

An example of a "two hash function and q bit hash value generation" strategy would be simultaneous use the circuit which implements the computation as mentioned by two equations (4.8) and (4.12). On the other hand a "one hash function and q bit hash value generation" strategy would be simultaneous use circuits that implement the computations as given by three equations (4.8), (4.15) and (4.12). Similarly, an "alternate one hash function and q bit hash value generation" strategy would be to carry out computations as given by equation (4.17) along with equation (4.12).

4.5.5.4 Contribution of different methods to reduce circuit complexity

The hash function generation circuitry will contribute most of the circuit complexity, followed by 64 to 17 bit hash value conversion circuitry (as given by equations (4.12), (4.13) and (4.14)) and hash value composition circuitry (as given by equations (4.8), (4.9) and (4.11)). This is because hash function generation requires complex sequential logic circuitry, whereas 64 to 17 bit hash value conversion and hash value composition involves simpler combinational logic circuits. However the alternate one hash function strategy, i.e. using equation (4.17) along with either one of the equations (4.12) to (4.14), would generate the greatest savings in circuit complexity, because there is no need for any circuit to combine the first and second hash functions.

4.5.5.5 Alternative architectures

The corresponding architectures which exploit the techniques described above are presented below in Table 4.1.

Table 4.1 Alternative architectures due to variation in stage A

Sl.	**# Distinct hash values**	**Stage A**			**Stage B**	**Comple -xity rank**
		Second hash number generation, if any	**i^{th} Hash value generation**	**64 bit to 17 bit conversion**	**Type of BF**	
1	k + 1	-	-	$Lsb(H^{64})$	unpartitioned	D
2	k + 1	-	-	$Lsb(H^{64})$	partitioned	D
3	2	Distinct H_2	$bitwiseXOR(H_1, rot(H_2, i))$	$FNVfold(H^{64})$	unpartitioned	C6
4	2	Distinct H_2	$H_1 + i*H_2$	$fold(H^{64})$	unpartitioned	C5
5	2	Distinct H_2	$H_1 + rot(H_2, i)$	$fold(H^{64})$	unpartitioned	C4
6	2	Distinct H_2	$H_1 + 2^i * H_2$	$fold(H^{64})$	unpartitioned	C3
7	2	Distinct H_2	$bitwiseXOR(H_1, rot(H_2, i))$	$fold(H^{64})$	unpartitioned	C2
8	2	Distinct H_2	$H_1 + i*H_2$	$Lsb(H^{64})$	unpartitioned	C1
9	2	Distinct H_2	$H_1 + i*H_2$	$Lsb(H^{64})$	partitioned	C1
10	1	$H'_2 = scram(H_1)$	$bitwiseXOR(H_1, rot(H_2, i))$	$fold(H^{64})$	unpartitioned	B
11	1	$H'_2 = rot(H_1,a)$	$bitwiseXOR(H_1, rot(H_2, i))$	$FNVfold(H^{64})$	unpartitioned	A6
12	1	$H'_2 = rot(H_1,a)$	$H_1 + i*H_2$	$fold(H^{64})$	unpartitioned	A5
13	1	$H'_2 = rot(H_1,a)$	$H_1 + rot(H_2, i)$	$fold(H^{64})$	unpartitioned	A4
14	1	$H'_2 = rot(H_1,a)$	$H_1 + 2^i * H_2$	$fold(H^{64})$	unpartitioned	A3
15	1	$H'_2 = rot(H_1,a)$	$bitwiseXOR(H_1, rot(H_2, i))$	$fold(H^{64})$	unpartitioned	A2
16	1	-	$scram_i (H_1)$	$Lsb(H^{64})$	partitioned	A1

We will evaluate all these alternative architectures. These architectures have been assigned ranks solely based on circuit complexity: A1 means least complex and most preferable, followed by A2 and so on. The lowest rank D is least preferable. These rankings were carried out taking into consideration each stage’s relative contribution to the circuit complexity as explained in the previous section.

4.5.5.6 Performance tradeoff analysis for alternative architectures

We matched the output of the simulator with the dot product numbers generated by a direct multiply-add method as indicated in section 2.5.3 of Chapter II. The output of the simulator was found to be correct. However there is an extremely small probability that the proposed comparator will generate incorrect values. The analysis of such situation is presented below.

There are three indeterminsitic phenomena which are at play here. These affect the correctness and execution time of the tensor comparison operation as carried out by the proposed architecture. The first one is the hash collision phenomena (“phenomena 1”) that happen when vector ids are generated in step III as shown in Fig. 3.2 in section 3.3 of Chapter III. When this hash collision happens then, even though the basis vector terms are different they generate same vector ids. The second phenomena (“phenomena 2”) is the generation of the same set of BF index values taking place in step IV-1 in stage A of the architecture, even when the vector id are different. The conditional probability of that happening is $(2^{-q})^k$, because *k* distinct *q* (=17) bit BF bit index values are being generated. The third one (“phenomena 3”) is the BF false positive phenomena whose conditional probability is described earlier in section 2.12 of Chapter II. The event tree for all these three phenomena is shown in Fig. 4.6 along with conditional and unconditional probabilities of all events.

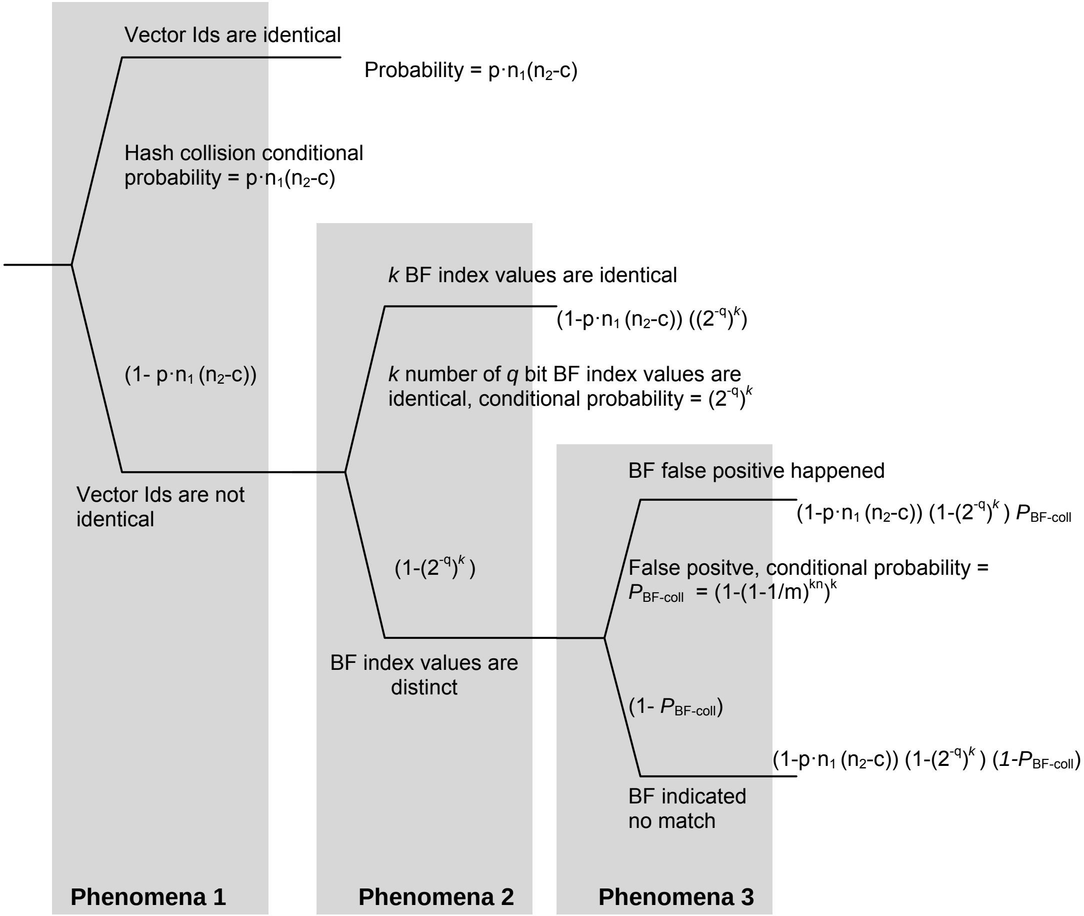

Vector Ids are identical
Probability = $p \cdot n_1(n_2\text{-}c)$
Hash collision conditional probability = $p \cdot n_1(n_2\text{-}c)$
$(1\text{-}\ p \cdot n_1\,(n_2\text{-}c))$
Vector Ids are not identical
k BF index values are identical
$(1\text{-}p \cdot n_1\,(n_2\text{-}c))\ ((2^{-q})^k)$
k number of q bit BF index values are identical, conditional probability = $(2^{-q})^k$
$(1\text{-}(2^{-q})^k\,)$
BF index values are distinct
BF false positive happened
$(1\text{-}p \cdot n_1\,(n_2\text{-}c))\ (1\text{-}(2^{-q})^k\,)\ P_{BF\text{-}coll}$
False positve, conditional probability = $P_{BF\text{-}coll}\ = (1\text{-}(1\text{-}1/m)^{kn})^k$
$(1\text{-}\ P_{BF\text{-}coll})$
$(1\text{-}p \cdot n_1\,(n_2\text{-}c))\ (1\text{-}(2^{-q})^k\,)\ (1\text{-}P_{BF\text{-}coll})$
BF indicated no match
Phenomena 1
Phenomena 2
Phenomena 3

Fig. 4.6 Event tree for the proposed architecture

The false positives in the Bloom Filter either due to any of phenomena 2 or 3 in stage C, gets filtered out by the CAM lookups in stage D. Thus, phenomena 2 and 3 do not have any impact on correctness. These two phenomena leads to false positives in the BF operations which only increases the number of CAM lookups that are necessary and the execution time for CAM lookups in stage D. The proposed dot product computation algorithm will yield wrong values only if there is a hash collision during vector id generation (phenomena 1) in stage A. The probability of that happening is in order of $p \cdot n_1 \cdot (n_2\ \text{-}c)$, where p = the hash collision probability and n_1, n_2 = number of basis vectors in each tensor, c = number of common basis vectors.

With a good quality 64 bit hash function, that satisfies strict avalanche criterion (refer section 2.13, Chapter II), this collision probability $p \approx 2^{-64}$. For $n_1, n_2 < 10^4$, wrong values are very unlikely (probability in order of ~10^{-11} or less). The numerical impact of one of such collision is also restricted because there is a limited contribution of a single basis vector in the dot product computation. This likelihood of hash collision can be reduced by choosing a 128 bit hash vector id if necessary.

When k distinct hash functions and q (=17) bit BF indexes are used then the false positive probability is due to the combined effect of phenomena 2 and 3, is approximately given by the following equation. This is a more accurate model compared to the one that can be derived solely based on the equation (2.5) in section 2.12 of Chapter II.

$$P_{false+ve} = \left(1 - p \cdot n_1\left(n_2 - c\right)\right)\left(2^{-q \cdot k}\right) + \left(1 - p \cdot n_1\left(n_2 - c\right)\right)\left(1 - 2^{-q \cdot k}\right)\left(1 - \left(1/m\right)^{k \cdot n}\right)^k \qquad (4.18)$$

Where p = collision probability during vector id generation ~2^{-64}, n_1, n_2 = number of basis vectors in each tensor, c = number of common basis vectors in these tensors, q = number of bits in the BF index, k = number of hash functions used in BF operation, m = size of BF bit array.

With $p \approx 2^{-64}$, $n_1, n_2 = 10^4$, $q = 17$, $k = 7$, $c = 1\%$ of n_1, and $m = 131072$, this false positive probability is 0.002077.

When the circuit complexity reduction strategies as given by equations (4.8), (4.9), (4.10) and (4.11), are used instead of k distinct hash functions in the BF, then the probability of false positive is higher than the one given by equation (4.18). For all these cases, this probability is approximately given by the following equations:

$$P_{false+ve} = \left(1 - p \cdot n_1 \left(n_2 - c\right)\right)\left(2^{-2q}\right) + \left(1 - p \cdot n_1 \left(n_2 - c\right)\right)\left(1 - 2^{-2q}\right)\left(1 - \left(1/m\right)^{k \cdot n}\right)^k \qquad (4.19)$$

Whereas, when the strategy as given by equation (4.15) or (4.16), is used as alternatives to distinct *k* hash functions to generate BF bit index values, then the probability of false positive is further increased compared to the one given by equation (4.19). This probability is approximately given by the following equation:

$$P_{false+ve} = \left(1 - p \cdot n_1 \left(n_2 - c\right)\right)\left(2^{-q}\right) + \left(1 - p \cdot n_1 \left(n_2 - c\right)\right)\left(1 - 2^{-q}\right)\left(1 - \left(1/m\right)^{k \cdot n}\right)^k \qquad (4.20)$$

This means that when we use inexpensive alternatives to distinct hash functions, we get a higher number of suspects which need larger number of CAM lookups and take more execution time. For this case, the probability of false positive as given by equation (4.20) is 0.00208 for the same example that was used earlier to show the value of 0.002077 according to equation (4.17). We see that this increase predicted by these equations is small. In Chapter VII, we will see that this increase in execution time is insignificant compared to reduction in circuit complexity that is achieved.

Use of one hash function, as given by equation (4.17), saves execution time in stage A, because these strategies only involve re-routing of bit lines and nothing more. As these does not involve any extra clock cycles, therefore this execution time is zero and it is given as:

$$T_A = 0 \qquad (4.19)$$

This execution time of stage A in all other cases was given by equation (4.2). So the savings in time due to use of one hash function is this amount that is given by equation (4.2). When the number of bytes in the vector id is 8 (=64 bits) and there is no loop unrolling, i.e. loop unrolling factor l = 1, then this savings = 8/1 = 8 clock cycles, which is modest. However due to increase in false positives in stage C, the execution time will increase by a few clock cycles, so the net savings in execution time will be little smaller that what is predicted by equation (4.19). In Chapter VII, we will see that use of one hash function results in a modest decrease in execution time. This is in addition to the significant reduction in circuit complexity.

The impact of using lower 17 bit LSB, i.e., the strategy corresponding to equation (4.14) instead of folding as given by equation (4.12) and (4.13), is expected to result in higher collisions during BF index value generation (i.e. in phenomena 2) and higher actual false positive rate. Thus we expect the actual false positive rate will further deviate from the expression as given by equations (4.18), (4.19) and (4.20). In Chapter VII, we will see that this increase in execution time is rather insignificant when weighed against the amount of reduction in circuit complexity that is achieved by this strategy.

4.6 Summary

To materialize a semantic router which can route messages based on their meanings, we need to represent and compare composite meanings. A composite meaning is represented as a tensor in an infinite dimensional vector space. The similarity between two meanings as represented by two tensors is computed as dot (cosine) product of the tensors. These tensors are expressed in an infinite dimensional vector space, even though the tensors have limited number of basis vector components with non-zero coefficients. This gives rise to the problem which is how to quickly identify the basis vectors that are common to both tensors, so that we can pair up their coefficients for multiplication and addition to compute the dot product value. To quickly identify the common basis vectors and finally compute the dot product, we proposed a hardware centric algorithm and an information processing architecture. Finally we presented several computational alternatives to reduce the circuit complexity and circuit power draw of the design, followed by an analysis of the design tradeoffs.

CHAPTER V

SEMANTIC ROUTED NETWORK

Systematic organization of the index entries can save number of servers necessary to build a distributed index system and result in reduction of power consumption. However this system requires a network to redirect search queries to the particular index server which is most likely to contain matching index entries. In this chapter, we present a design of an overlay network called Semantic Routed Network which can route messages based on their meanings. This network can be used to deliver queries to the destination index server based on the meaning of the query messages. We begin by explaining how such overlay network will be incorporated in a distributed index system. Next we explain the principle of meaning based semantic routing, followed by the design of this overlay network.

5.1 Semantic Routed Network

5.1.1 Recapitulation: What is Semantic Routed Network

Semantic Routed Network (SRN) is a collection of networking appliances called semantic routers and index nodes (index server pools) connected in a peer-to-peer fashion. SRN is implemented as an overlay network on top of existing IP network or cluster systems. This SRN implements the proposed distributed index system in a search engine (refer section 3.2, Chapter III).

5.1.2 Generalized versions of networking concepts

To achieve message routing, the proposed SRN incorporates the several notions which are analogous to the following fundamental networking concepts like:

1. Assigning addresses to destinations.
2. Grouping destinations to form sub-networks.
3. Assigning address to a sub-network (or the gateway router).
4. Deciding next hop address during routing, based on the similarity of the message destination address and the next hop destination address.

However these are achieved in a different manner in SRN, compared to an IP network. The following sections explain how these notions are applied in SRN.

5.1.3 Destinations, routers and destination address assignment scheme

In the Semantic Routed Network (SRN), there are three kinds of nodes. There are two kinds of physical node and one kind of virtual node. Each index server pool is considered as a unique physical destination node, we call these nodes as "index nodes". The other kind of physical node is called "semantic router". These router nodes forward messages to other routers or destination nodes. The SRN is actually implemented as an overlay network of semantic routers interconnected to each other in peer-to-peer fashion and also connected to the index nodes. Index nodes are only connected to the semantic router nodes.

In SRN, each index entry is considered as a virtual destination node. In section 3.8, we mentioned that each index entry has a semantic descriptor that describes the meaning of the corresponding web page or document content. In SRN, for the purpose of semantic routing, this descriptor is considered as the address of the index entry.

When a query (or message) is delivered to the index server node, the index node performs index look up to identify the matching index entry. This is analogous to message delivery to the index entry which is the final virtual destination located at a physical destination (index server node). On the other hand this index lookup is considered as a form of virtual routing to deliver the message to the final virtual destinations (the index entries). Thus the index nodes are viewed as virtual routers with only virtual routing ability (i.e. only index look up function) and also as physical destination nodes (having a unique physical address). The overlay SRN can be implemented on top of an open standard network such as IP (e.g., HTTP on TCP/UDP over IP), or on top of a proprietary or standard node clustering technology. In case the underlying network is HTTP on IP, then all physical address of the SRN are preferably represented in form of URI [143]. If the underlying technology is a clustering system, then this address would be the cluster id. Henceforth in all our examples and explanations we will consider all physical addresses to be in URI format.

This scheme entails the following network (and search) operation. To send a message to a specific index entry, the sender constructs a message with a semantic descriptor as the message address and submits it to the SRN. The SRN delivers the message to those index entries whose keys (semantic descriptors) are similar to the message key. The message address key is constructed to be similar, in terms of meaning, to the key of the destination to which the sender intends to send the message. Once the destination gets the message it can respond back to the message originator or some other predetermined node (e.g. document server) and present itself. This routing behavior can be also used for searching. The originator can use the query descriptor as the message key and when then the destinations, i.e., the index entries having keys similar to the query will present themselves to the originator. Actually the index nodes present the index entries to the originator but this action is same as if index entries are presenting themselves. In this way, the originator can identify the index entries that it is searching for. Thus this kind of address assignment and network operation enables searching of index entries from a cluster of index nodes.

As explained in earlier Chapter II and 3, the descriptor (or meaning) similarity is not based on exact matching of the key to the document content or its data schema items, so the sender do not need to know exact descriptions of the destinations to identify them. The sender can use alternate forms of descriptions as long as they convey same meaning and yet successfully send message to the intended index entries. Therefore this kind of addressing and descriptor comparison can enable meaning based query delivery and searching.

5.1.4 Network organization and router address assignment scheme

In SRN, each semantic router has a semantic routing table which stores addresses and description of destinations and other peer semantic routers. Each semantic router also has an assigned descriptor that describes its interest or specialization. A router tends to maintain addresses of only those destinations and routers in its routing table whose descriptors convey somewhat similar meanings as its own descriptor. This is analogous to human beings maintaining the contacts information of their associates and friends, who are interesting to them, in their address book. Here the semantic routers are analogous to people, the router descriptions are analogous to people’s interest profile and the semantic routing tables are similar to the address books.

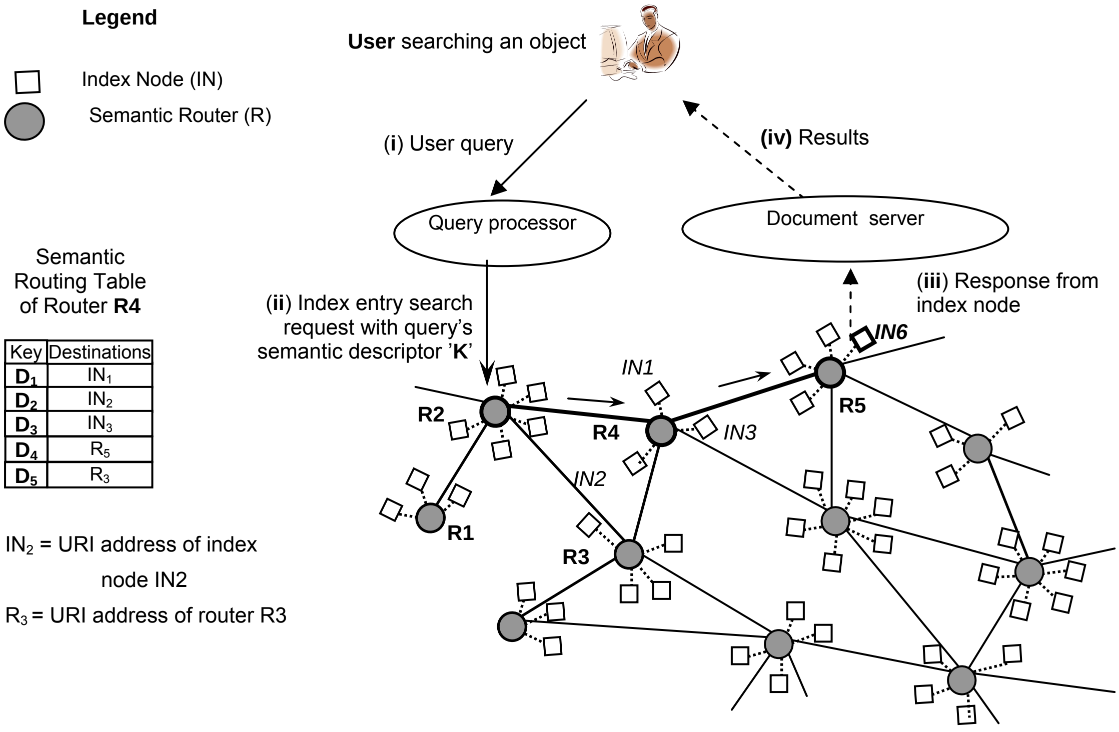
This ensures that index entries of similar kind and routers having similar interests are clustered together to form a routable network. This use of routing table along with router’s descriptor, makes it is possible to use a single uniform destination decision (routing table lookup) mechanism to achieve both: forwarding of message and message delivery to the final destination. This is similar to packet routing in an IP network.

As the index nodes are considered as virtual routers, they too are assigned an interest descriptor based on type of index entries that are stored in it. Therefore a semantic router or an index node is analogous to a gateway router node, all the destination addresses in their routing (and index) tables considered together are analogous to a sub-network and the interest descriptor of this router (or index node) is analogous to this sub-network address. Each semantic router that is also connected to a group, other than the given group, is analogous to a gateway node for the given group. In IP networks, the hierarchy is very strict where the sub-networks do not overlap at all and generally there is only one designated gateway. But in the SRN the sub-networks overlap with each other and there may be multiple gateway nodes.

5.1.5 Routing operation

Fig.5.1 illustrates the routing operation in Semantic Routed Network. Message delivery is achieved by forwarding (routing) messages among semantic routers which are connected to each other. In this figure, the semantic routers are indicated as small dark circles and physical destinations (index nodes) as squares.

A portion of a semantic routing table for router “R4” is shown at the right side. The routes/links between peer routers and index nodes are represented as next hop destination addresses in the routing tables. For example, a uni-directional link between router node R4 to node index node IN1 is represented by the first row in the routing table of R4, as shown in Fig.5.1. There can be a bi-directional link (or two uni-directional links) between two routers, in that case the routing table of each of these routers will have each other’s URI addresses. Similarly an index node will at least have a link from index node to a router, so that an index node may send a message to that "known router". The index nodes may maintain a list of known routers, which serves similar function as the routing table but only for outgoing messages which originates from that index node. These entries in the semantic routing tables and known router lists represent the links between the respective nodes. Henceforth by "links/routes/edges" we will always means these entries. A router may connect to either another router or an index node, but an index node will always connect to only routers. Here we have only shown a few index nodes connected to each router, however in a real system many more index nodes will be connected to a single semantic router, than what is shown in this figure.


Legend
□ Index Node (IN)
● Semantic Router (R)
User searching an object
(i) User query
(iv) Results
Query processor
Document server
(ii) Index entry search request with query's semantic descriptor 'K'
(iii) Response from index node
IN6
IN1
IN3
IN2
R1
R2
R3
R4
R5
Semantic Routing Table of Router R4
IN2 = URI address of index node IN2
R3 = URI address of router R3

Key	Destinations
D_1	IN_1
D_2	IN_2
D_3	IN_3
D_4	R_5
D_5	R_3

Fig.5.1 Semantic Routed Network as a distributed index system

Each router forward/route messages to other per routers or index nodes that can best handle the message. The principle of semantic routing is similar to social networking which works as follows:

> *Jim is looking for an expert on "processor architecture", so he asks his friend Bill, who is a computer engineer, for help. Bill forwards this request to his friend Tom who designs processor chips. Tom calls Jim with an offer to help.*

In SRN, each router is a like a person having a network of friends and associates (i.e. routers connected to other routers). In this proposed system, a user can pose a search query (transaction labeled as "i" and indicated as an arrow in Fig.5.1) to the query processor. The query processor constructs a suitable search/query key "K" and submits an "index entry search request" message with this key to any randomly chosen semantic router (arrow "ii" to R2). The message also includes a similarity threshold value and the address of the document server as the message originator address (so that the destination can respond to the document server). The significance of the similarity threshold value is explained later. The search key represents the meaning of the desired web page or document to the extent what the user can define. The query recipient semantic router will forward the query by multiple hops, to the final physical destination (index node IN6). In Fig.5.1, router "R2" accepts a message from the query processor and routes it to router "R4", which routes it to "R5", and finally "R5" delivers the message to the index node "IN6" which may have matching index entries. The index node IN6 will compare the similarity between the message key and descriptors of all the index entries which are best matches and if the compared values are greater than the message's similarity threshold value, then it will send the document ids from these selected index entries to the document server (transaction labeled "iii"). The document server will then retrieve the documents and present them to the user (arrow “iv”). The mediating query processor that initiated the search may decide to relax the similarity threshold value to a certain extent, if sufficient responses are not received then the sender may send additional queries with the relaxed threshold values.

The next hop destination to forward the query message is decided using a table lookup mechanism (semantic lookup). For example, in Fig.5.1, semantic router “R2” receives a message and finds its key “K” to be most similar to its semantic routing table row key “D_4”. In this case, this particular routing table row has a corresponding address belonging to router “R4”. This address is either a cluster id or an URI. R2 carries out a DNS lookup look up with R5’s address to ascertain R5’s IP addresses to forward the query message as the IP payload. In case there is no routing table match, the query may be broadcasted to all addresses. In this manner the semantic routing will operate on top of IP routing network.

As the semantic descriptors represent the meaning of the destination’s or router’s description and sender’s intended target, messages are routed and delivered based on the similarity of meaning (semantics) between index entry’s descriptor and sender’s intended target. Hence this message delivery system is called as Semantic Routed Network.

5.1.6 Semantic routing table and their content

As index nodes takes care of the last leg of semantic routing to deliver the messages to index entries (final virtual destinations), so the index nodes need to store the routes to all the index entries (i.e. descriptors of all index entries and their memory addresses). Whereas, the physical semantic routers need to store only the route information for the physical semantic routers and index nodes in their semantic routing tables. Therefore the semantic routing table stores descriptors and addresses of the physical destination nodes i.e., index nodes and other semantic routers. This avoids the need to store addresses of a large number of virtual destinations in the semantic routers and yet enables delivery of messages to the final virtual destinations (i.e. the index entries).

5.1.7 Semantic routing table lookup mechanism

Semantic routing table look up involves comparing the semantic descriptor of the message against all descriptors in the semantic routing table rows, and then identify the row whose similarity value is the largest. A simple table lookup strategy is conceived where *n* comparisons are made to identify the best matching row key, when there are *n* routing table rows. All these similarity comparisons will use this descriptor comparison technique and the proposed comparator architecture as presented in the earlier chapter. The pseudo-code for the semantic routing table lookup algorithm is presented in Fig.5.2.

Semantic Routing Table as a specimen input to the algorithm

Key	Destinations
D_1	IN_1
D_2	IN_2
D_3	IN3
D_4	R_5
D_5	R_3

Output of the algorithm
is
Row # 4

Reflects the example
situation as in Fig.5.1

Semantic routing table lookup algorithm

```
Extract destination descriptor key from message;
For each row in the routing table row, do {
        Compare the row key with the message to get the similarity value ;
         Store the current row number in a min-heap using the similarity value as the key ;
 }
From the min-heap's root node get the row number ;
 Using the row number extract the destination addresses from routng table;
```

Fig.5.2 Semantic routing table lookup algorithm

The algorithm operates on the semantic routing table (also shown in the same figure) and generates the destination addresses of those nodes where the message should be forwarded to, as the output. The complexity of this algorithm is O(n), where there are n rows in the semantic routing table. The min-heap data structure always stores the minimum key in the root node [144]. This data structure helps to identify the minimum value-key pair. Here we are using a min-heap data structure instead of a simple minimum value storage register, because the heap can be parallelized. This kind of semantic table lookup leads to greedy routing of messages in the SRN, where messages are routed by a general sense of direction with respect to the current routing node.

5.1.8 Notion of semantic space and greedy routing in SRN

To fully appreciate the significance of greedy routing in SRN and the associated challenges, we need to understand how SRN relates to the notion of semantic space and the idea of greedy routing in that space. Semantic space is a collection of points. Each of these points corresponds to a composite concept. Thus the semantic space represents all possible composite concepts that could ever exist. The distance between any pair of points represents the similarity of the meaning of the corresponding concepts. We decided to represent the meaning of concepts by a sparse tensor and the similarity between two concepts by the dot product of their tensors, as explained in Chapter III. Therefore, the distance $s_{\text{sem-dist}}$ between the points in the semantic space can be conceived as the modulus of reciprocal of the dot product s_{dotprd} quantity minus 1, i.e. $s_{\text{sem-dist}} = \left|(1/s_{\text{dotprd}} -1)\right|$. This means, when the dot product s_{dotprd} is very small or 0, it leads to a very large or infinite distance between points in the semantic space, and when s_{dotprd} is 1, the distance is vanishingly small. Thus this semantic space is distinct from the vector space in which the meaning vector/tensor is defined. However the relationship defined by $s_{\text{sem-dist}} = \left|(1/s_{\text{dotprd}} -1)\right|$ is the mapping between points in these two spaces.

Each SRN node, which is either an index entry, index node and semantic router, has an associated meaning, which is either the meaning of its content (when the node is an index entry), or its specializations (in case of index nodes or routers). Therefore these SRN nodes can be mapped on to the semantic space by positioning each node on a point that corresponds to its associated meaning. When the nodes have similar meaning, they are closer in this space, other wise they are long distance apart. Henceforth we use this map of SRN in the semantic space to analyze and explain all problems and solutions. All diagrammatic representations of SRN in rest this dissertation will be assumed to be mapped on this semantic space. The semantic routing table lookup mechanism, as explained in the earlier section, leads to greedy routing in SRN in this semantic space.

5.1.9 Overview of the SRN protocol stack

In SRN the semantic routing and IP (or cluster) routing will operate together. The SRN is supposed to be implemented as a P2P network of semantic routers which is overlaid on top of the IP (or cluster) network. The semantic routers are dedicated network nodes which redirects message over the IP (or cluster) network. Once implemented over an underlying IP network, the SRN protocol stack will look like Fig.5.3 [145].

Network Paradigm	Protocol Layers	Address Key used	Mapping (Performed by)
Semantic Routed Network	SRN Layer	Semantic Descriptor	Semantic Descriptor to URI (Semantic Routing Table)
Overlay Networking	Application Layer (HTTP, SOAP)	URI	URI to IP (DNS Service)
Traditional Networking	Transport Layer (TCP)	TCP Port	
	Network Layer (IP)	IP address	IP to Router Interface (IP Routing Table)

Fig.5.3 Network protocol stack of the proposed SRN

The semantic routing table provides the mapping between semantic descriptor and URI addresses, the DNS service provides the same between URI and IP addresses, whereas the IP routing table maps IP address to a router's interface.

5.2 Performance concerns and design imperatives

5.2.1 Determinants of performance

In distributed search engine using SRN, the search is completed only when the search query percolates to all distant routers, as necessary, and finally reaches the intended destinations. In routers, message processing takes time. The nodes that are too far away from the routers that first received the query from user, will take time receive the search queries (messages) and thus the search can take longer time to complete. But users can wait for only a certain time duration within which the results will have to be returned. Many relevant matching document ids will not be returned within that wait time window and they will not be available as results. This means that that the search response time is directly related to end-to-end query routing response time in the SRN. In addition, how routers are connected to each other is also important, because some pattern of connections (SRN topology) will not be able to always successfully route a message to an intended destination. So search recall and precision as defined in section 2.2, Chapter II, also depends on the success of message routing.

5.2.2 Design imperatives

To enable faster searching (lower search response time), the query message should be routed to the final destinations as quickly and as successfully as possible. This will depend on the how the routers are connected, i.e. SRN topology and how next hop destinations are actually decided, i.e. routing method – either pure greedy routing or indirect-greedy routing. Thus in SRN, the search problem becomes message routing and topology construction problem. In the next section we delve into the details of this problem.

5.3 Choice of network topology

Several network topology options are available, but not all suitable for SRN. To help choose a suitable network topology, we first identify the requirements, then evaluate the available options and finally choose the right topology for the SRN.

5.3.1 Network topology requirements

Search performance is directly affected by choice of topology and routing method. Therefore network topology and routing method should be chosen carefully based on the following requirements. The chosen topology should:

(A) facilitate search over large number of index entries while maximizing routing (and search) success rate and minimizing the average time required to route messages to final destination (and get search results).

(B) minimize number of messages arriving at any particular SRN router node. Message processing load should be preferably spread evenly among all nodes to avoid congestion.

(C) minimize message overhead (number of messages generate per query) to avoid overloading the network.

(D) minimize number of routing nodes required to build a SRN.

(E) minimize number of rows that are required in the semantic routing tables to attain a given level of search success rate for a given number of destinations to be searched. Large number of rows in routing table means more comparisons and slower routing decisions. On the other hand searching among a large number of index entries and achieving good search performance require more destination addresses in the routing table. Therefore optimizations are necessary to tackle this tradeoff situation;

(F) have sufficient number of routes (i.e., addresses in the routing table) available to route messages using greedy routing algorithm. Routers can only have a local knowledge of the network that is immediately around it because they can store only limited number of routes in their routing tables. We call a topology routable if using the greedy routing paradigm, the routers are able to route the message to the intended destination with high degree of success, using limited local knowledge of the topology and a general sense of direction.

5.3.2 Network topology options

Here we discuss six categories of network topologies to evaluate them. These are presented below [15].

5.3.2.1 Hierarchical networks with uniform node degree

Fig.5.4 shows a typical hierarchical network, where the destination nodes are shown as squares, and router nodes are shown as circles, with links between them. This kind of network is hierarchically organized where nodes are strictly grouped into non-overlapping sub-networks and sub-networks are grouped into higher level of sub-networks. Nodes having similar addresses are put into a common group (sub-network). The common portion (e.g. IP address prefix) of all node or sub-network addresses in a group becomes the address of that group (sub-network), and the addresses of all peer nodes or sub-nets in that group falls under the scope of the sub-network address. Each group has a node which acts as the message router for the group. This router node mediates the message exchange between peer nodes or sub-networks and serves as the gateway to and from the sub-network to the external world.

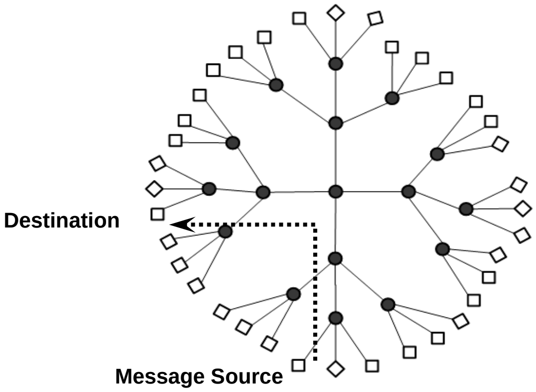


Fig.5.4 Hierarchical network with uniform degree

The highest level router is shown at the center of the network, and lowest level routers are at the periphery. Lowest level routers only connect to the destinations (i.e. destinations can only connect to the lowest level routers). A representative message path is shown in the figure with a broken line in Fig.5.4. To support a hierarchical network of "*N*" destination nodes, ($\log_k N$)

hierarchical levels are needed, where "*k*" is the degree of the router nodes (assuming all routers have similar degree). Therefore the worst case message path length in terms of number of hops necessary to reach a node from another, is 2*($\log_k N$). The average search path length has value that is of similar order of the worst case message path length. The routing table size is of order O(*k*+1) = O(*k*), because each router can route the message to either *k* nodes in its sub-network or to the external world. The main weakness of this topology is that most messages have to pass through the central hub nodes, thereby causing higher message traffic which increases the chance of congestions in those nodes. Hierarchical networks are routable when greedy routing method is adopted.

5.3.2.2 Hierarchical power law networks

Fig.5.5 presents a typical hierarchical power law network. The degree distribution obeys an asymptotic power law (at large *k* values): $P(k) \approx k^{-\lambda}$, where P(*k*) is the fraction of nodes that has degree *k*, and λ is a constant and $2 < \lambda < 3$. There are very few hub nodes that have very large degrees (number of links/edges connected to a node), and large number of hub nodes having smaller degrees. Hierarchical power law networks are also routable when we use greedy routing method. This also being a hierarchical network, the destinations (shown as squares in Fig.5.5) can only connect to the lowest level edge routers.

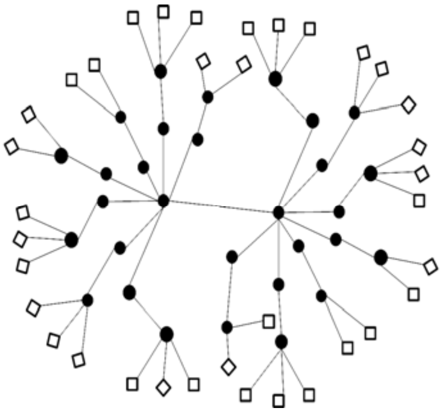


Fig.5.5 Hierarchical power law network

The average search path length is much smaller than the one for hierarchical network with uniform node degree. Whereas the size of the routing table size is in order of O(k_{max}), where k_{max} is the maximum node degree in this power law network.

5.3.2.3 Problems in implementing hierarchical topology in SRN

A balanced tree is the most optimum hierarchical network which has least tree depth and least average path length (Fig.5.6). To construct a balanced tree hierarchical network, the address space has to be partitioned suitably and allocated among the sub-networks. It is necessary that the number of sub-networks and nodes are equally distributed among the peers (children) at the same level. To achieve this allocation, the maximum number of routers and destination nodes that may join the network in future has to be known beforehand and a method to partition the address space should be available. For IP addresses, the address space partitioning and allocation is not a problem, as the address key is a discrete numerical value and the address space is finite. This means that address range can be defined (i.e. all addresses can be enumerated) and partitioned as required.

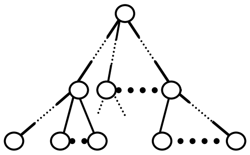


Fig.5.6 Balanced hierarchical tree

However with semantic descriptor as the address key, address space partitioning and allocation among sub-networks to make a balanced tree becomes a problem. This is because of several reasons. Semantic descriptors are not discrete numerical values and the range of semantic descriptor space can not be defined (i.e., all possible values can not be enumerated) and partitioned. Therefore there is no way to uniformly allocate destinations under sub-networks to generate a balanced tree. In addition, the SRN is likely to grow as more nodes are added, and there will large numbers of them at any point of time. Hence, there is no way of knowing beforehand the distribution of these descriptors in the entire semantic descriptor space. This means that we can not plan and implement a balanced hierarchical network organization. Trying any arbitrary address space partitioning and address allocation scheme would lead to an unbalanced tree (Fig.5.7).

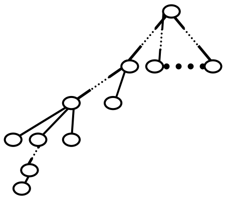


Fig.5.7 Unbalanced hierarchical tree

Trying to balance the tree on the fly as more nodes gets added is not feasible, as balancing extremely large trees are computationally expensive and during such balancing operation the network will remain nonfunctional. So a balanced hierarchical network organization is not feasible for SRN.

5.3.2.4 Random networks with uniform node degree

In a random network, routers are randomly connected with each other and destinations are randomly connected (assigned) to the routers (Fig.5.8). This randomness achieves uniformity in node distribution to avoid problems that lead to unbalanced trees. Thus use of a random network limits the average path length and search path length (both are monotonically related when shortest paths are identified). All nodes might have uniform degree (or normally distributed around a finite average value), hence order of routing table size is O(k). However, these networks have extremely low clustering coefficients [117]. Random networks are not routable when greedy routing method is used. This is because there is no guarantee that routes that are available, will be found. This was explained in section 2.10.4.3, Chapter II.

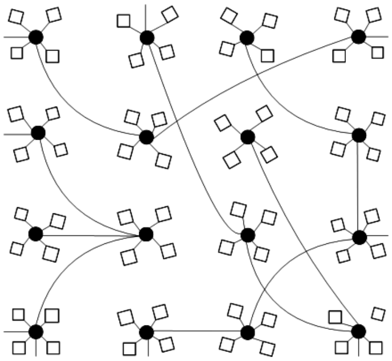


Fig.5.8 Random network with uniform degree

5.3.2.5 Random power law networks

In these random networks the nodes have degree distribution that obeys an asymptotic power law (at large k values) (Fig.5.9). The average path length is in order of O(loglog$N$), which is extremely small for even a very large *N*. The size of the routing table size is in order of O($k_{\max}$), where $k_{\max}$ is the maximum node degree in this power law network. Random power law networks are not routable when greedy routing method is adopted for same reasons explained above.

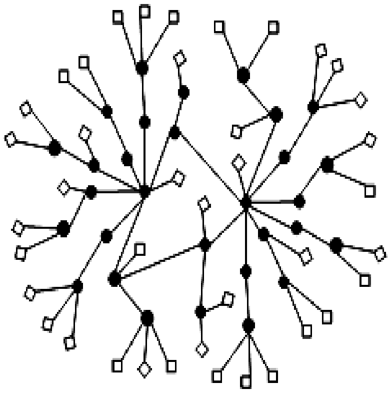


Fig.5.9 Random power law network

5.3.2.6 Lattice networks

Fig.5.10 shows a typical 2 dimensional lattice network. In lattice networks the nodes are connected to only nearby neighbors. This is a feasible topology for SRN. The nodes (routers or destinations) having a specific attribute/metric (semantic descriptor in the case of SRN) is positioned at specific points in the graph. In case of SRN, the distance between the nodes represents the similarity between the node's semantic descriptors. This means that, nodes will connect to other nodes which have similar attributes (similar semantic descriptors).

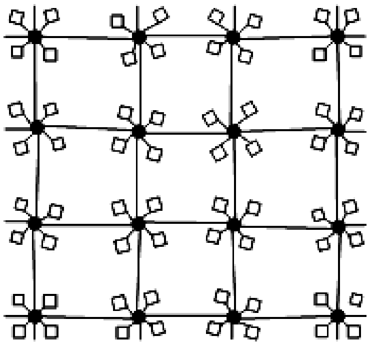


Fig.5.10 Lattice network

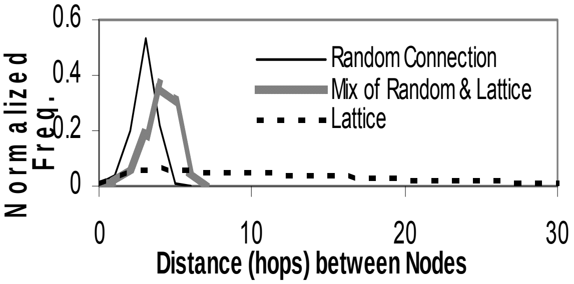


Fig.5.11 Average path length distribution

These networks have high clustering coefficient (probability that two nodes are connected if they have a common peer) and large average path lengths. Fig.5.11 shows the distribution of the characteristic path length of lattice network in comparison to that of the random network. It is possible to uniformly distribute index entries to the routers based on their semantic descriptors in a lattice network. The average search path length is in order of O($N^{1/D}$), where $D = k/2$, and k is the node degree. This is moderately large for a very large N. The size of the routing table size is in order of O(k). Lattice networks are routable when greedy routing method is applied.

5.3.2.7 Small world networks (SWN)

This topology is shown in Fig.5.12. This network is generated if a very small fraction of nodes in a highly clustered lattice network randomly connect to nodes which are far away [118], [119]. Even though this topology has high clustering coefficient property of lattice networks, but its average path length is small enough and quite similar to that of random networks. When a high clustering coefficient value is maintained, this topology is considered routable for all practical purposes because most of the message will be able to get successfully routed to the destinations when a indirect-greedy routing method is used (satisfies requirement (F)). The average search path length is in order of $O((1/l^{1/D})(\log_k N)^{1+1/D})$, where $D = k/2$, and k is the node degree and l is the number of long distance links per node. This is quite small for a very large N. The size of the routing table size is also small, in order of $O(k+((\log_k N)^2)$.

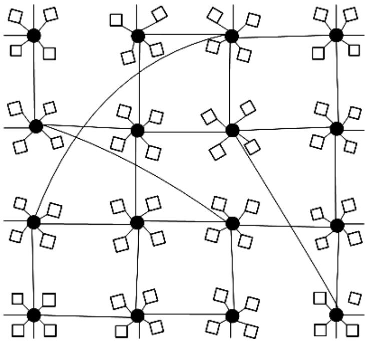


Fig.5.12 Small world network

5.3.3 Performance comparison of topologies and choice for SRN

To assess their suitability for SRN, these six classes of topologies are evaluated against six criteria (Table 5.1) [145],[146]:

1) routability;

2) routing table size;

3) implementability;

4) probability of congestion at semantic routers;

5) number of routers necessary to build the topology; and

6) average path length.

Here routablity of a network is judged by its ability to support greedy routing or indirect-greedy routing. Greedy routing involves simple rules and does not require prior identification of each and every shortest path route between each pair of nodes. This enables a very small routing table, where only the route information to immediate neighbors needs to be maintained. This is because the number of rows in the routing table is in the similar order as router node's degree. On the other hand, large node degree means large number of routing table rows which makes it unpractical, so we need smaller routers with smaller number of routing table rows. A large routing table size is not desirable for SRN because maintaining a large semantic routing table may not be feasible. For indirect-greedy routing, as mentioned in section 2.10.4.5, Chapter II, the router nodes need to maintain only a few additional shortest path routes to a relatively small number of nodes which are not immediate neighbors but located at its vicinity. Therefore routing table size in this case is little larger than that of the plain greedy routing case.

The implement-ability is judged based on the ease of allocating nodes under routers and subnets as described earlier (i.e. ease of organizing the network). Whereas, the intensity of congestion is assessed by the probability that a message has to pass through a particular hub which is likely to be the most congested one.

These congestion probability figures are higher for hierarchical and power law networks because a large number of messages have to pass through a very small number of hubs. For lattice and small world networks, the traffic is uniformly distributed among all hubs equally so the possibility of congestion in any particular hub node is lower. Small world networks also require less number of routers to interconnect a given number of search engines. Empirical simulation of small world network (SWN) [118], [119] indicated that average path lengths of SWN and uniform random topologies (the best case) are comparable. This comparison in Table 5.1, clearly shows that small world topology is the best choice as it has a good combination of all required properties. It is evident that the small world network topology satisfies requirements (B), (C) and (D). Therefore we have chosen Small World Network topology for our SRN design.

Table 5.1 Comparison and suitability of network topologies

Criteria *Topology*	**Routable ?** (Supports greedy routing ?)	**Routing table size**	**Implementable** (balanced ?)	**Congestion** (probability that message passes most congested router)	**Number of routers needed**	**Average search path length** (response time)	**Suitable for SRN ?**
Hierarchical with uniform degree	Yes	Very small $\mathrm{O}(k)$	No (unbalanced)	Highest $\frac{(r-1)*N}{r*(N-1)} \approx 1$	Lower bound (for balanced tree) is $\sum_{i=1}^{\log N/\log k} N*k^{-i} \approx \frac{N}{k}+\frac{N}{k^2}$ No upper bound (for unbalanced tree)	Lower bound (for balanced tree) is $2*\log N$ Upper bound (for unbalanced tree) is N	**No** (balanced tree not possible)
Hierarchical power law	Yes	Large $\mathrm{O}(k_{\max})$	No (unbalanced)	Highest $1-\frac{(N-k_{\max})}{k_{\max}*(N-1)} \approx 1$	Lower bound is $N*\left(\frac{1}{P(k=1)}-1\right) \approx O(N)$ No upper bound	Lower bound is $< 2*\log N$ Upper bound is N	**No** (balanced tree not possible, large router)
Random	No	Very large $\mathrm{O}(N)$ (all routes needs to explicitly programmed)	Yes	Extremely small $\approx \frac{(\log\log N+1)}{N}$	Smaller $\frac{N}{(k-r)} \approx \frac{N}{k}$	Small $\approx \mathrm{O}(\log N)$	**No** (not routable)
Random power law	No	Very large $\mathrm{O}(N+k_{\max})$ (all routes have to be explicitly programmed)	Yes (balanced)	High	Larger $\approx \mathrm{O}(N)$	Smallest $\approx \mathrm{O}(\log\log N)$	**No** (not routable, large router)
Lattice	Yes	Very small $\mathrm{O}(k)$	Yes (balanced)	Small	Smaller $\frac{N}{(k-r)} \approx \frac{N}{k}$	Large $\approx \mathrm{O}(N^{1/D})$ $D = k/2$	**No** (large path length)
Small world	Supports indirect-greedy routing when high clustering coefficient	Small	Yes (balanced)	Very small $\frac{(k-l-r)}{N} \approx \frac{k}{N}$	Smaller $\frac{N}{(k-l-r)} \approx \frac{N}{k}$	Small $\approx \mathrm{O}\left(l^{-1/D}(\log N)^{(1+1/D)}\right)$	**Yes**

N = number of index entries, k = number of index entries and routers registered by each router in networks with uniform router node degree, k_{max} = number of index entries and routers registered by the biggest router in power law networks, r = number of routers registered to each router, l = number of long edges (shortcuts) per router in small world topology, *P(k)* is the power law distribution (e.g. Zipf distribution), D = dimension of the lattice = $k/2$, , $1 < l << r << N$, All logarithms have base k.

5.4 Summary

When the index entries are systematically organized in a distributed search engine index system, then it is possible to save number of servers that are necessary to build the system. This scheme needs a special network to redirect the search queries to the particular index node which is most likely to contain matching index entries. An overlay Semantic Routed Network (SRN) which can route messages based on their meanings can deliver queries to the destination index node. This SRN is the arrangement of peer-to-peer connected semantic routers and index nodes. In this chapter, we explained how such overlay network will be placed in a distributed index system and illustrated the principle of meaning based semantic routing. We also presented the high level design of this overlay network and explained why we would chose to build this SRN to have Small World Network topology. When the SRN is organized as a small world network topology then search response time is minimized and search success rate is maximized.

CHAPTER VI

SELF ORGANIZATION OF THE INDEX SYSTEM

Semantic Routed Network (SRN) should be organized as a small world network topology. However there are several challenges regarding - how to implement such topology in a practical SRN. Furthermore, adoption of a small world network topology in the SRN does not solve all problems, several additional solutions are necessary to enable optimum operation of the SRN. In this chapter we examine these practical problems and propose algorithmic solutions to address them. We begin by explaining how small world topology may be applied in a SRN, and then we present mechanisms that are necessary to optimize the SRN.

6.1 Generation of small world topology Semantic Routed Network

6.1.1 Basic mechanism

A small world topology in a SRN is generated when there is a high level of clustering (i.e. high local clustering coefficient value) among neighboring semantic routers similar to lattice topology and small number of long connections (routes) among distant ones similar to random network topology. Here we assume a SRN representation in the semantic space, as explained in section 5.1.8, Chapter V. High clustering among neighbors means that there is lot of short links among the neighboring nodes which have similar contents or specialization in terms of meanings. This small world network topology is generated when two conditions are simultaneously satisfied [119], [146]. The first condition requires that the routers and index nodes should be homophilic (have preference of those who are like themselves). So that index nodes and semantic routers form links with other semantic routers whose specializations are similar. Whereas, the second condition requires that routers have simultaneous memberships in multiple diverse groups. In a group, all member routers have links to each other, i.e. have key-address mappings of each other in their routing tables (for routers) or addresses in the known router lists (for index nodes).

The homophilic behavior is materialized by routers' preference to register only those routers and index nodes in its routing table, whose descriptors are similar to one of its own specialty (or interest) descriptors. This homophilia creates a regular lattice network, where the short edges connect to immediate neighbors having similar specialties/interests. This behavior results in high local clustering coefficient. In addition, a small number of dissimilar descriptors (specializations) are assigned to a few routers, so that each of those routers can become member in multiple but small number of dissimilar groups at the same time. Membership in multiple diverse groups will effectively create the long edges connecting routers that are semantically far away (i.e. have dissimilar descriptions/interests). Membership to a very limited number of diverse group ensure that this network will have a small Watts and Strogartz generation parameter p [119]. Alternatively, the same effect can be created by permanently connecting (bonding) together two or more routers having a dissimilar specialties. To understand this aspect, consider the graph shown in Fig. 6.1 [15] which is representation of SRN as graph/network (collection of nodes and edges) in semantic space.

Each router is represented as a single or multiple nodes (shown as grey circles in Fig. 6.1) in this graph depending on the number of interest descriptors assigned to this router. Routers having multiple interest descriptors are shown as multiple nodes which are connected by long edges. For example, router R8 has two interest descriptors and therefore it is represented by two nodes each of one belonging to two different groups which are farther away in the space. The same is true for R2. The relative position and proximity among the router nodes are based on how similar (semantically closer) their interest descriptors are. Due to the greedy routing mechanism described earlier, a message is always routed in the general direction towards the final destination with respect to the forwarding router.

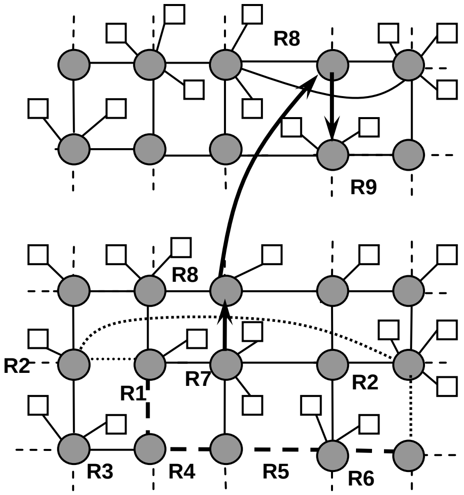


Fig. 6.1 Small world network operations

6.1.2 Small world operation in SRN

The operations of a small world topology SRN can be understood if we consider the example in Fig. 6.1. A message can arrive at a router ("R8") from another router ("R7") based on one of its descriptions/interests ("D1") and then it may be forwarded to another peer router ("R9") whose description (say "D2") is quite dissimilar (and far away) from the initial description (say "D1"). This is equivalent to a traversal over two short edges (R7 to R8 and R8 to R9), corresponding to two physical hops (heavy short straight arrows) and a long edge (no physical hops) in between (heavy curved arrow). Two phenomena: (1) semantic routing; and (2) the proposed router membership behavior working together effectively creates a small world network of semantic routers. Earlier, we had presented the semantic routing table lookup mechanism that materializes semantic routing. Whereas, in the next section, we explain the mechanism which implements the needed membership behaviors of routers and index nodes.

6.1.3 Necessary algorithms for the self organizing mechanism

The self-organization mechanisms are embodied as individual behaviors in part of the routers and index nodes. These mechanisms enable self-organization of a small world topology SRN. Over time, the collective behavior of all these nodes improve (or retain) the small world properties of the network as more routers, index nodes and index entries join the network. These behaviors are presented below.

6.1.3.1 Router behavior

A router periodically identifies new routers that can be better matching group members and develops bidirectional connections with them. To develop a link, one router request the other one to registers its key-address (semantic descriptor-URI) pair in the other router's semantic routing table. Similarly a router, which we call as the "seeking router", also seeks out index nodes that have descriptor keys that are similar to its interest descriptors. Once a new interesting index node has been identified, the seeking router requests the other router, which we call as the "incumbent router", that currently has registered the given index node's address in its routing table, to register the index node with the seeking router (i.e., its routing table). If that incumbent router perceives that the seeking router to be a better matching one than itself, then the incumbent router may agree to shift the registration, i.e. deregister from the incumbent router and register to the seeking one.

To achieve the above, routers periodically send "router search request" and "index node search request" messages, one for each interest description, along with a similarity comparison threshold to identify peer routers and index nodes with similar interest descriptors. If the routing table has vacancy, then the router progressively expands its horizon by sending queries with smaller similarity threshold value to allow more dissimilar index nodes to get registered. A router send these requests repeatedly with a given time period, which we call "router period". This mechanism generates the short edges in a lattice formation, whereas the multiple interests of routers implicitly generate the long edges. These algorithms are presented below in Fig. 6.2.

```
// Routers periodically initiates search.
Every router period clock cycles do {
    For each of its own interest descriptor, do {
        Decrement router search similarity threshold and index node search similarity threshold values ;
        Send router search request message with its own descriptor & router search similarity threshold ;
        Send index node search request message with its own descriptor & index node search similarity threshold ;
       }
  }

//Action when a response to a router search request message is received
When a response to router search request message is received, do {
   If needed, compact own semantic routing table;
   Insert the key-address pair of the responding router in own routing table;
   Request the responding router to insert own key-address to the responding router's table;
 }

//Action when a response to an index node search request message is received
When a response to index node search request message is received, do {
   If needed, compact own semantic routing table;
   Insert the key-address pair of the responding index node in own routing table;
 }

//Action when a router search request message is received
When a router search request message is received, do {
   Compare similarity of all own interest descriptor against the sender's interest descriptor;
    If the similarity is greater than the threshold, then do {
          Respond back to the sender;
       }
 }

//Action when a request to insert a key-address pair is received
If needed, compact own semantic routing table;
Insert the key-address pair of the requesting router;
```

Fig. 6.2 Pseudo code of semantic router behavior

This kind of exploration and discovery of similar index nodes and routers, adds more rows to a router's routing table. A routing table is provisioned for a limited number of rows. So when all routing table rows are filled up, the router may compact the routing table and reclaim more row space. This mechanism is explained in details in section 6.3.

6.1.3.2 Index node behavior

Similarly the index nodes periodically identify better matching routers to register itself, and when it finds one, it requests for a shift of registration to those routers in the similar lines as mentioned above. To initiate this, index nodes also send "router search request" messages with a certain periodicity called "index node period". When a router responds back it verifies which router, the newly identified one, or the incumbent one, to which it is already registered, is a better match to its own interest descriptor. If the newly discovered router is a better match then, it requests to the new router to registers itself to the newly found router's routing table. Once that is successful it deregisters from the incumbent router's routing table.

Similarly on behalf of each index entry in its own index table, the index node also searches for other index nodes that are better match for the given index entry. Upon finding a better matching index node, it negotiates with that other index node (for space in its index table) and if possible transfers the given index entry to that better matching index node. In addition, for an index entry, the index node sends "index node search request" messages to identify a more suitable index node that might host that particular index entry. These algorithms are presented below in Fig. 6.3.

```
// Index nodes periodically initiates search to find similar routers
Every node period clock cycles do {
    Decrement router search request similarity threshold value;
    Send router search request message with its own descriptor & router search request similarity threshold;
 }

//Action when a response to a router search request message is received
When a response to router search request message is received, do {
   Request the responding router to insert own key-address to the responding router's table;
 }

//Action when a response to an index node search request message is received
When a response to index node search request message is received, do {
    Compare similarity of all own interest descriptor against the sender's interest descriptor;
    If the similarity is greater than the threshold, then do {
          Respond back to the sender;
       }
 }

// Index nodes initiates search to find suitable index node for an index entry
For all index entries, for every index entry period clock cycles do {
    Decrement index node search request similarity threshold value;
    Send index node search request message with the given index entry's descriptor & similarity threshold value;
 }

//Action when a response to an index node search request message is received
When a response to index node search request message is received, do {
    Request the responding index node to register the given index entry;
    If the registration to other index node is successful, deregister the index entry from own index table;
 }

//Action when an index node search request message is received
When a response to index node search request message is received, do {
    Compare similarity of all own interest descriptor against the sender's index entry's descriptor;
    If the similarity is greater than the threshold, then do {
          Respond back to the sender;
       }
 }
```

Fig. 6.3 Pseudo code of index node behavior.

6.1.3.3 SRN message protocol

The "index entry search request" and "index node search request" messages are used to search for index entries and physical index nodes respectively. The sender includes a similarity comparison threshold value in these messages. In case of the "index entry search request" message, the destination is an index entry, so the index node compares the message key against all the index entry's descriptors that are stored in the index node. If the comparison value is greater than the threshold, then the index node returns the index entry's URI address to the sender of the request or to an alternate address as specified in the message. For the "index node search request" the destination is the index node itself, therefore the index node compares the message's descriptor against all of its own interest descriptor. If the comparison value is greater than the threshold, the index node returns its own URI address to the sender or to an alternate address as specified in the message.

6.1.4 Adding index entries

A method is needed to add index entries so that the index system may grow. This can be achieved by carrying out the following steps. First the object (URL of the document or web page URL) has to be put it in the document server, then a semantic key for the document has to be generated by a method discussed in earlier chapter, and then the document id and the semantic descriptor (i.e. the index entry) needs to be submitted to any arbitrary index node. Using the SRN's search capability, the recipient index node will identify a more suitable index node which is interested in hosting the object's key-address pair permanently in its index. We already discussed in the earlier sections, how index node implements such behavior.

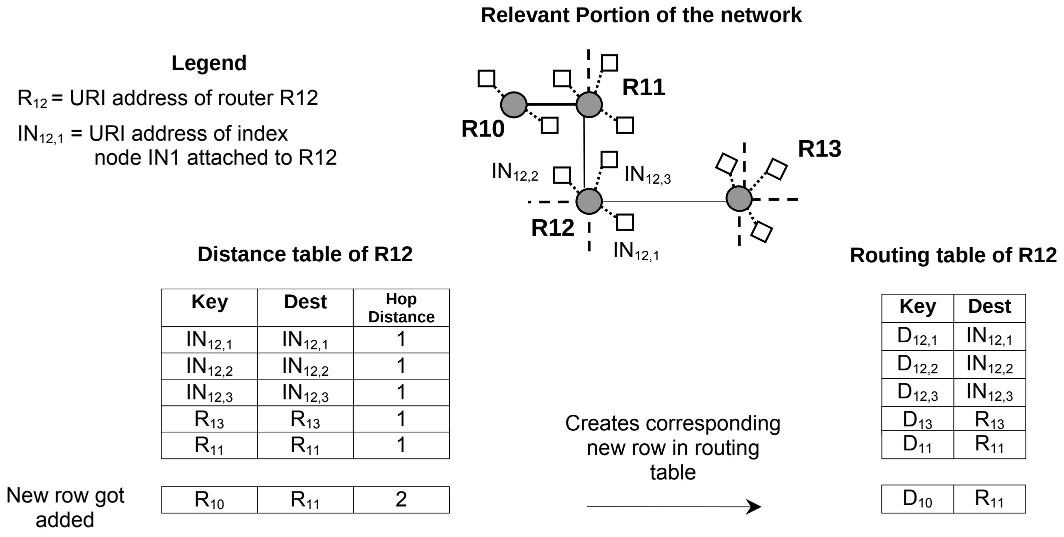
6.2 Programming shortest paths in semantic routing tables

6.2.1 Motivation

The presence of shortest route in a small world topology does not ensure that a message will be routed through it during semantic routing (greedy routing). For example, in the small world network as shown in Fig. 6.1, though a shortest path via router number R1→R2→R6 (shown by thin broken lines) exists but in absence of any additional measure, the semantic routing will direct messages through the longer path R1→R3→R4→R5→R6 (shown by heavy broken lines). To ensure that messages follow the shortest routes, these paths need to be programmed into semantic routing tables using a technique which is inspired by the Open Shortest Path First algorithm (OSPF RFC 1247). This is needed to satisfy requirement (A). However our technique has several difference compared to OSPF, regarding how routing tables are managed and how table lookups are carried out.

6.2.2 Separate distance table and semantic routing table

To aid generation of the routing table which includes these shortest paths, each router maintains neighboring router distance tables as shown in Fig. 6.4, in addition to a semantic routing table.


Relevant Portion of the network
Legend
R_{12} = URI address of router R12
$IN_{12,1}$ = URI address of index node IN1 attached to R12
R10
R11
R12
R13
$IN_{12,1}$
$IN_{12,2}$
$IN_{12,3}$
Creates corresponding new row in routing table
New row got added

Distance table of R12

Key	Dest	Hop Distance
$IN_{12,1}$	$IN_{12,1}$	1
$IN_{12,2}$	$IN_{12,2}$	1
$IN_{12,3}$	$IN_{12,3}$	1
R_{13}	R_{13}	1
R_{11}	R_{11}	1
R_{10}	R_{11}	2

Routing table of R12

Key	Dest
$D_{12,1}$	$IN_{12,1}$
$D_{12,2}$	$IN_{12,2}$
$D_{12,3}$	$IN_{12,3}$
D_{13}	R_{13}
D_{11}	R_{11}
D_{10}	R_{11}

Fig. 6.4 Generation of routing table entries from distance table

This distance table has three columns. The first column is for the URI address of a destination node, which is either an index node or router. The second one for the router's URI address which is the destination of the next hop that would finally take the message to the destination whose URI address is in the first column. The third column is for the hop distance in terms of number of hops necessary for the message to reach the destination (as in the first column) through the given route, whose first hop destination is given by the second column. The hop distance (third) column value gives the shortest distance, required to reach the final destination. The second column of the distance table has similar content as the second column of the routing table. In the example in this figure, in the newly added row of the distance table of router R12, the first column contains the URI address of node R10 (presented as "R_{10}" in the figure), the second column contains the URI address of R11, which is the destination of the first node in the route from the current node R12 to R10 through R11 (i.e. the path R12→R11→R10). The URI addresses in the first column of the distance table acts as the key for lookup purpose in the distance table. For faster lookup with O(1) order of time, this distance table is implemented as a hash table where instead of URI addresses, their hash values are used as the key.

In semantic routers, the distance table is maintained separately from the routing table unlike as in IP routers. This separation allows compaction of semantic routing table, whereas distance table is allowed to remain as it is. During compaction of routing table, rows are merged under a common key to reduce the number of routing table rows. But distance table can not be compacted as it will cause hazards in the shortest route identification operation in the distance table. The shortest route identification mechanism is explained in the section 6.2.5 and the compaction of routing tables is explained in the section 6.3.

6.2.3 Generation of semantic routing table

Whenever a new distance table row is created, a corresponding row is also created in the routing table using the semantic descriptor of the node whose URI address (from the first column of the distance table) and the next hop destination's URI address (from second column in the distance table) as the next hop destination address for the destination column of the routing table. This is illustrated with an example in Fig. 6.4. In this example router R11 after having constructed its initial distance table with the topology information about its immediate adjacent routers R10 and

R12, sends all the new rows in a distance update message to router R12. Router R12 creates a new row in its distance table based on the content of this update message. By this way router R12 gets the topological knowledge that to send a message to R10, it has to be routed through router R11. To illustrate the operation, here we considered the possibility when R11 initiates messages, but other possibilities are also possible.

6.2.4 Correspondence between distance table and routing table

To avoid duplication of information (URI addresses) in distance and routing tables and to simplify simultaneous updation of both tables, the index node and router node addresses will be maintained in a separate address list (Fig. 6.5).

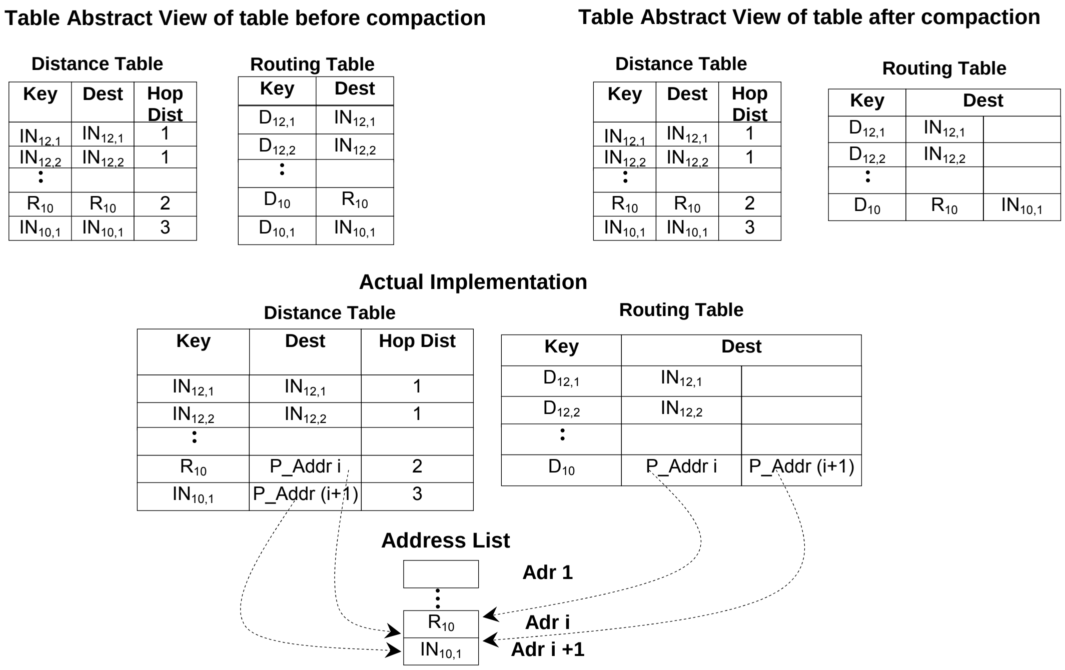


Table Abstract View of table before compaction

Distance Table

Key	Dest	Hop Dist
$IN_{12,1}$	$IN_{12,1}$	1
$IN_{12,2}$	$IN_{12,2}$	1
⋮		
R_{10}	R_{10}	2
$IN_{10,1}$	$IN_{10,1}$	3

Routing Table

Key	Dest
$D_{12,1}$	$IN_{12,1}$
$D_{12,2}$	$IN_{12,2}$
⋮	
D_{10}	R_{10}
$D_{10,1}$	$IN_{10,1}$

Table Abstract View of table after compaction

Distance Table

Key	Dest	Hop Dist
$IN_{12,1}$	$IN_{12,1}$	1
$IN_{12,2}$	$IN_{12,2}$	1
⋮		
R_{10}	R_{10}	2
$IN_{10,1}$	$IN_{10,1}$	3

Routing Table

Key	Dest	
$D_{12,1}$	$IN_{12,1}$	
$D_{12,2}$	$IN_{12,2}$	
⋮		
D_{10}	R_{10}	$IN_{10,1}$

Actual Implementation

Distance Table

Key	Dest	Hop Dist
$IN_{12,1}$	$IN_{12,1}$	1
$IN_{12,2}$	$IN_{12,2}$	1
⋮		
R_{10}	P_Addr i	2
$IN_{10,1}$	P_Addr (i+1)	3

Routing Table

Key	Dest	
$D_{12,1}$	$IN_{12,1}$	
$D_{12,2}$	$IN_{12,2}$	
⋮		
D_{10}	P_Addr i	P_Addr (i+1)

Address List

	Adr 1
⋮	
R_{10}	Adr i
$IN_{10,1}$	Adr i +1

Fig. 6.5 Actual implementation of distance and routing table

The second columns of both distance and routing tables will contain pointers to those locations in the address list. When an address is inserted in any of these tables, it means that the address is inserted in the address list and pointers to this address list location are inserted in both tables. When an address is updated in any of these tables, it means that the address is only updated in the address list, with no change in the tables.

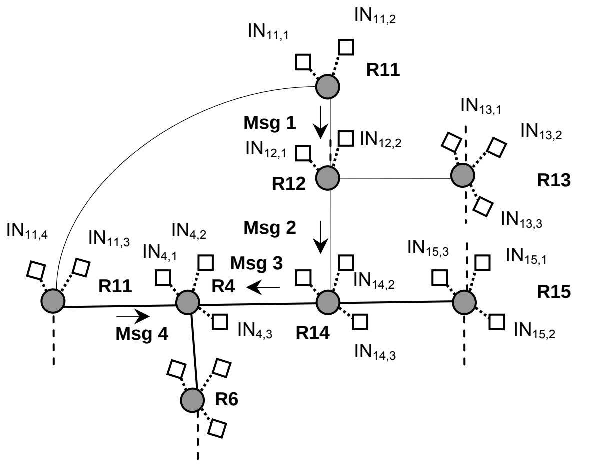
However for sake of simplicity we will show the distance table to contain index node and router addresses instead of their address list location pointers. Routing table rows may contain multiple destination columns after routing table compaction. This aspect is explained in later sections. Due to such compaction operations, the keys in corresponding rows in the distance and routing table rows may be different, but that does not create any problem. In Fig. 6.5 we have shown sample tables before and after compaction along with their actual implementation. In Fig. 6.5, the last two rows of the distance and routing tables before compaction, has different node keys and destination addresses. After compaction, the last two rows of the routing table got merged in to one single row having one descriptor key and two destination addresses.

6.2.5 Identification of shortest paths

The shortest paths in the distance table get recorded through a gradual process where each router shares its distance table contents with its neighboring routers and updates its own distance table as it gets to know the distance table contents of its neighbors. We start this algorithm with an initial state where each router only has a minimal distance table which has rows having the adjacent router URI addresses as keys, hop distance as 1 and the adjacent routers URI address in the third column. Periodically, each router sends out the contents of this distance table to all other adjacent routers using a distance update message. Only those distance table rows which have been updated recently are sent in the update message. These update message are sent using URI address as the message address key, that means the underlying IP network is used for this purpose not the SRN.

For every distance table row received in the update message, a router uses the key from the first column of this row to look up its own distance table and check whether the next hop router URI address sent in the update message offers a shorter route than the one that was recorded earlier in the existing distance table. If a shorter route is available then the distance and the URI address (second and third column) of the router's distance table is updated using values sent by the distance table row in the update message. Gradually, in this manner, the existing shortest paths are incorporated in the routing tables. This distance table update operation is illustrated with an example (Fig. 6.6, Fig. 6.7).

Relevant portion of the SRN



Distance table updation process

Initial Distance Table of **R11**

Key	Route	Hops
$IN_{11,1}$	$IN_{11,1}$	1
$IN_{11,2}$	$IN_{11,2}$	1
$IN_{11,3}$	$IN_{11,3}$	1
$IN_{11,4}$	$IN_{11,4}$	1
IN_{12}	R_{12}	1

Information about new rows are send in the **Msg1**

Distance Update Message to R12

(Msg1)

Key	Route	Ho
$IN_{11,1}$	$IN_{11,1}$	1
$IN_{11,2}$	$IN_{11,2}$	1
$IN_{11,3}$	$IN_{11,3}$	1
$IN_{11,4}$	$IN_{11,4}$	1

Rows are added due to **Msg1**

Initial Distance Table of **R12**

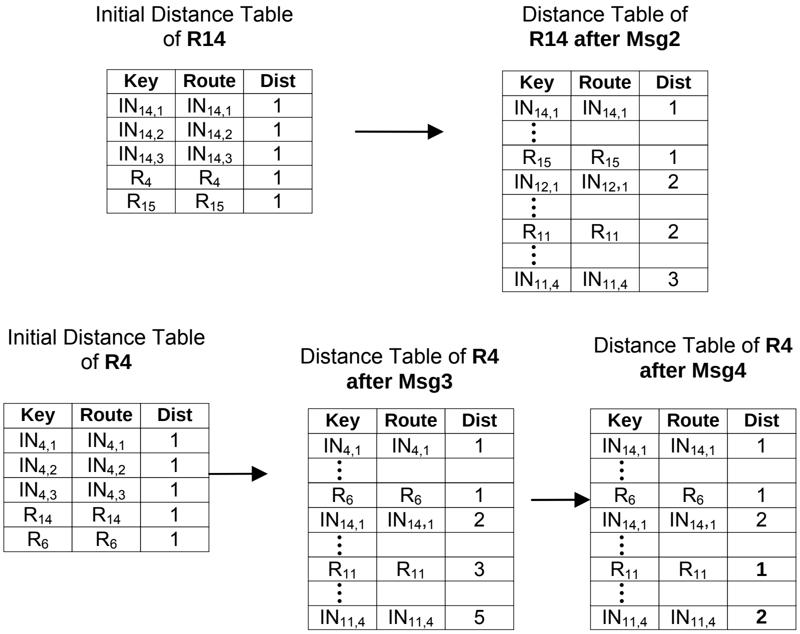
Key	Route	Hops
$IN_{12,1}$	$IN_{12,1}$	1
$IN_{12,2}$	$IN_{12,2}$	1
R_{11}	R_{11}	1
R_{13}	R_{13}	1
R_{14}	R_{14}	1

Distance table of **R12 after Msg1**

Key	Route	Hops
$IN_{12,1}$	$IN_{12,1}$	1
$IN_{12,2}$	$IN_{12,2}$	1
R_{11}	R_{11}	1
R_{13}	R_{13}	1
R_{14}	R_{14}	1
$IN_{11,1}$	$IN_{11,1}$	2
$IN_{11,2}$	$IN_{11,2}$	2
$IN_{11,3}$	$IN_{11,3}$	2
$IN_{11,4}$	$IN_{11,4}$	2

Fig. 6.6 Updation of distance table

In the network as shown in Fig. 6.6, R11 has two descriptors and therefore it is shown to have two nodes with a long edge between them, in two locations, one at the left of R4, another on top of R12. Here we assume that initially, an uni-directional link exists from R12 to R14 and from R11 to R4. Router R11 sends update message Msg. 1 to router R12. After getting Msg. 1 and updating its distance table R12 sends distance update Msg. 2 to router R14, which in turn sends Msg. 3 to R4.



Initial Distance Table of **R14**

Key	Route	Dist
$IN_{14,1}$	$IN_{14,1}$	1
$IN_{14,2}$	$IN_{14,2}$	1
$IN_{14,3}$	$IN_{14,3}$	1
R_4	R_4	1
R_{15}	R_{15}	1

Distance Table of **R14 after Msg2**

Key	Route	Dist
$IN_{14,1}$	$IN_{14,1}$	1
⋮		
R_{15}	R_{15}	1
$IN_{12,1}$	$IN_{12,1}$	2
⋮		
R_{11}	R_{11}	2
⋮		
$IN_{11,4}$	$IN_{11,4}$	3

Initial Distance Table of **R4**

Key	Route	Dist
$IN_{4,1}$	$IN_{4,1}$	1
$IN_{4,2}$	$IN_{4,2}$	1
$IN_{4,3}$	$IN_{4,3}$	1
R_{14}	R_{14}	1
R_6	R_6	1

Distance Table of **R4 after Msg3**

Key	Route	Dist
$IN_{4,1}$	$IN_{4,1}$	1
⋮		
R_6	R_6	1
$IN_{14,1}$	$IN_{14,1}$	2
⋮		
R_{11}	R_{11}	3
⋮		
$IN_{11,4}$	$IN_{11,4}$	5

Distance Table of **R4 after Msg4**

Key	Route	Dist
$IN_{14,1}$	$IN_{14,1}$	1
⋮		
R_6	R_6	1
$IN_{14,1}$	$IN_{14,1}$	2
⋮		
R_{11}	R_{11}	**1**
⋮		
$IN_{11,4}$	$IN_{11,4}$	**2**

Fig. 6.7 Identification of shortest path

Fig. 6.7 shows the situation that, after getting Msg. 3, router R4 makes an entry about router R11 in its distance table. The impact of these messages (Msg.1 to Msg.3) on the distance tables at R14 and R4 has been shown in Fig. 6.7. R11 also sends a distance update Msg. 4 directly after R4 had made an update based on Msg. 3. This particular message helps R4 to find a shorter route towards router R11 and index nodes $IN_{11,1}$, $IN_{11,2}$, $IN_{11,3}$, $IN_{11,4}$.

To introduce new routers or index nodes in the network, they are bootstrapped with a small number of router key address pairs in their distance tables. The new nodes start participating by sending update messages to the routers they know.

6.2.6 Programming shortest paths in semantic routing tables

Due to the correspondence between the distance table and routing table, when the distance table gets updated (which means updating the address list location) with a shortest path, this shortest route also gets programmed in the semantic routing table.

6.2.7 Difference in table lookup speeds in distance and routing tables

Routing table lookups need to be carried out at high speeds, to cater to high message traffic rate. Whereas the distance table update is a much slower process requiring slower table lookup speeds. Thus routing table definitely needs a mechanism to reduce the number of key comparisons (and number of table rows), but distance table can still be implemented by traditional hash table based techniques.

6.3 Compacting the semantic routing table

6.3.1 Motivation

The number of descriptor (key) comparisons required to carry out a semantic routing table lookup depends on the number of routing table rows. The descriptor comparison is fairly computation intensive, hence there is a motivation to limit the number of rows in the routing table. On the other hand the routing table has to accommodate a large number of destination routes to have superior routing success and response time. There is a way to minimize number of rows while storing more destination addresses in the routing table. This is done by allowing multiple destinations in each routing table row. Multiple destination columns in a routing table row creates selective message multi-casting, because multiple destinations having similar descriptors will now receive a copy of the message. This selective multicasting represents multiple short links in the SRN which enables higher routing success (and search recall). This is because all intended destinations are now more likely to receive the message.

To ensure faster semantic routing table look up, number of descriptor comparisons is reduced by limiting the number of rows in the semantic routing table. However during generation of the small world topology, as new index nodes and routers are discovered by a router, new rows (connection) are added in the routing table. As new rows get added in the routing table it runs out of rows. Therefore there is a need to free up rows to record newer and more useful (from routers viewpoint) outgoing connections. Routing table compaction is carried out to free up routing table rows.

6.3.2 Approach

Two strategies are applied to reclaim routing table space without loosing out too much on routing performance. As the first strategy, destination addresses are selectively evicted from columns. Similarly, entire rows may be also evicted from routing tables. The second method involves reallocating the evicted routing table entries so that routing links are reclaimed to avoid deterioration of routing performance. These two mechanisms are applied when the need for row and destination column space arises. These two mechanisms include: eviction of address entries from routing table rows; and then reallocation of these evicted entries in existing routing table rows. The need for eviction arises when the routers are seeking out new routers having similar interests and need row space to record their association links (descriptor-address pairs). The reallocation is a best effort attempt, which means if space is not found to reallocate an entry it is permanently dropped. Successful reallocation avoids permanent loss of a link and thus enables high clustering coefficient in the emergent network. Subsequent paragraphs describe how these mechanisms work.

6.3.3 Routing table entry eviction policies and significance

Each semantic router strives to enhance the space-quality efficiency of the routing table contents over time by replacing the routing table entries with more relevant ones. This is explained by an example as shown in Fig. 6.8. Here, more relevant nodes are those nodes (e.g. R5) whose interest descriptors are more similar to that of the given node (e.g. R11). Such operation effectively replaces distant nodes by neighbors (e.g. R12) of the given node (R11) having more similar address descriptor. In this example node R12 is near to node R11 compared to R5. This transforms a random network to a lattice network, by replacing long distance random edges (links) by shorter ones to immediate neighbors. The long distance link between R11 and R12, which was removed in this example, had been there either due to prior bootstrapping or due to similar operation. It is possible that R11 was earlier disconnected from another node which was further away compared to R5 and then R11 was connected to R5. At that time R12 did not participate in the network.

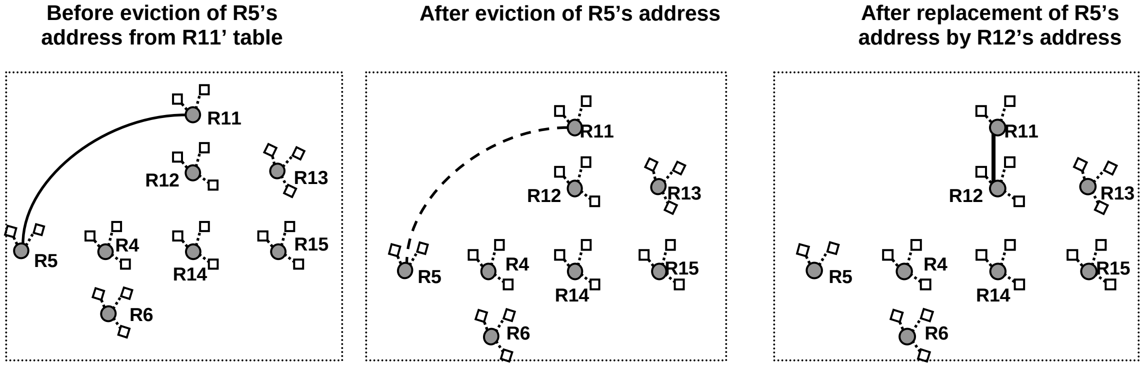

Before eviction of R5's address from R11' table
After eviction of R5's address
After replacement of R5's address by R12's address
R11
R12
R13
R5
R4
R14
R15
R6
R11
R12
R13
R5
R4
R14
R15
R6
R11
R12
R13
R5
R4
R14
R15
R6

Fig. 6.8 Lattice formation by routing table entry eviction

6.3.3.1 Column eviction policy

The destination addresses in the routing table columns are either conserved or evicted based on the “relevancy” of the destination node’s description with respect to the row’s key. The address, whose description is semantically dissimilar to the row’s key, is evicted. This allows creation of a new column space to store a more relevant address and create a new link to a neighboring node in the similar manner as explained above.

6.3.3.2 Row eviction policy

An entire row may be evicted by evicting all destination columns of the row. The candidate row for eviction is chosen based on router’s relative interest on the row’s key, by comparing similarity between the row’s key and router’s interest descriptors. This allows creation of an entire new row to store more relevant addresses and create a new links to a neighboring node.

6.3.4 Routing table reorganization algorithms and significance

When a destination is evicted from a row, an attempt is made to reallocate it in another row. The next best possible place for the relocated entry is determined by a specific logic. This destination reallocation reclaims row and column space and yet avoids complete discarding of destination routes to nodes that are further away from immediate neighborhood. This allows a router to retain knowledge about the topology and routes over a wider geography. This results in higher end-to-end routing success and lower average routing response. The algorithms to reallocate destinations from a single column or the entire row are described below.

6.3.4.1 Reallocation of column entry

The row whose key is semantically most similar to the key in the current row from which the destination is being evicted, is chosen to accept the relocated entry provided it has a vacancy.

The check for vacancy is carried out as follows. If there is a free column space, then the vacancy test returns a straight forward “true”. If there is no free column space then a check is made to ascertain whether this destination being reallocated is likely to be evicted from this row in near future or not. If it appears that the reallocated destination is likely to be evicted in near future, then the next best matching row is considered. The check for possible future eviction is made by comparing the relevance of the reallocated destination node’s description with the lowest relevant destination’s description that is already in that row. This relevance test is carried out with respect to the key of the row that is being considered.

If the vacancy test returns true for a row which does not actually have a free column space, then the lowest relevant destination is evicted from this row to make space for the destination which is being reallocated. The destination, which is swapped out, is reallocated to the best matching row using the same principle. This leads to a recursion of reallocation column entries (domino effect). To identify the best matching row, only those rows are considered which haven’t suffered a destination eviction so far in the current recursion tree. The recursion may terminate either when a free column space is found (which doesn’t lead to another destination being reallocated) or when there are no rows left to consider. This is explained with an example routing table (Fig. 6.9), where the columns are arranged from left to right and rows has been arranged from top to bottom in order of increasing “relevance”. The destination IN_5 from row with key value D_{11} is being reallocated. First destination R_1 from row D_9 is swapped out, then finally destination $IN_{8,3}$ from row D_6 is swapped out.

Step 1: Destination IN_5 being reallocated, from row D_{11}

Key	Destinations			
D_{11}	***IN_5***			
D_9	R_1	$IN_{7,8}$	$IN_{1,3}$	$IN_{7,5}$
D_6	$IN_{8,3}$	R_{11}	$IN_{4,3}$	$IN_{1,8}$
D_8	$IN_{1,3}$	R_2	$IN_{1,1}$	$IN_{1,8}$

Step 2: Dest. R_1 has been swapped out from row D_9

Key	Destinations			
D_{11}				
D_9	IN_5	$IN_{7,8}$	$IN_{1,3}$	$IN_{7,5}$
D_6	$IN_{8,3}$	R_{11}	$IN_{4,3}$	$IN_{1,8}$
D_8	$IN_{1,3}$	R_2	$IN_{1,1}$	$IN_{1,8}$

Step 3: Dest. $IN_{8,3}$ has been swapped out from row D_6

Key	Destinations			
D_9	IN_5	$IN_{7,8}$	$IN_{1,3}$	$IN_{7,5}$
D_6	R_1	R_{11}	$IN_{4,3}$	$IN_{1,8}$
D_8	$IN_{1,3}$	R_2	$IN_{1,1}$	$IN_{1,8}$

Fig. 6.9 Reallocation of destination addresses

6.3.4.2 Reallocation of an entire row

When an entire row is evicted, an attempt is made to reallocate all its destinations using the column entry reallocation algorithm. Row with key D_{11} is an example of row eviction in Fig. 6.9.

6.4 Duplicate message elimination

High clustering coefficient in the small world topology and selective multi-casting during semantic routing results in substantial amount of duplicate messages. To avoid network overloading, duplicate messages are identified and eliminated at the earliest at the routers by several strategies, like:

1) employing a message cache at the routers to detect and drop duplicates;
2) use of limited time-to-live messages; and
3) maintaining a "list of routers visited" in the messages to detect looping message.

Routers maintain a cache to hold a message's signature for some time. The message's signature can be implemented by the combination of message sequence number and the URI address of the message originator. When a new message arrives, its signature is checked against the cache content. If the message is already present then it is dropped, otherwise it is processed. Older signatures are retired from the cache after a timeout period to make space for newer messages.

Smaller timeout periods are desirable to limit cache capacity and scalability. This cache can be implemented by a counting Bloom Filter.

Messages incorporates a time-to-live (TTL) field, which counts number of semantic routing hops by decrementing the time-to-live value set by the sender. Once this value expires to zero the message is discarded by a router. This TTL value is set according to the time window that a user is willing to wait to get search results. This technique enables discarding older messages, which could not reach the destination within the waiting time window as expected by user. Thus the older messages that are serving no further purpose are not processed by the SRN, thereby limiting the network load.

In a small world network topology there are chances that messages may loop back. Message cache with smaller timeout can not detect looping messages. As there is a possibility that messages are selectively forwarded to more than one destinations (due to presence of multiple address columns in a row) therefore bigger TTL values will leave an window for possible message looping and exponential message spawning causing network storms. In such situations, a "list of routers visited" incorporated in the messages enables detection of message looping. Messages maintain a list of address of routers it visited, and a router finding its own address in that list will discard the message.

6.5 Automatic storage space balancing

The aforementioned router and index node behaviors, also have additional benefits. Due to its behavior, the routers work with each other to automatically balance their storage space utilization, so that a situation does not happen when one router's routing table gets completely filled up, while other routers' tables remain empty, causing resource wastages. This works as follows.

Whenever a new index node is added in the semantic neighborhood of an existing semantic router (i.e. the added index node's specialization is similar to the router's descriptor), then that router will eventually discover the index node. If that router has already filled up and does not have any vacant space in its routing table, then the router will drop a different index node from its routing table which is least relevant to its interest. This process is carried out by the routing table compaction mechanism to accommodate the new and more relevant index node. The index node which is now dropped will eventually get accommodated by another router which is located nearby and the next closest one to this dropped index node. This happens because all routers are seeking to add interesting index node in their routing table. The discovery process can be hastened if the router, which is dropping the index node, makes a direct request to all of its neighbors and finally selects one to take up the orphan index node. If that router which is interested to take up the orphan index node also has space problem and has to drop another index node to accommodate the current index node, it will do so. This domino effect of reshuffling and reallocation of index nodes will continue, till all routers balance out their space utilizations. This means that all index nodes in a particular semantic space neighborhood will be accommodated if there is space in any one of the routers in that neighborhood.

This kind of collective dynamic behavior also implies that when population of index nodes in a particular neighborhood in the semantic space becomes large, then it may be necessary to add a few more semantic routers in that neighborhood to accommodate the population growth. Adding a semantic router will entail picking up a model document/object from the semantic neighborhood and assigning its semantic descriptor as the interest descriptor of the new router.

This will mean choosing a document which is similar to those which are in the index nodes that are being reshuffled around. It is possible to detect those index nodes and routers to identify the neighborhoods which need extra capacity. This localized monitoring of capacity utilization is needed because the entire semantic space is not single continuous fabric, but likely to be a fragmented one in real applications. This is because there will be index nodes which are so far away from a semantic router, that those index nodes will never get accommodated with that router, even if the router had vacancy.

Similarly, as more index entries are added, other index entries will also get redistributed between the index nodes. This will also require adding more index node in a semantic neighborhood to accommodate the organic growth of the index entries.

6.6 Automatic reorganization of the distributed index

An index system where index entries are randomly distributed corresponds to a random network topology. This aforementioned mechanism which transforms a random network topology to a desired small world network topology, can also re-organize any arbitrary index system to a organized one where index entries are systematically organized. This re-organization is automatic when the semantic routers are deployed and the required behaviors are imparted (retrofitted) to the existing index nodes.

6.7 Semantic routing table lookup mechanism

For semantic table lookup, the message key is effectively compared with all the keys that are there in the routing table rows to identify the most similar row key and the corresponding row. This comparison can be implemented by two alternative methods. The first one is an exhaustive *n* comparison against all *n* row keys. The second one involves constructing a meaning based index system inside the memory of the semantic router using a small world network graph. To construct a small world network based index, memory locations are considered as the small world network destinations, and traversal tables in memory which helps index structure traversals, are considered as routers. Such a index structure requires only O($\log_k n$) number of comparison, where n is the number of semantic routing table rows and k is the number of rows in the traversal tables. How ever this second kind of index structure will be complicated to implement within a single router node. So we prefer the simple exhaustive comparison scheme. A suggested implementation is described below.

6.7.1 Exhaustive comparison against all row keys

Here all the exhaustive n comparisons against all n row keys will be carried out using the comparator architecture presented in the last chapter. The mechanism is repeated here for sake of continuity. In this scheme, there will be total n comparators to concurrently compare the message descriptor against n routing table row key descriptors, where there are n rows in the semantic routing table. For each comparator, the row key will be considered as the first descriptor whose Bloom Filter component is loaded in the Bloom Filter (BF) RAM locate in stage C of the architecture presented in Fig. 4.5 in section 4.5.4.1of Chapter IV. The message keys will be passed through the comparator one by one and the comparison values will be noted. This means a row key will be loaded only once in a comparator. This will avoid multiple setup times involved in loading the BF RAM again and again.

6.7.2 Design and analysis of semantic routing table lookup scheme

6.7.2.1 The design

To carry out *n* comparisons in parallel, we suggest the high level hardware architecture as shown in Fig. 6.10. This system has a network interface unit, which is connected to *r* comparator modules using a high speed interconnect (shown as “ingress interconnect” in Fig. 6.10). These interconnects will be implemented using Gigabit per second speed systems that links all rack mounted modules. Nowadays, it is possible to get 10GBps or higher speed interconnects, as they are getting available in the market.

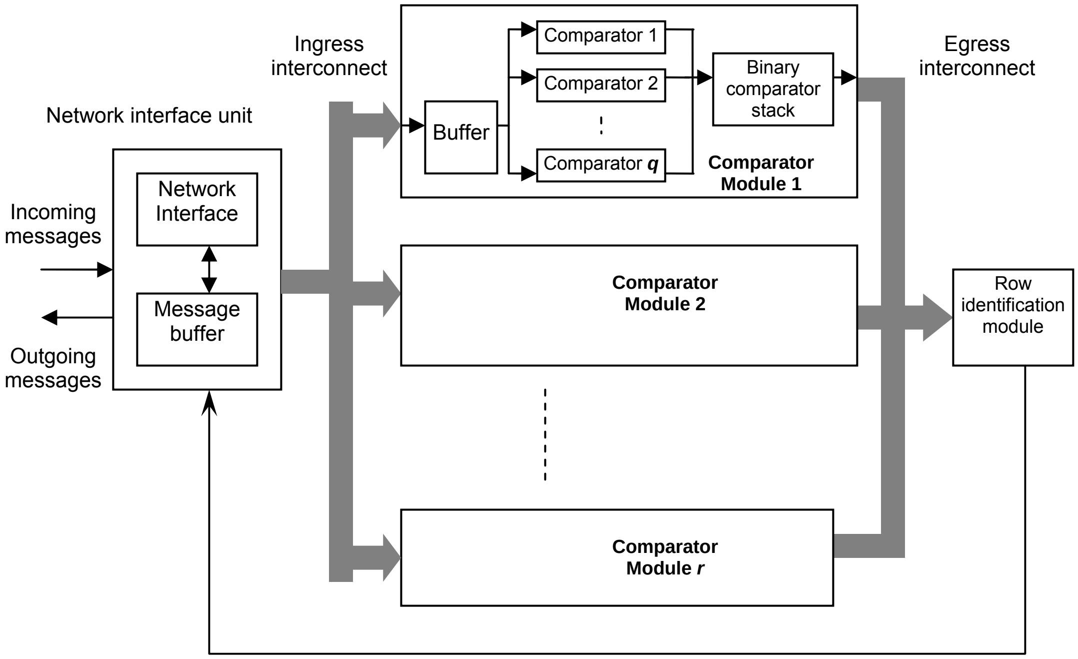


Fig. 6.10 Suggested high level architecture for the semantic lookup sub-system

Each comparator module has a memory (RAM) to buffer the incoming message key and a bank of *q* hardware comparators. Preferably we should choose $q * r = n$, where *n* is the maximum number of routing table rows that the router should accommodate. Packing *q* comparator processors in a single comparator module allows this system to be modular and yet scalable. More comparator modules can be added up to integrate a router with large number of routing table rows. In practice the comparator modules will be implemented as a rack mounted unit (e.g. a form factor similar to a blade server) and to scale up the system, more units will be added in the rack to match the requirement. For our design we assume that each module packs in q (= 32) comparators. Therefore to achieve a router with n = 1000 routing table rows, we need r to be = 32.

In this scheme, each routing table row key descriptor will be permanently loaded in the Bloom Filter of the hardware comparator, and the message keys will be sequentially passed through the comparator hardware as suggested by the earlier section. The timing analysis of the comparator, assuming this design was already presented in section 4.5.4 in Chapter IV. As messages arrive at the network interface, their destination address keys (descriptors) are extracted and passed to the local buffers located inside the comparator modules through the ingress interconnect. A copy of the message will be buffered in the message buffer located at the network interface. The common buffer in the comparator module will feed the message descriptor to all the comparators in that module. As and when the comparison values are obtained they are passed to the binary comparator stack which resides within the comparator module. In total q number of comparison values will be passed on to this binary comparator stack. This stack identifies the minimum descriptor comparison value from the q input values. Once this is identified, the minimum value is forwarded through the "egress interconnect" to the row identification module that is outside the comparators. This module identifies the minimum of the r values arriving from r comparator modules and finally identifies the corresponding routing table row. Thus the minimum value identification takes place at two levels, first within each comparator, then finally at the row identification module. This enables modularity, so that a lookup system can also function with a single comparator module.

The binary comparator stacks and the row identification module together implement the min-heap operation as mentioned in section 5.1.7 of Chapter V. The implementation of the binary comparator sack can be done simply by hooking up several binary comparators in a hierarchical fashion. Each binary comparator compares two values and identifies the minimum. At each hierarchical level, two lower level binary comparators feed a higher level comparator. Therefore to identify minimum of q values, only $s = \lceil \log_2 q \rceil$ steps are necessary, and in total (2^s-1) binary comparators are needed. Actually this can be implemented by simple digital circuits, where each comparison level can be completed with a clock cycle, and the comparison of all the q values can be completed within $s = \lceil \log_2 q \rceil$ clock cycles. Similarly the comparison of r values in the higher level row identification module will take $u = \lceil \log_2 r \rceil$ clock cycles.

The next hop destination location is picked up from the identified routing table row. Once the destination addresses are identified, the message which was buffered at the network interface unit, is transmitted by the network interface to those destinations. The time necessary to buffer the message at the network interface buffer and to send it to the comparator unit buffers, will be of similar order and these two buffering operations should be carried out in parallel. The time necessary for these buffering operations will depend on the interconnect bandwidth and size of the message keys. Next we analyze the response time of this design.

6.7.2.2 Response time analysis

The aforementioned lookup mechanism involves transferring the message data from the network interface hardware to the comparators over the system interconnect, identifying the minimum comparison value, then deciding the next destination address and finally transmitting the message stored in the buffer. For simplicity, we assume that the system (memory) bus and the ingress interconnect is the bottleneck because each message may have large descriptor size. In Chapter VII we show that worst possible size of a descriptor is around 100KBytes. For large messages of this size, the time required to transfer the message data through the ingress interconnect are significantly larger than the message protocol processing time and the time needed to transmit *r* values to the row identification module. Hence the total time $t_{\text{I/O}}$ needed for all transfers, is in the order of the time taken by the memory and ingress interconnect system. Assuming that the memory and ingress interconnect have similar bandwidths, then this $t_{\text{I/O}} = 2$ times the ingress interconnect transfer time.

Similarly, we assume that the time to decide the route is the sum of the following three quantities: t_{comp}, the time taken to compare the message's key against all the keys in the semantic routing table, $t_{\text{min-dest-id}}$ is the time necessary to identify the minimum value and corresponding addresses from the destination column of the routing table, and $t_{\text{I/O}}$.

Therefore we can assert that the time $t_{\text{msgprocess}}$ needed to process a message is given as:

$$t_{\text{msgprocess}} \sim (t_{\text{I/O}} + t_{\text{comp}} + t_{\text{min-dest-id}}) \tag{6.1}$$

Whereas the time $t_{\text{min-dest-id}}$ is given as:

$$t_{\text{min-dest-id}} \sim \left(\lceil \log_2 q \rceil + \lceil \log_2 r \rceil\right) \tag{6.2}$$

6.8 Summary

Semantic Routed Network (SRN) should be organized as a small world network. However there are several challenges regarding how to implement such topology in a practical SRN and how to enable optimum operation of the SRN. In this chapter, we examined some of the practical problems and provided necessary solutions. We presented a scheme to enable automatic self-organization of a small world topology. Next we explained how shortest routes may be identified so that greedy semantic routing scheme can send messages over available shortest paths. Then we presented a scheme to limit the number of rows in a semantic routing table to make the table lookup process faster and yet effective. Finally we presented several mechanisms to detect and eliminate duplicate messages that afflict a small world network topology. All these mechanisms may be applied in any arbitrary distributed index system by deploying semantic routers and imparting the necessary behaviors to the index nodes. These semantic routers and index nodes working together and collectively will transform an arbitrary distributed index to an optimal index system having small world network topology.

CHAPTER VII

EVALUATION, RESULTS AND DISCUSSIONS

In a distributed search engine, a systematic organization of the index entries can lead to efficient use of resources. Such organization leads to smaller number of index servers and less power consumption, compared to a random index distribution scheme. This desired index organization and the associated search operation can be facilitated by a network appliance called semantic router. These semantic routers, when deployed, can materialize an overlay network to route messages based on their meanings. To materialize this router we had identified four problem areas and investigated them. In this chapter we present our claims, thesis, and their substantiation by arguments, analysis, reference materials and simulation results.

7.1 Evaluation approach

7.1.1 Hypothesis and their significance

Here we explain how we will evaluate our proposed solution approaches, which are classified under four research areas (objectives) as identified earlier, along with the respective hypotheses and their significance. These are explained below:

Primary hypothesis. A systematically organized distributed index system, that uses a semantic routed network, requires smaller number of index servers and consumes less power to operate.

Objective 1. *Design of meaning representation data structure and comparison technique.*

Claim 1. The proposed tensor based meaning representation and comparison model is physiologically more realistic compared to vector models.

Sub-claim 1.1 Our proposed tensor model incorporates meaning composition information, therefore it has superior capability to discern composite meanings compared to vector models.

Sub-claim 1.2 This tensor model also represents regenerative composition.

Sub-claim 1.3 This tensor model has beneficial properties that are congruent to how humans interpret meanings. Thus this can be applied to solve some typical problems in information retrieval systems, like term disambiguation, improving specificity of a context, ability to convey meaning by describing the context, etc.

Sub-claim 1.4 This model is congruent with multiple theories, models and hypothesis from cognitive science domain, which shows that the model is physiologically realistic.

<u>Significance:</u> A meaning representation and comparison model, that fits well with the theories of human cognition (psychology), that has beneficial properties to address typical problems and which can incorporate generative meaning composition, is a good candidate.

Objective 2. *Design of a suitable, high speed information processing architecture for use in the semantic routers.*

Claim 2. The proposed information processing architecture has a significant speed advantage in computing tensor/vector dot products compared to traditional processors, hence suitable for use in semantic routers.

Sub-claim 2.1 The computation carried out by this proposed architecture yields correct results.

Sub-claim 2.2 The proposed architecture has much lower execution times compared to those of the software code executing on traditional processors and existing dot product computation hardware designs.

Sub-claim 2.3 The proposed architecture has manageable memory space requirement, thus it is feasible to implement using existing hardware technology.

Sub-claim 2.4 The proposed processing architecture satisfies response time constraints of the semantic router application.

<u>Significance:</u> A processing architecture, that yields correct results and has high speed and manageable memory requirements, is a good choice for hardware design and implementation of semantic router.

Objective 3. *Design of an overlay networking scheme to deliver messages based on their meanings.*

Claim 3. A routable overlay network can be conceived for meaning based message delivery.

Significance: Once we are able to conceive a meaning based addressing scheme and design a routable overlay network topology and a semantic router, then we can address the primary theoretical challenges in materializing a meaning based message delivery network.

Objective 4. *Design of mechanisms that enable construction of a meaning based index using the aforementioned message delivery network.*

Claim 4. The proposed network organization techniques can enable automatic transformation of any arbitrary index organization to an optimal one which has superior search success rate and response time.

Sub-claim 4.1 Semantic Routed Network can implement a distributed index system.

Sub-claim 4.2 The designed semantic routed network can deliver messages based on their meanings.

Sub-claim 4.3 In a Semantic Routed Network, the proposed semantic routing table compaction algorithm allows replacement of large routers (having large number of routing table rows) by small routers without deterioration of routing performance (routing success rate and response time).

Sub-claim 4.4 The proposed self-organization scheme can automatically build a small world network topology Semantic Routed Network. This scheme enables gradual transformation of any random network topology to a small world topology where the message routing delay reduces and the message routing success rate improves with time. The emergent small world topology which is generated has lower message routing delay and superior routing success rate compared to those of a random network topology.

Sub-claim 4.5 It is possible to design an efficient meaning based routing table lookup mechanism which is suitable for use in semantic routers.

Significance: These sub-claims indicate that it is possible to automatically build (self-organize) the required small world network topology that has high routing success and low routing response time, using smaller sized semantic routers. This means that semantic routers need not support large routing tables and thus need not carry out overly large number of meaning comparison to route each packets. Reduction on number of comparison makes the routers fast and feasible for implementation.

This also means that when these semantic routers are deployed in any arbitrary index, then they will self-organize the index system so that all index entries are systematically organized. This emergent index organization will have small world network topology properties, i.e. high search (i.e. message routing) success rate and low response time. Addressing these challenges removes some of the significant theoretical hurdles.

7.1.2 Evaluation plan to substantiate the claims

In this section we explain how we will substantiate the aforementioned claims. We present an overview of our arguments, by referring to existing research work, our own analysis and numerical results. The actual numerical results, illustrations, conclusion and final closure of the arguments are presented in later sections.

7.1.2.1 Systematically organized index system is resource efficient

To establish this resource efficiency aspect of the proposed distributed index system, in section 7.3, we present an analytical model of large distributed search engine system that uses semantic routed network. We will compare this model with a typical distributed index system that does not use SRN (as presented in section 7.2). By this comparison we will show that this proposed distributed index scheme, which uses SRN, will save hardware costs and power consumption compared to the one which does not use SRN. Our proposed distributed index system uses similar load sharing scheme as described in [4]. Hence our proposed design has similar scalability properties as the existing distributed index system. As this scalability solution is not our contribution, hence we will not discuss it in our dissertation.

7.1.2.2 Meaning composition aspect of the proposed tensor model

Linguistic experts have proposed that hierarchical (or nested) sets, as explained in Chapter II, can capture the generative composition aspect in meaning representation and interpretation. Our proposed tensor model for meaning representation materializes the hierarchical or nested set model to address the meaning composition problem.

On the other hand, vector models are established meaning representation and comparison techniques [9], even though they lack in the meaning composition aspect. As our proposed tensor model extends the vector model, hence it inherits all the good properties of the vector model. Thus our tensor model will at least perform as good as the vector model. Next we explain why our tensor model is even better.

To substantiate our hypothesis 1.1, in section 7.4, we illustrate how our proposed tensor model incorporates meaning composition information, and therefore the tensor model is able to discern composite meanings, which the existing vector models can not. Existing results in [12] and [46], had already shown that when vector products (i.e. product of basis vectors) are used to implement rudimentary single level meaning composition, then better search results are obtained compared to plain vector model. As our tensor model extends the use of vector products, hence it inherits vector product's benefit of meaning composition. Thus we can assert that our hypothesis 1.1 is substantiated.

In addition, our tensor model extends this use of vector products to materialize a hierarchical multilevel generative composition. This incorporation of the generative composition is by virtue of its design, as explained in section 3.6.6 of Chapter III. That section substantiates hypothesis 1.2.

Then in section 7.4, we discuss the beneficial properties of the tensor approach that may be used to model how humans interpret composite meanings. Whereas, in section 3.8 of Chapter III, we showed how some typical problems in information retrieval domain may be solved using the tensor model. All these illustrations substantiate our claim 1.3.

Finally, we refer to section 2.4.7 of Chapter II, where we analyzed various theories and models of human memory and cognition from the psychology domain, to show how our adopted hierarchical set model is congruent with these understandings. Through these discussions we substantiate our hypothesis 1.4, that the proposed tensor model is physiologically realistic.

Considering all these aforementioned arguments together we can claim that tensor model supports generative meaning composition and therefore physiologically more realistic compared to existing meaning representation models like vectors and sets. This concludes the evaluation of the meaning representation data structure and comparison technique (objective 1).

7.1.2.3 Speed benefit of the proposed comparator architecture

To evaluate the proposed algorithms and information processing architecture for the meaning comparator (objective 2), we have conducted simulation studies. Using simulation results, as presented in section 7.6, we show that the proposed comparator generates correct results (substantiates hypothesis 2.1). We also show that it is significantly faster than an efficient software implementation and other existing hardware based designs under broad range of operational conditions (substantiates hypothesis 2.2).

Using analytical calculations and results from aforementioned simulations as presented in section 7.6.3 and 7.10, we show that the memory requirements of the proposed semantic router using this processing architecture is manageable (substantiates claim 2.3) and the worst case response time for a semantic routing table lookup is well within the required limits (substantiates claim 2.4). The hardware design of this proposed processing architecture, as carried out by [131], shows its synthesizability and manageable circuit power consumption. Furthermore, we have shown in section 4.5.5, how to reduce circuit complexity and circuit power consumption. This further establishes the viability of this proposed design approach. All these show that the proposed representation satisfies requirement (F) as presented in section 3.1 of Chapter III.

7.1.2.4 Design of an overlay networking scheme to deliver messages based on meaning

In Chapter VI, we had illustrated how the Semantic Routed Network (SRN) will operate. This conception of SRN substantiates hypothesis 3. This hypothesis is further substantiated when we demonstrate this by our SRN simulation results in section 7.8.1.

7.1.2.5 Self-organization of the meaning based index

In the beginning of Chapter VI, in section 6.1, we had explained how a meaning based message forwarding network can co-ordinate and bind smaller index fragments to form a large scalable meaning based index. That section substantiates hypothesis 4.1. Whereas, simulation results in section 7.8.1 shows that such SRN can effectively route messages based on meanings, therefore it substantiates hypothesis 4.2.

To substantiate hypothesis 4.3 and 4.4, we use simulation results in section 7.8.3 and 7.8.4, to illustrate how routing table compaction algorithms enables small routers (having small number of rows in the routing table) to build the SRN which self-organizes to form a small world network topology. We will show that this topology progressively improves the message routing and information retrieval success rates and response times. By using the same results we also show that this success rate and response time is superior compared to those of a random network topology. In addition, we illustrate the efficacy of the underlying algorithms and techniques that enable small routers to built this SRN and materialize this self organizing behavior. This self organizing behavior and meaning based query/message delivery properties enables automatic construction of a meaning based index, as explained in section 6.1.

Finally, we refer to section 6.7 in Chapter VI, and by using the analysis in section 7.10, we show how an efficient lookup mechanism can be designed to implement the semantic routing table which is the heart of a semantic router. These substantiate hypothesis 4.5.

7.2 Analysis of a typical distributed index system

7.2.1 Number of servers required and response time

In our analysis model we assume that index entries are uniformly distributed across all index node pools and user queries are distributed across all objects/documents uniformly. User queries arrive at a rate "*Q*" per second to the search engine, and the internal representations of the queries (one for each raw query send by user) are broadcasted at the same rate to all "N_P" number of index server pools (Fig. 7.1).

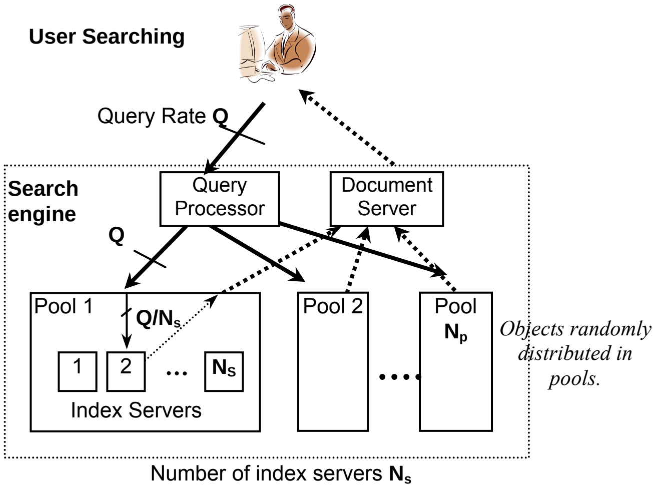


Fig. 7.1 Index distribution and query delivery in a typical distributed search engine

This number N_P is determined by the capacity of each server to host certain "*d*" number of index entries and the need to index a total number of "*D*" objects in the index, according to the following equation:

$$N_P = D/d \tag{7.1}$$

The query arrival rate "$q_{arrival}$" at each server within each pool decides how many servers "N_S" is required in each pool. The number of servers needed in each pool is given by the index look up (or query serving) capacity "$q_{capacity}$" of each server. To achieve stable steady state, $q_{capacity}$ has to be at least matched to $q_{arrival}$, therefore number of servers "N_S" needed at each index pool is:

$$N_s \geq Q/q_{\text{arrival}} = Q/q_{\text{capacity}} \tag{7.2}$$

Therefore total number of "$N_{indexpool}$" servers needed in all the index server pools is:

$$N_{\text{indexpool}} = N_P \cdot N_S \geq \left(D/d\right) \cdot \left(Q/q_{\text{capacity}}\right) \tag{7.3}$$

The search response time "t" of the infrastructure is the sum of the response times for: query preprocessing ("$t_{qryprocess}$"); query delivery to the index ("$t_{delivery}$"); index lookup ("t_{index}"); and document serving ("$t_{docserver}$"). Query delivery is the simplest process involving message transmission (hence faster), whereas the other three involves more intensive computation and expected to involve data reads from memory and hard disks (hence slower). Hence we expect that $t_{delivery} << t_{qryprocess}, t_{index}, t_{docserver}$. So the search response time is given by:

$$\begin{aligned} T &= t_{\text{qryprocess}} + t_{\text{delivery}} + t_{\text{index}} + t_{\text{docserver}} \approx t_{\text{qryprocess}} + t_{\text{index}} + t_{\text{docserver}} \\ &\approx k_1 \cdot t_{\text{index}} + t_{\text{index}} + k_2 \cdot t_{\text{index}} = t_{\text{index}}(k_1 + 1 + k_2) \\ &\approx \frac{1}{q_{\text{capacity}}}(k_1 + 1 + k_2) \end{aligned} \tag{7.4}$$

where $t_{\text{qryprocess}} = k_1 \cdot t_{\text{index}}$, $t_{\text{docserver}} = k_1 \cdot t_{\text{index}}$, where $0 < k_1, k_2$

Here the constants: k_1 and k_2, are introduced to ease the modeling computation and analysis. It is reasonable to expect that k_1 , k_2 have similar order magnitude near to 1 because document serving, query processing and index lookup all three involves memory read operations (either disk or RAM based), therefore involves similar response times.

The number of computers/servers "$N_{criticalpath}$" in the search critical path is given by the sum of computers in the query process $N_{qryprocesspath}$, index $N_{indexpath}$ and document server path $N_{docserverpath}$ which is:

$$N_{\mathrm{criticalpa\ th}} = N_{\mathrm{qryprocesp\ ath}} + N_{\mathrm{indexpath}} + N_{\mathrm{docserverp\ ath}} \approx 3 \tag{7.5}$$

These $N_{qryprocesspath} \approx 1$, $N_{indexpath} \approx 1$, and $N_{docserverpath} \approx 1$, because the query passes through only one server in the load sharing server pool which process the query, and that path constitute the critical path.

As only one server replica in a load sharing server pool is required to process a particular query, therefore, the number of servers that gets used for every search query is the sum of the servers used in each pools multiplied by number of concurrent pools used. Assuming there is only one server pool for the query processor stage, one pool in the document server stage and N_{P} concurrent pools in the index server stage, this total number of server used is given by:

$$N_{used} = N_{\mathrm{qryprocesp\ ath}} + \frac{D}{d} \cdot N_{\mathrm{indexpath}} + N_{\mathrm{docserverp\ ath}} \approx {}^{\mathrm{D}}\!/_{\mathrm{d}} + 2 \tag{7.6}$$

The total number of machines (servers) used in the entire system is given by the sum of servers in query processor pool "$N_{\mathrm{qryprocessorpool}}$", index server pools "$N_{\mathrm{indexpool}}$" and document server pool "$N_{\mathrm{docserverpool}}$". More servers are deployed in the pools in those stages of the processing which are slower. This kind of throughput balancing (i.e. removing processing bottlenecks) is a regular exercise in distributed system deployment and thus this is also assumed here. We also assume that these stages are properly provisioned so that their response times and execution times are very similar (refer queuing theory basics as presented in Chapter II, section 2.11). Thus number of server deployed in each stage is reciprocal to the proportions of their execution (or response) times. Therefore the total sum of required servers is:

$$\begin{aligned} N_{total} &= N_{\text{qryprocess orpool}} + N_{\text{indexpool}} + N_{\text{docserverp ool}} \\ &= k_1 \cdot N_{\text{indexpool}} + N_{\text{indexpool}} + k_2 \cdot N_{\text{indexpool}} = N_{\text{indexpool}} (k_1 + 1 + k_2) \end{aligned} \tag{7.7}$$

This expression excludes the machines/routers in the query forwarding network. Only single router or only few hundred of micro sized routers (refer to section 6.7 in Chapter VI) are sufficient for the query delivery network for this case, compared to the thousands of servers in the pools, so this approximation is reasonable.

7.2.2 Estimated values of some parameters

We need to know the magnitude of these aforementioned variables to appreciate the problem. We estimate their approximate order of magnitude by assimilating indirect information from multiple sources. As more information is available about Google, we consider scale of Google's infrastructure (as reported) as a representative one.

Google [2] and others [3] implied that Google had indexed 26 billion web pages/documents out of unique 1 trillions URLs, so we can consider D to be $2.6*10^{10}$. To estimate *d* we will need to find out how many objects (index entries) (matrix columns in the inverted index as shown in Fig. 2.8 in Chapter II) can be accommodated in a single server.

Here we assume latent semantic indexing is used, and we expect that number of dimensions will be in order of 1000 or less. The argument to support this is available in [147]. For each object (column) and a dimension (row) we will need say 16 bits (or 2 Bytes) for representing the weight coefficient (for a matrix cell in Fig. 2.8) to get an accuracy of 4 decimal places. So for each index entry (an entire column) the memory foot print is 2000 Bytes. Each server having 2GB RAM (or 200GB) hard disk can host 2GB/2KB≈ 10^6 (or 200GB/2KB≈ 10^8) entries, so $d \approx 10^6$ (or 10^8). This indicates that the design parameter $N_P = D/d$, is in order of $2.6{*}10^2$ or $2.6{*}10^4$, depending on whether RAM or disk is used for the lookups.

Google used to report that each search involved about 10^3 computers [5]. We expect this 10^3 number to be number of computers activated for a particular search. This is also given by equation (7.6). So this fact indicates that a more accurate estimate of N_P or D/d is expected to be in order of 10^3.

Google reports the search response time for each query to be in order of 0.2 seconds [5]. So for our purpose we can take a typical response time "*T*" to be ≈ 0.2 seconds (values vary between 0.17 to 0.3 sec. based on our actual observations). Hence using equation (7.4), we can say:

$$T \approx 0.2 \approx \frac{1}{q_{\text{capacity}}}(k_1 + 1 + k_2)\text{ , therefore} \qquad q_{\text{capacity}} \approx 5(k_1 + 1 + k_2)$$

From [1] we know that average Q = 3500 queries per second (as in case of Google), so using equation (7.3), we can compute number of servers in index system N_{servers} is:

$$N_{\text{indexpool}} = N_P \cdot N_S = \left(D/d\right)\cdot\left(Q/q_{\text{capacity}}\right) = \frac{(1000{*}3500)}{5(k_1+1+k_2)}$$

Whereas using equation (7.7), we can estimate the total number of machines used is at least:

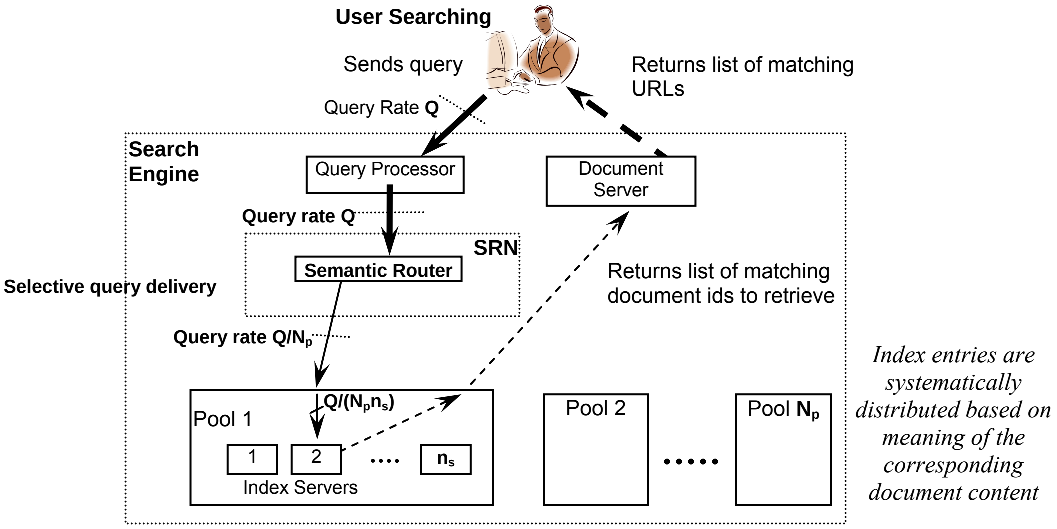
$$N_{\text{total}} = N_{\text{indexpool}}\,(k_1 + 1 + k_2) = \frac{(1000\ * 3500)}{5(k_1 + 1 + k_2)}(k_1 + 1 + k_2) = 700{,}000$$

This is quite close to the value what others have estimated about Google [6].

7.3 Analysis of the proposed index distribution scheme

7.3.1 Meaning based object distribution

We propose an alternative distributed index architecture (Fig. 7.2), where index entries of similar objects are pooled and stored together under single pool instead of randomly distributing over the entire collection of pools. For the sake of fair comparison, we assume that this scheme has the same number of index server pools as the typical search engine model presented earlier in Fig. 7.1.



Index entries are systematically distributed based on meaning of the corresponding document content

Fig. 7.2 Proposed index distribution and query delivery model

7.3.2 Number of servers and response time

In this proposed model (Fig. 7.2), queries are selectively sent to a single (or few) pool(s), hence rate of query arrival is much smaller. The query arrival rate *Q* is equally divided among the number of pools N_p (assuming all kinds of documents are equally preferred). In that case the total required number of servers "$N^P{}_{\text{indexpool}}$" in all the index pools is in order of (note that servers can be allocated to a pool in whole numbers):

$$N^P_{\text{indexpool}} = N_P \cdot n_S = N_P \cdot \left\lceil Q \Big/ \left(N_P \cdot q_{\text{capacity}} \right) \right\rceil \tag{7.8}$$

The rationale behind this equation is same as that behind equations (7.1) to (7.3). Here "n_S" is the number of server in each pool, which is smaller than N_S as in the earlier case (equation (7.3)). This is because each pool here has to handle much smaller query traffic. Here we assume that the SRN does not perturb the response time of the search engine, and only a small number of servers/routers are required to implement the SRN. We shall show later that these assumptions are justified. Therefore the response time and the number of computers/servers in the search critical path are still given by equation (7.4) and (7.5), whereas number of servers that gets used for a search query is given by:

$$N_{\text{used}} = N_{\text{qryprocesp ath}} + N_{\text{indexpath}} + N_{\text{docserverp ath}} \approx 3 \tag{7.9}$$

Whereas the total number of servers needed in all index pools $N^P{}_{\text{total}}$ is:

$$N^P_{\text{total}} = N_{\text{qryprocess orpool}} + N^P_{\text{indexpool}} + N_{\text{docserverp ool}} = N_{\text{indexpool}} \ (k_1 + k_2) + N^P_{\text{indexpool}} \tag{7.10}$$

where $N_{\text{indexpool}}$ is given by equation (7.3).

7.3.3 Benefits of the proposed index distribution scheme

In case of the proposed index organization scheme the $N^P{}_{\text{total}}$ is:

$$N^{P}_{\text{total}} = N_{\text{qryprocess orpool}} + N^{P}_{\text{indexpool}} + N_{\text{docserverp ool}} = N_{\text{indexpool}}\ (k_1 + k_2) + N^{P}_{\text{indexpool}}$$

$$= N_{\text{indexpool}}\ (k_1 + k_2) + N_P \cdot \left\lceil Q \big/ \left(N_P \cdot 5(k_1 + 1 + k_2)\right) \right\rceil$$

$$= \frac{(1000\ *3500)}{5(k_1 + 1 + k_2)}(k_1 + k_2) + 1000\ * \left\lceil \frac{3500}{5000\ *(k_1 + 1 + k_2)} \right\rceil$$

This value is between ~117,000 to ~667,000, depending on value of k_1 and k_2 ($0.1< k_1$, $k_2< 10$). This means that if the index servers are slow compared to the query processor and document servers, then the number of index servers in the pools would dominate the total number of servers and in that case we would get greater saving of servers by this proposed index distribution approach ($N^{\mathrm{P}}_{\text{total}}$ = 117,000, which means 83% savings), otherwise we would get smaller reduction ($N^{\mathrm{P}}_{\text{total}}$ = 667,000, which means 5% savings). We expect k_1 and k_2 to be similar and a equal to a number near to 1, in that case $N^{\mathrm{P}}_{\text{total}}$ ~ 467,000, which means a 33% reduction in number of servers compared to the random object distribution case (N_{total} = 700,000).

This analysis shows that there can be a reasonable reduction in numbers of servers in the index pools. Therefore, we can expect a significant amount of hardware savings by adopting the proposed model in an actual distributed index system (search engine). In addition, such approach will save data center floor space, cooling infrastructure, power costs, all of which means reduction in data center capital investments and operating expenditures. This proposed index distribution scheme is materialized by the SRN, which enables automatic formation of the proposed distributed index and also acts as the query delivery network. To cater to only 1000 pools or destinations, only a single large semantic router (or ~200 of micro sized routers) is good enough. Thus this arrangement does not significantly increase number of machines to offset the server reduction which is in order of several thousands (~10^4).

7.4 Evaluation of the tensor model of meaning representation

7.4.1 Definition of alternative tree comparison metrics

As explained earlier, composite meanings are represented as a nested set or a hierarchical tree structure with basic terms as the leaves. The tensor based model represents and compares two trees that represent two composite meanings. The tensor model has some properties which are useful for realistic meaning representation and comparison.

To explain these properties, we introduce some terminologies and metrics. These are explained in Fig. 7.3. In this figure we show two trees that are compared against each other using three metrics. We have devised these metrics to compare two trees. These metrics gives a deeper insight on the role of composition during the tree comparison process, which the available tree comparison metrics [65], [66], [67] do not. These metrics were mentioned in section 2.6 of Chapter II. We define the "overlap" metric to indicate the amount of leaves present in similar locations in both trees. In the example as shown in Fig. 7.3, the leaves "A" and "D" are common to both trees and located in similar positions in these trees.

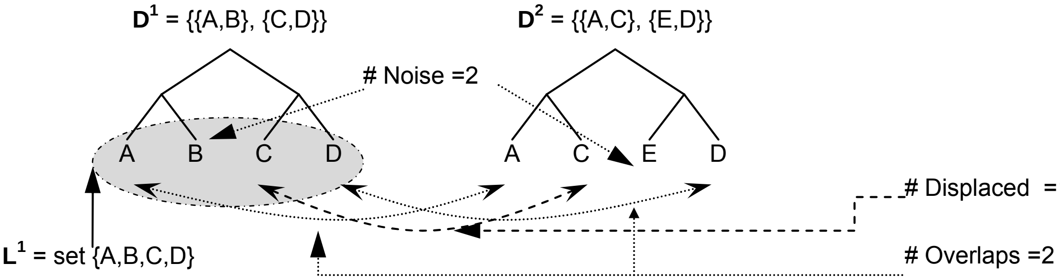


Fig. 7.3 Terminologies for tree comparison

As there are two such leaves, so the overlap metric is 2 in this case. Similarly the number of "noise" denotes the count of leaves that are not common and hence considered as noise when the other tree is taken as the reference one. Here the “B” and “E” are two leaves that are considered noise. Whereas, the “displaced” metric indicates the number for displaced leaves in the two trees (e.g. “C” in Fig. 7.3 [13]).

Based on these three numbers (*Overlap*, *Noise*, *Displaced*), three ratios, one for each number, are derived by dividing them with the total number of leaves in the two trees taken together (number of elements in the union of set of leaves). For example, if L^1 is the set of leaves in the first tree and L^2 is for the second tree then $|L_1 \cup L_2|$ denotes number of unique leaves in both trees. In this example, the set $L_1 \cup L_2$ = {A, B, C} and thus $|L_1 \cup L_2|$ = 3. The overlap ratio is given by $Overlap / |L_1 \cup L_2|$. For this example, this ratio is 2/3 = 0.67. The noise and displaced ratios are similarly computed. These ratios are analogous to “Jaccard similarity” [148] measure used to compare two sets. These three ratios together convey the structural similarity or dissimilarity between a pair of trees. The overlap ratio indicates the extent of similarity of two trees, for example an overlap ratio of 1 indicates that two tree are exactly same, a ratio of 0 indicate totally dissimilar trees and a value in between 0 and 1 indicates that the trees are partially similar. Similarly the noise ratio indicate the extent of the dissimilarity due to presence of dissimilar leaves, whereas the displaced ratio indicates the extent of dissimilarity that is caused when the same leaves are in two dissimilar locations in the two trees. Though we have introduced these three tree comparison metrics to get deeper insight on the tree comparison process, but these metrics are unsuitable for use in an operational search system. On the other hand the tensor based similarity comparison technique is more preferable than these ratio metrics, due to three reasons. The tensor model:

1) does not involve the cumbersome and computation intensive leaf counting required to compute these ratio metrics;

2) uses a single number to convey the similarity compared to using three separate ratio metrics;

3) can be used for multi-level unbalanced trees without any need for additional computations;

4) has other beneficial properties that the ratio metrics do not have. These are explained in the next section.

7.4.2 Tensor comparison is consistent with other comparison metrics

To evaluate the tensor model, we consider couple of balanced trees, and then compare these trees using three methods: tensor; vector; and the metrics described above and present all these similarity values in Table 7.1 [13].

Table 7.1 Consistency between tensor and ratio based tree comparison metric

Row #	D_1 *having set of leaves* L_1	D_2 *having set of leaves* L_2	$D_1 \bullet D_2$		$\frac{\textit{Overlap}}{\left\|L_1 \cup L_2\right\|}$	$\frac{\textit{Noise}}{\left\|L_1 \cup L_2\right\|}$	$\frac{\textit{Displaced}}{\left\|L_1 \cup L_2\right\|}$
			Tensor of the tree	Vector of leaves			
1	{A,{C,D,E}}	{A,{C,D,E}}	1	1	1/4 = 1	-	-
2	{A,{C,D,E}}	{A,{C,D}}	0.61	0.87	3/4 = 0.75	1/4 = 0.25	-
3	{A,{C,D}}	{A,C}	0.416	0.82	2/3 = 0.67	1/3 = 0.33	-
4	{A,{C,D,E}}	{A,{C,D,F}}	0.408	0.75	3/5 = 0.6	2/5 = 0.4	-
5	{A,{C,D,E}}	{A,C}	0.29	0.71	2/4 = 0.5	2/4 = 0.5	-
6	{A,{C,D,E}}	{A,{C,F}}	0.116	0.67	2/5 = 0.4	3/5 = 0.6	-
**	********	**********	*****	*****	**********	***********	***********
7	{{A,B},{C,D}}	{{A,B},{C,D}}	1	1	4/4 = 1	-	-
8	{{A,B},{C,D}}	{A,{C,D}}	0.136	0.87	3/4 = 0.75	1/4 = 0.25	-
9	{A,{B,D}}	{A,{C,D}}	0.03	0.67	2/4 = 0.5	2/4 = 0.5	-
10	{{A,B},{C,D}}	{A,{B,D}}	0.02	0.87	2/4 = 0.5	1/4 = 0.25	1/4 = 0.25
11	{{A,B},{C,D}}	{{A,C},{B,D}}	0.008	1	2/4 = 0.5	-	2/4 = 0.5

Here, the trees in all these rows are constructed using two different kinds of tree composition templates. Both of these templates have two levels of compositions. For both templates, the top level composition is skewed towards conjunction (i.e. for a composition {A, B}, the composition parameters are $h_{AB} = 0.8$, $h_A = h_B = 0.2$). The use of composition parameters was explained in section 3.6.7 of Chapter III. In one template the lower level composition is equally balanced between conjunction and disjunction (i.e. for a composition {C, D, E or F}, the composition parameters are $h_{CDE/F} = 0.19$, $h_{CD} = h_{DE/F} = h_{CE/F} = 0.15$, $h_C = h_D = h_{E/F} = 0.12$), and in the other case it is skewed towards conjunction (i.e. $h_{AB} = 0.8$, $h_A = h_B = 0.2$). The trees/descriptors (D_1, D_2) in row # 1 to 6 in Table 7.1, are composed using the first kind of template described above, and the ones in row # 7 to 11 are based on the second kind of template. These particular compositions were used because they have practical use. For example the compositions (trees) in row # 1 to 6 can be used as templates to disambiguate a concept “A”. For example A = “store”, which defined by a set of alternative attributes, say C = “sale”, D = “purchase”, E = “supply” and F = “buy”.

Table 7.1 illustrates that for different kinds of compositions as shown in row # 1 to 11, the tensor similarity is consistent with the favorable effect of overlap and unfavorable effect of noise and displaced ratios. The similarity values derived according to the tensor model consistently indicates that the tree similarity decreases with decrease with overlap ratio and increase of noise and displaced ratios. This shows that the tensor model is performing the basic task. However the similarity according to the vector model is not always consistent with these ratios. For example for the comparisons in row # 7 to 11, even though the trees progressively get dissimilar, i.e. the overlay ratio decreases and noise and displaced ratio increase, but the vector based similarity value does not monotonically drops, as it should. This indicates that the tensor model represents and compares trees (and composite meanings) more faithfully than the vector model.

In addition, the tensor model has other desirable behaviors due to the certain properties, which are explained below. The effect of these properties are additive.

7.4.3 Property I: Composition information is included

The tree comparison in row 10 and 11 of Table 7.1 above, illustrates the composition property of the tensor based model. This comparison shows that that tensor similarity measure can distinguish trees with similar leaves having different compositions, but vector based similarity can not. This illustrates that the tensor model does a better job in discerning dissimilar compositions (trees) and meanings compared to the vector model.

The same insight, as presented above, is deduced using algebraic logic but in a different manner. Using algebra, we will show that, when there is a composition which is different then the tensor based similarity measure gives lower value than the vector method. Thus we can assert that the tensor model is able to discern the compositions more faithfully than the vector model. To show this, we consider two normalized vectors: $V^1 = x_a^1\vec{a} + x_b^1\vec{b} + x_{c/d}^1\vec{c}$, $V^2 = x_a^2\vec{a} + x_b^2\vec{b} + x_{c/d}^2\vec{d}$ and two normalized tensors: $T^3 = x_a^3\vec{a} + x_b^3\vec{b} + x_{c/d}^3\vec{c} + x_{(b,c/d)}^3\vec{b}\vec{c}$, $T^4 = x_a^4\vec{a} + x_b^4\vec{b} + x_{c/d}^4\vec{d} + x_{(b,c/d)}^3\vec{b}\vec{d}$. For these vectors and tensors, the scalar coefficients are shown as x_i^j, where the subscript i denotes the associated basis vector and the superscript j denotes the entire meaning vector/tensor. As these vectors and tensors are normalized, hence:

$$\left(x_a^1\right)^2 + \left(x_b^1\right)^2 + \left(x_{c/d}^1\right)^2 = \left(x_a^2\right)^2 + \left(x_b^2\right)^2 + \left(x_{c/d}^2\right)^2 = \left(x_a^3\right)^2 + \left(x_b^3\right)^2 + \left(x_{c/d}^3\right)^2 + \left(x_{(b,c/d)}^3\right)^2 = \left(x_a^4\right)^2 + \left(x_b^4\right)^2 + \left(x_{c/d}^4\right)^2 + \left(x_{(b,c/d)}^4\right)^2 = 1 .$$

We consider that vectors V^1 and V^2 have similar kind of composition templates but having different compositions $\{\vec{b}, \vec{c}\}$ and $\{\vec{b}, \vec{d}\}$. Therefore we consider these two vectors have similar weights for their basis vector components, so: $x^1{}_a = x^2{}_a$, $x^1{}_b = x^2{}_b$ and $x^1{}_{c/d} = x^2{}_{c/d}$. We also consider that the tensor T^3 represents the same meaning which is represented by the vector V^1. Here the composition between terms $\vec{b}$ and $\vec{c}$ are considered as simple disjunction in the vector V^1, whereas it is considered as the composition $\{\vec{b}, \vec{c}\}$ in the tensor T^3. Thus we consider $x^1{}_a = x^3{}_a$. Similarly we consider tensor T^4 represents the same meaning which is represented by vector V^2 and therefore $x^2{}_a = x^4{}_a$. We also consider that tensors T^3 and T^4 have similar kind of composition templates, so $x^3{}_a = x^4{}_a$, $x^3{}_b = x^4{}_b$ and $x^3{}_{c/d} = x^4{}_{c/d}$. This means $\left(x_b^1\right)^2 + \left(x_{c/d}^1\right)^2 = \left(x_b^3\right)^2 + \left(x_{c/d}^3\right)^2 + \left(x_{(b,c/d)}^3\right)^2$. Here if $x^1{}_b > x^1{}_{c/d}$ then $x^3{}_b > x^3{}_{c/d}$ or if $x^1{}_b < x^1{}_{c/d}$ then $x^3{}_b < x^3{}_{c/d}$. As $x^3{}_{(b,c/d)} > 0$, therefore, $x^1{}_b > x^3{}_b$.

The similarity between V^1, V^2 is given as $s^{1,2} = V^1 \bullet V^2 = x_a^1 x_a^2 + x_b^1 x_b^2 = \left(x_a^1\right)^2 + \left(x_b^1\right)^2$ and similarity between T^1, T^2 is given as $s^{3,4} = T^3 \bullet T^4 = x_a^3 x_a^4 + x_b^3 x_b^4 = \left(x_a^3\right)^2 + \left(x_b^3\right)^2$. As $x^1{}_b > x^3{}_b$, therefore $s^{1,2} > s^{3,4}$. This shows that the tensor based similarity measure has more discriminating power than the vector model. This logic can be extended for cases when there are more components in both: within the composition or outside it.

7.4.4 Property II: A partial set can represent composite meaning

Two similar composite meanings may be expressed by two different but overlapping set of elementary meanings (i.e. they share many common elements) and yet they will be recognized as similar ones by the tensor model (see row 1 & 2 in Table 7.1 above), as it is in case of vector model. This property is useful to identify similarity between contexts which are described by a slightly different set of elementary meanings. This is congruent to several understandings from cognitive science domain as presented in section 2.4.7 in Chapter II. This property can be also established by algebraic logic as presented below.

Here we will show that when a composition has an extra attribute, even then the tensor similarity measure will yield a non zero value, indicating that the extra term does not make the composition entirely dissimilar than the tensor which does not have that term. This effect is opposite to the one due to the property presented in the last section. So by the proof of this property II, it is implied that the behavior of the tensor model is not polarized to one single property but it is a mix of multiple properties. Here we consider two normalized tensors: $T^1 = x_b^1\vec{b}$, $T^2 = \{\vec{b},\vec{b1}\} = x_b^2\vec{b} + x_{b1}^2\vec{b1} + x_{(b,b1)}^2\vec{b}\vec{b1}$, where the extra term in the composition is $\vec{b1}$.

We consider that tensor T^1 and T^2 convey the same meaning, and it is just that one attribute $\vec{b1}$ in the composition $\{\vec{b},\vec{b1}\}$ is missing in the representation T^1. The similarity between T^1, T^2 is given as $s^{1,2} = T^1 \bullet T^2 = x_b^1 x_b^2$, which is smaller than 1 but a non zero value. This shows that even though the tensor based similarity measure has more discriminating power w.r.t. a composition, but it does not yield a zero similarity when a term, e.g., $\vec{b1}$ in the composition is different or missing. This logic can be extended for cases when there are more components in both within the composition or outside it.

7.4.5 Property III: Higher level compositions are more important

The differences or similarities of elements at higher level compositions in a tree have larger impact on the similarity of the entire tree. The tensor based comparison in Table 7.2 below illustrates this (only the dissimilarity case is shown). In row 1, the trees have dissimilarity in the higher level of the composition (dissimilar leaves are D and H). Whereas in the row 2, the dissimilarity is at the lower level (dissimilar leaves are G and I). All compositions are uniform mix of conjunction and disjunction compositions. In this example, row #1 has two trees, where, the leaves D and H are dissimilar ones located within the higher level composition. The similarity between these two trees is 0.42. Whereas, row #2 has two trees, where the dissimilar leaves are G and I located inside the lower level composition. The similarity between these two trees is 0.54. This shows when the higher level compositions are dissimilar then that impacts the similarity value more adversely.

Table 7.2 Importance of higher level compositions

Sl.	D_1	D_2	$D_1 \bullet D_2$
1	{A,{C, D, {E,F,G}}}	{A,{C, H, {E,F,G}}}	0.42
2	{A,{C, D, {E,F,G}}}	{A,{C, D, {E,F,I}}}	0.54

The real world analogy of this property is that two objects will be considered similar if the big picture meanings of objects are similar even though the finer detailed meanings may be somewhat different. This property can be actually used to model this manner how humans compare and interpret meanings.

7.5 Experimental setups for meaning comparator evaluation

7.5.1 Comparator architecture simulator

An in-house developed simulator of the comparator architectures was used for all related experiments. Some of the simulation parameters used here is based on the actual hardware design carried out by [131]. The following baseline parameters were used during these simulations: number of basis vectors *n* in the input coefficient tables (vector/tensors) = 10^4, number of common basis vectors c in the input coefficient tables = 500 (5% of n), BF size m = 131072 (= 2^{17}), number of CAM units b = 32 and number of multipliers p = 16. These particular parameters for BF yield reasonably small false positive probabilities (~ 0.0021) to make the architecture workable. Whereas the rationale for choosing the b and p parameters had been presented in section 4.5.2.3 of Chapter IV. The baseline system used un-partitioned BF and total k (=7) hash functions for BF operations. In some experiments, wherever different values or assumptions were used, those are explicitly mentioned in the figures or in the associated text. The time equivalent of the clock cycles were reported assuming 4Ghz clock.

7.5.2 Optimal software implementation of the comparator

The execution timing for a representative server class processor (Intel Xeon) was measured for software code which computes the dot product (Fig. 7.4). The dot product software code identifies the common basis vectors by first constructing a balanced binary search tree of basis vectors (char strings) and coefficient pairs (with vectors as search keys) from the second table, and searching for each vector from the first table. Thus this searching time is of the order of $O(n_1 \cdot \log n_2)$, where n_1 and n_2 are number of basis vectors in the two input tensors. The balanced tree is implemented using GCC C++ STL's highly optimized map container which implements Red-Black tree. The pseudo code in Fig. 7.4 [131] indicates the code segment for which execution time is measured. This software code is efficient because it uses Red-Black tree based search algorithm having logarithmic execution time compared to naïve linear search time algorithm. In addition the C++ STL's implementation is known to be very optimum in computing practice domain. Therefore we consider this software based comparator implementation as a near optimal one.

```
// Table data structure declarations.
struct rows{ string basis_vector;  float coefficient };
…..
rows coeff_table1[NUM_BASIS_VECT1], coeff_table2[NUM_BASIS_VECT2];
….
// Declaration for map data structure for the red-black tree.
std::map < string, float> rbtree;
std::map < string, float>::iterator rbtree_itr;
….
//Execution time measurement starts here
//Code to construct tree
for (i = 0; i <  NUM_BASIS_VECT1; i++)
    rbtree.insert( std::make_pair( coeff_table1[i]. basis_vector, coeff_table1[i].coeff) );

//Code to search tree
for (i = 0, dot_prod =0; i < NUM_BASIS_VECT2; i++) {
  rbtree_itr= rbtree.find(coeff_table1[i]. basis_vector) ;
  if ( rbtree_itr!=rbtree.end() )
        dot_prod = dot_prod + ( (*rbtree_itr).second *  coeff_table2[counter].coeff);  }
//Execution time measurement ends here
```

Fig. 7.4 Pseudo code for optimum software implementation of dot product

7.5.3 Existing hardware design of dot product processors

We would compare performance of our architecture with existing dot product processor hardware designs proposed in [149], [150] and [151]. The design in [149], which we call “H/w design 1”, offered speedups of 784 times for small number of basis vectors (only 8 basis vectors) when compared to an equivalent software code. The architecture in [150], which we denote as “H/w design 2” requires around 200 clock cycles to find dot product for two vectors having 400 basis vector components, and in the design in [151], which we identify as “H/w design 3”, the execution time was in order of 8.3 million clock cycles for 1024 basis vectors.

7.5.4 Experiments and rationale

To compare performance of our designs and existing alternatives in a clock speed neutral manner, we use clock cycles in lieu of execution time in seconds. This was done to fairly compare the hardware designs that have been implemented and evaluated with different clock speeds in [149], [150] and [151].

To help us optimally design the dot product hardware, we study execution clock cycles, as primary point of interest and number of CAM lookups needed, for greater visibility of the process, for the various operation scenarios as mentioned below:

(1) For different design parameters like: size of BF bit array *m*, number of hash functions *k*.

(2) For different numbers of: basis vectors *n*, and fraction of common basis vectors *c*.

(3) For various design alternatives as presented in section 4.5.5.5.

Queries generated by users generally have small sizes, however search engines often encourage users to use a known object as the reference and search objects similar to the reference object (because such strategy yields superior search performance [8]). Advertising disseminating mechanisms may also use the descriptors of the currently viewed web page to dynamically retrieve advertisements or sponsor's web pages and display them as banners or sponsored links. In all these cases the reference object's semantic descriptor becomes the search query or the semantic query. Such queries have large sizes. So in all our experiments we used semantic descriptors for queries as large as the object's descriptors to simulate worst case scenario.

7.6 Results for meaning comparator evaluation

7.6.1 Performance evaluation of the comparator

7.6.1.1 Execution speed comparison against software implementation

Table 7.3 compares the average execution time in terms of clock cycles (necessary to execute the dot product execution for a pair of tensors/vectors) for the proposed dot product processing architecture against the optimum software code as presented in Fig. 7.4 and reports the speedups for different *c*, the numbers of common basis vectors (terms).

Table 7.3 Superior execution time of the comparator architecture

Number of common basis vectors (c)	Clock cycles for proposed architecture	Clock cycles for Intel Xeon	Speedup (times)
100%	969	$7.686*10^6$	79,320
5%	80	$7.712*10^6$	964,096
1%	42.39	$7.688*10^6$	1,813,666

The typical average number of common basis vectors *c* ~0.1% of the total number of basis vectors *n* (refer section 4.5.2.3, Chapter IV). We observe that the speedups for small value of *c* are in order of ~10^6 or higher. This establishes the superior performance of our proposed architecture in terms of speed. The conclusion drawn from this comparison remains valid across multiple processors. The Intel Xeon (3Ghz version), used in this example, has a maximum instruction per cycle (IPC) value of 4. The IPC indicates the maximum level of instruction level parallelization that can be achieved per processor core. All other high performance sequential processors have IPC of very similar order. Hence this speedup can not be significantly dropped any further on a traditional CPU core. Multi-core CPU/GPUs are available with a limited number of cores (not in order of hundreds of thousands), hence can't give very large (~hundreds of thousands) speedup, unless super computers with several thousands of processor are used. Using super computers are not practical because they are too expensive, they have high processing (setup) time overheads and consumes significant amounts of power and space to be comfortably accommodated as a small component in a search engine. In fact multi-core or multi-processor system based parallelization strategy will not deliver large amount of speedups (~10^5 times) for dot product computation for reasons explained in section 4.5.2.2 in Chapter II.

This rationale is briefly repeated here for sake of continuity. The parallel threads in the dot product computation are simple and extremely short computations. Multi-processor based parallelization requires significant overhead (time) to distribute the processing task (i.e. distribute all the necessary inputs) to multiple cores/processors. Similarly, there is also a large overhead to consolidate the processing (i.e. bring all the data back to a single point). These overheads are absent in our proposed architecture. Our architecture is most suitable here because each individual parallel thread (computation) is short. In contrast a multi-core system is more suitable when each thread is long enough to amortize the distribution and consolidation overheads. Hence our design will do better than GPUs also in terms of speed. Moreover a system with 10^6 or more processor cores, in theory, can give the same order of speedup as our design, but it will need more power and space compared to our design which is on a single chip. The data presented in Table 7.3 illustrates this point.

7.6.1.2 Execution speed comparison against existing hardware designs

Fig. 7.5 shows the comparison of speedup of our architecture against that of other available hardware designs. The speedup figures of the existing hardware designs (and our proposed architecture) are either computed as ratios of clock cycles of execution time of the existing hardware (and our proposed architecture) against the clock cycles of the efficient software code, as in Fig. 7.4, or simply as ratio of average clock cycles of our design vs. that of the existing hardware designs.

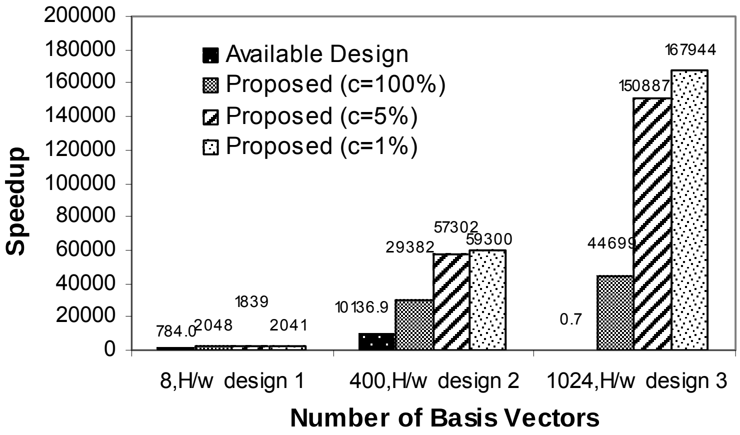


Fig. 7.5 Comparison of speedup against other designs

We show comparison for c=100%, 5% and 1%. The other hardware designs do not take into consideration the number of common basis vectors to avoid the unnecessary computations like we did in our design (refer section 4.2 in Chapter IV). In addition, our architecture is consistently doing better due to fine grained parallelism in our design for large meaning vectors (number of basis vectors = 400, 1024). Such parallelism has not been exploited by other hardware based designs [149], [150], [151], which carry out the computations sequentially.

In Table 7.4, we present a comparison of our design with those presented in [149], [150] and [151] ("H/w design 1,2 & 3") and show and show the factors of speedup by which our design performs better. For large meaning vectors (number of basis vectors = 1024) our design gives a speedup increase of 124,664 times for c=1% and 52,201 times for c=100% compared to the "H/w design 3". In case of smaller vectors (having smaller number of basis vectors) a much lower speedup is due to the overhead in our design. We do not optimize for extremely small vectors, this is because in [13] we have shown that the typical number of rows (basis vectors) is in order of few hundreds (10^3). Our design will reduce the constraint on the size of vectors that can be used, because our design's speedup is better for larger vector sizes. In search application, large vectors are preferable for precise meaning representation, comparison and searching, hence this overhead is not a problem.

Table 7.4 Speedup comparison with other hardware designs

Number of Basis Vectors	Compared Against	Improvement in speedup (times)		
		c = 1%	c = 5%	c = 100%
8	H/w design 1	2.60	2.34	2.61
400	H/w design 2	5.85	5.65	2.90
1024	H/w design 3	245,551	220,612	65,354

7.6.1.3 Characterization of the basic comparator

Fig. 7.6 shows the number of CAM lookups ("CAM" & "CAM_part"), and Fig. 7.7 shows the similarity comparison execution clock cycles ("Cyc" & "Cyc_part" for unpartitioned and partitioned BF designs).

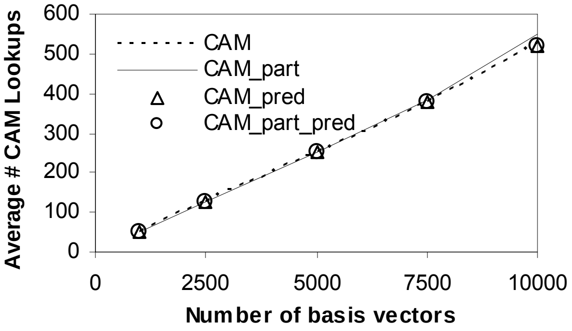


Fig. 7.6 Number of CAM lookups for different number of basis vectors

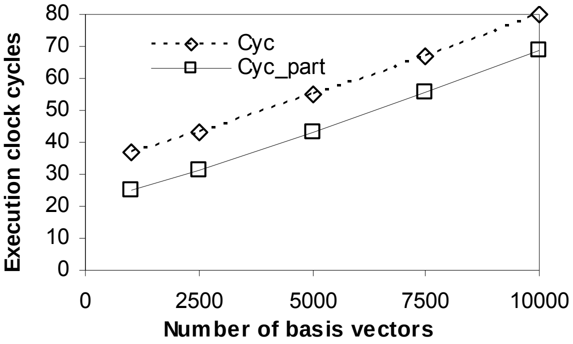


Fig. 7.7 Execution cycles for different number of basis vectors

Both number of CAM lookups and clock cycles increase with number of basis vectors *n* in the input. However for a large *n* = 10^4, c = 10^3, the times is still within 80 clock cycles or 20 nanosecond with 4Ghz clock for all cases. The CAM lookup values predicted ("CAM_pred" & "CAM_part_pred" for unpartitioned and partitioned BF designs) by the equations (2.5) and (2.6), are shown in the same figure with number of CAM lookups (for details refer section 2.12 and 4.5.4). This illustrates that there is a good agreement between experimental (simulated) values and their predictions.

Fig. 7.8 shows the increase in number of CAM lookups and Fig. 7.9 shows the same for execution clock cycles when number of common basis vectors *c* increases. With a maximum value of *c* (100% of *n*) the time/clock cycles is still 969 cycles (242.25 nanosec) for un-partitioned BF design and 957 cycles (64.25 nanosec) for partitioned BF alternative.

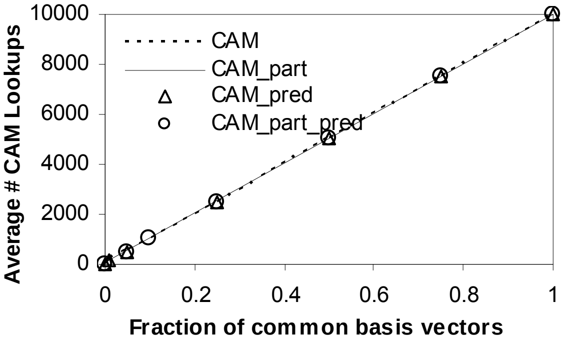


Fig. 7.8 Number of CAM lookups for various numbers of common basis vectors

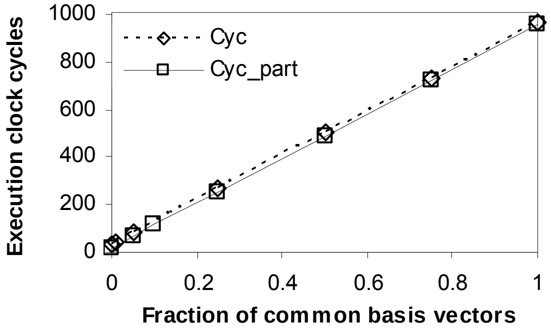


Fig. 7.9 Number of execution cycles for various numbers of common basis vectors

Fig. 7.10 shows the effect of variations of BF bit array size *m* on number of CAM lookups and Fig. 7.11 shows the same for execution clock cycles. With the small (worse case) value of *m* (=16,384) the time/clock cycles is still within 340 cycles (85 nanosec) for both BF designs. Increasing *m* does not reduce number of cycles and CAM lookups beyond a certain level, so there is no point to over designing the BF by allocating more bits than necessary. For a given *m* the knee point depends on number of CAM units and multipliers used. These figures show that it is possible to choose a smaller BF bit array size of 65,535, to reduce circuit complexity, without significant impact on the execution time.

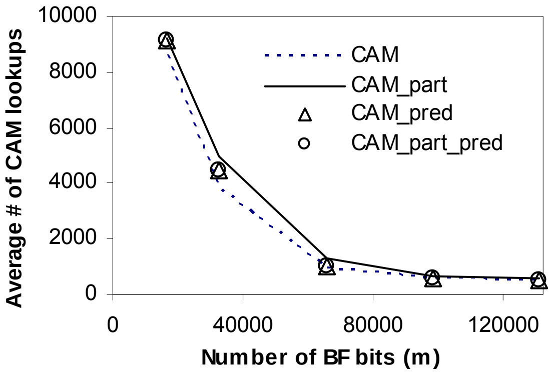


Fig. 7.10 Number of CAM lookups for various BF sizes

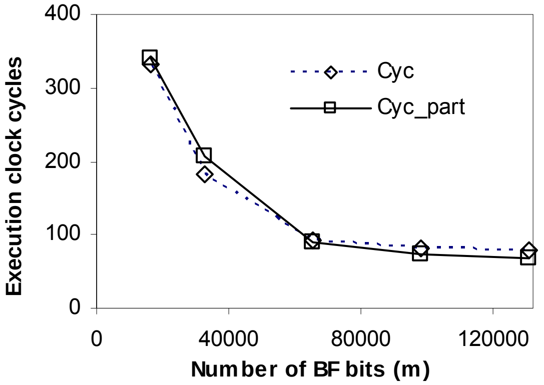


Fig. 7.11 Number of execution cycles for various BF sizes

Fig. 7.12 shows the effect of variations of number of BF hash functions *k* on CAM lookups and Fig. 7.13 shows the same for execution clock cycles. We observe that an optimum value of *k* (between 7 and 10, for given *n*, *m*) exists for number of CAM lookups, however as *k* determines the number of clock cycles, so there is a benefit in choosing lower value of *k*, for example *k* = 7 for that design.

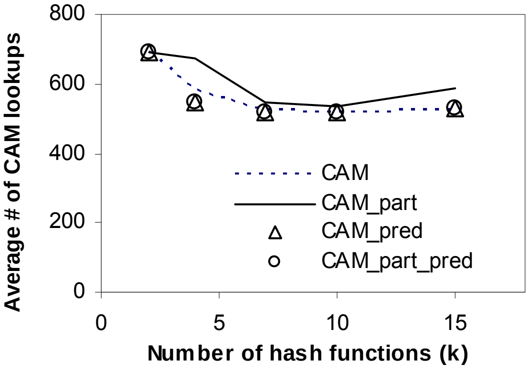


Fig. 7.12 Number of CAM lookups for various number of BF hash functions

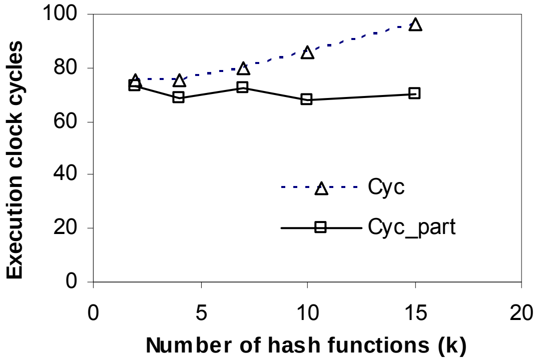


Fig. 7.13 Number of execution cycles for various number of BF hash functions

We have characterized the time response behavior for the proposed architecture for a wide range of input and design parameters to demonstrate the worst and average case behaviors. For example, on an average the *c* parameter value will in the lower side depending on the SRN and routing table lookup implementations. So to read out the average execution time for a large number of comparisons (e.g., 1000) from Fig. 7.9, we should use a very low *c* value (e.g. 0.1% of *n*). Whereas to read out the worst case response time for a single comparison, we should use a *c* value of 100% *n*.

7.6.1.4 Execution speed of alternative comparator architectures

The average number of CAM lookups and execution clock cycle for all the architecture alternatives, as discussed earlier in section 4.5.5.5, are presented below in Table 7.5. The execution time reported here are for m = 131072, k =7, n = 10,000 and c = 5% of n. As option 16 gives the best time ~61 clock cycles and it is also the least expensive in terms of circuit complexity, so it is the best architecture to implement.

Table 7.5 Performance comparison of alternative architectures

Architecture alternatives #	Complexity rank	Average CAM lookups	Cycles
1	D	524.023	79.994
2	D	548.859	68.723
3	C6	522.552	79.977
4	C5	519.907	79.963
5	C4	521.751	79.966
6	C3	520.298	79.963
7	C2	520.207	79.96
8	C1	520.295	79.955
9	C1	520.018	67.968
10	B	528.148	72.004
11	A6	520.216	71.966
12	A5	520.066	71.969
13	A4	522.322	71.978
14	A3	519.898	71.953
15	A2	522.995	71.982
16	A1	565.965	61.113

7.6.2 Setup time for the comparator

The time required to load the BF values in the bit addressable memory and the coefficient table values in the CAM and RAM units is 17 clock cycles (4.25 nanosec) for in all cases with unpartitioned BF design and only 11 clock cycle (2.75 nanosec) for all other cases which implements the partitioned BF design. The rationale for this small setup time has been explained in section 4.5.4 in Chapter IV.

7.6.3 Memory space scalability analysis for the proposed comparator

To evaluate the feasibility of the descriptor comparison processing on a single chip, here we examine the memory space requirement of the proposed approach. When standard controlled terms, as used at the leaves in the concept tree (as explained in Chapter III), convey composite meanings themselves, then small sized trees (number of leaves ~15) represent meanings of objects with sufficient specificity. These trees tend to generate limited number of basis vectors ($<10^4$) (e.g. as observed in [13]). For an object key tensor with 10^4 basis vectors (assumed worst case), 16 bit fixed point scalar representation and with extremely conservative BF parameters: $m = 131072$ and $k = 7$, the uncompressed key size is $10^4 \cdot (64+7\log_2(131072)+16)/8 \approx 249$KBytes. This space requirement of registers and RAM units can be accommodated on a single chip. This indicates that memory space requirement is not a hurdle towards feasibility of this dot product co-processor chip. Average keys with smaller BFs would be much smaller and a large number of them can fit in high density memories after compression. The memory requirement to store ~1000 destination keys in the semantic router will be in order of 249Mbytes. Such requirement can be easily satisfied by the existing memory device capacities. So buffering a large number of routing table row keys does not pose a significant problem.

For purpose of transmission across network or a system interconnect, we use a compact version of the descriptor which do not include the BF indices (the third column of the coefficient table as in Fig. 4.3 in Chapter IV). This third column can be regenerated from the vector ids (the first column of the coefficient table). So for purpose of transmission the size of key is much smaller, in order of $10^4 \cdot (64+16)/8 \approx 100$KBytes only. These key sizes are also stored in the buffer associated with the network interface as shown in Fig. 6.10 in section 6.7.2.1 of Chapter VI. Therefore buffering a large number of messages containing these keys will not be a problem as these can be easily accommodated in available high density memory devices.

7.7 Experiments and setups for Semantic Routed Network evaluation

7.7.1 Semantic Routed Network simulator

To evaluate the performances of the proposed mechanisms we developed a Semantic Routed Network (SRN) simulator to carry out simulation studies. The router and index nodes are bestowed with certain semantic descriptors. Each router and index node was bootstrapped to some random router nodes. The network links have finite delay. A global virtual clock and scheduler maintains the pace of the network maturity, node activity, network delay and response timeouts. All nodes are activated at once at the beginning of the simulation. Periodically message delivery requests were injected to random router nodes and delivery times were noted after a brief timeout period. Only a small sub-set of the routers and index nodes periodically send out search queries at any given point of time (as mentioned in section 6.1.3 in Chapter VI). This ensures that such queries from all the index nodes and routers are spread out across time.

7.7.2 Experiments and rationale

We compare the performance of the SRN against three criteria: (i) estimated expected end-to-end routing response time or message delivery time (hops required by a message to reach destination, lesser is better); (ii) end-to-end routing success rate (instances of destinations reached if they exist, higher is better) observed within a time frame of 6 messaging delays; and (iii) message overhead (number of messages generated in the network to carry out a search). The network is dynamic because the routers are constantly seeking each other and exchanging index/routing table entries. By this way the network is using its own capabilities to evolve, therefore virtual clock cycles indicate network maturity and usage.

Two different sets of simulations were run. In the first case we wanted to see the best performance of the network. Here we evaluate a SRN paradigm which is built with a large number of small semantic routers. When the semantic routers are small and inexpensive it is possible to deploy them in large quantities, in an order of magnitude which is similar to that of the deployed destinations/resource nodes. Values of some parameters used were: number of index nodes = 800, number of routers = 200, known router list size = 2, routing table max row length = 5, routing table max column width = 5, periodicity of resource and router nodes were 10 virtual clock cycles, message delivery request injection rate was 5 queries per 20 clock cycles, the timeout period after which the message delivery success was noted = 20 cycles, network link delay = 3, duplicate message cache timeout = 10 clock cycles respectively. Routing table size, cache time out and effective cache size were kept very small to simulate the scalability hurdle.

In the second set of experiments we wanted the network to under-perform by turning off some of the performance enabling and optimization features, so that we can observe the performance with and without some of these performance enabling features. For the experiment with large routers, we used 20 routers and table row length and width = 50, instead of 5 and timeout period=6.

7.8 Results for Semantic Routed Network

7.8.1 Message routing capability and self organizing behavior

For this simulation the first set of simulation parameters were used. Fig. 7.14 shows how routing success improves as the SRN self-organizes to full maturity at 841 cycles. At full maturity the success rate reaches ~100%. This increase in routing success translates to increase in search success rates, because higher percentage of query messages can reach the destination objects, and they can then respond back.

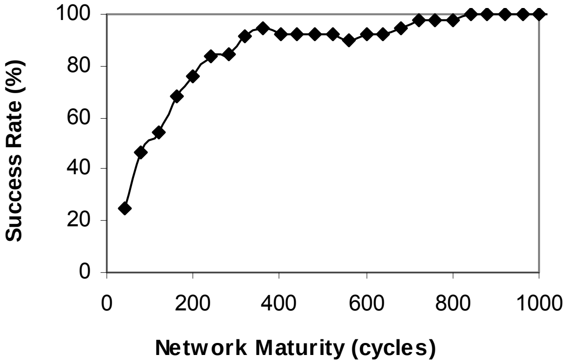


Fig. 7.14 Improvement of routing success rate with network maturity.

Fig. 7.15 shows how the expected search response also improves over the life-time of SRN. The "expected" number of hops to traverse before reaching destination drops to ~1.89 hops after the SRN has self-organized for 1000 virtual clock cycles.

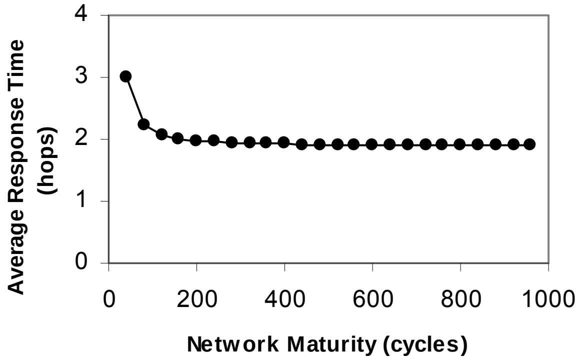


Fig. 7.15 Improvement of routing response time with network maturity

The network had a random topology to begin with, so the routing success rate and response times for a random topology are given by the points at 25 clock cycles (when the first observation took place) in Fig. 7.14 and Fig. 7.15. These points correspond to 25% and 3 hops. Therefore we can compare these performances for random and small world network topologies as presented in Table 7.6 (from the values from Fig. 7.14 and Fig. 7.15). Based on this we conclude that small world topology (success rate ~100%, number of hops =1.89) performs better than random network topology (success rate 25%, number of hops = 3 hops).

Table 7.6 Routing and search performance comparison

Topology	**Message routing success rate (≈search recall rate)**	**Message routing response time (≈ search response time)**
Random network	25%	3 hops
Small world network	~100%	1.89 hops

7.8.2 Effectiveness of duplicate message suppression techniques

Fig. 7.16 shows the improvement trend of the message overhead for the proposed SRN. When the SRN matures, only 4.8 messages are generated per query. This small value can be compared with a very large overhead in order of hundreds and thousands in other search networks [108].

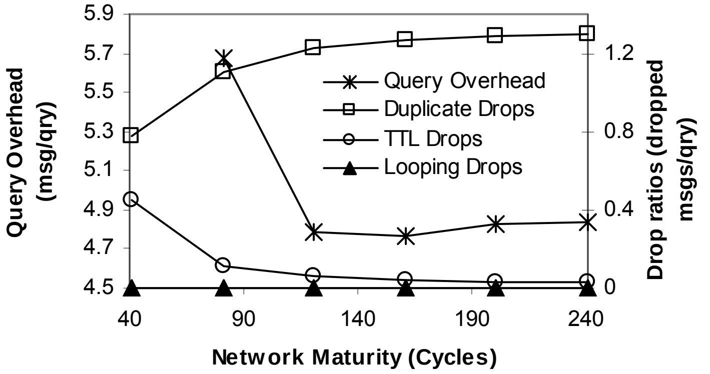


Fig. 7.16 Message overhead, duplicate, TTL & looping message drop ratios

This demonstrates the effectiveness of the duplicate message detection cache, time to live (TTL) and loop detection mechanism working together. We see that the messages dropped due to duplicate detection by cache (indicated by "Duplicate Drops" in the figure) are large compared to drops due to TTL ("TTL Drops"). This proves that TTL alone is not sufficient to eliminate network storms in a small world. Around 0.03% of messages were dropped because they were found looping (shown as "Looping Drops" in the figure). These looping messages were not eliminated by the cache and TTL mechanism. This indicates the need for a separate loop detection mechanism.

7.8.3 Effectiveness of routing table optimization techniques

Fig. 7.17 compares the success rate in three different SRNs: (a) the first one with routers which neither enjoys prioritized eviction, nor have routing table compression algorithms ("RT w/o optimization"); (b) the second one with routers that only have prioritized eviction of routing table and "known router" list entries ("RT w/ prioritization"); and (c) the third one which has routers with full routing table optimization ("RT fully optimized").

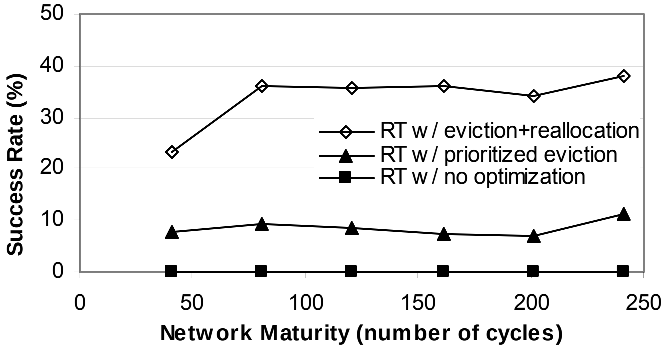


Fig. 7.17 Role of different routing table optimization algorithms

This experiment was conducted without the process that explicitly programs the semantic routing tables with the shortest paths. This was done to observe the impact of only the semantic routing table optimization algorithm. In the first case hardly any successful semantic routing took place, even though the routers were looking for interesting resources and routers. In the second case some successful semantic routing took place, and the routing performance improved when the destination reallocation algorithm was applied in the third case. This comparison demonstrated the roles of the prioritized eviction and row and column reallocation algorithms.

7.8.4 Performance of small routers

Fig. 7.18 compares the routing success rate for two SRN paradigms: (a) large number of small semantic routers having small number of optimized routing tables; (b) small number of large routers having large non-optimized routing tables.

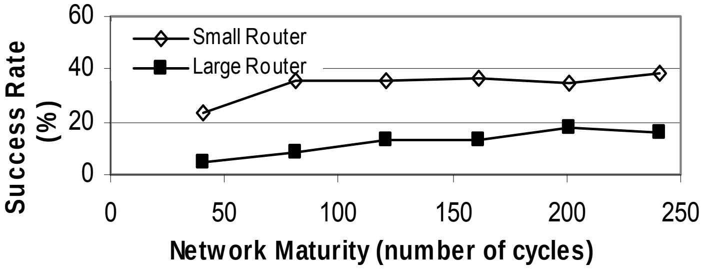


Fig. 7.18 Performance comparison between small & large Semantic Routers

This experiment was also conducted without the process that explicitly programs the semantic routing tables with the shortest paths to observe only the impact of routing table optimization algorithm. This shows that small optimized routers (35% success rate, response time of 3.3 messaging delays) outperform large routers (17.5% success rate, response time 4.4 messaging delays) when used in large numbers. Therefore it is possible to replace large routers with large numbers of small routers. For the case (a), after 80 virtual clock cycles the success rate saturates to the limiting value, which indicates the capacity limit of the semantic routers and the given set of optimization techniques.

The fact that smaller routers can yield better performance than the larger ones when equipped with the aforementioned optimizations techniques, have significance in our SRN design. This means that we can use inexpensive small routers in lieu of large ones. In certain applications these routers may not need special comparator hardware, because slow software based comparison can still be workable solution when number of comparisons are small.

7.9 Storage scalability of the proposed distributed index

The proposed distributed index with SRN will use similar clustering architecture to implement the index pools, as proposed in [4] (i.e. Google's architecture) for storage and throughput scalability. This architecture has the ability to index billions of documents/webpages [3] and serve 3500 queries per second [1]. Therefore we argue that our proposed architecture will have the similar throughput and ability to index to large volume (~billions) of documents/web, as it proposes to implement the same clustering scheme. However, in our proposed design we are responsible for ensuring that the semantic routers, which are our inclusion, should have the required response time and throughput capacity. We present that analysis in this section below.

7.10 Semantic Routed Network response time and throughput analysis

7.10.1 Analysis approach

In the following section we will show that the proposed distributed search engine design which incorporates SRN, will satisfy the time response and throughput requirements. This analysis uses estimated ball park figure of the response and execution times of SRN and semantic routers. These approximate time estimates are used to make the point that workable design of a SRN and semantic routers are indeed possible. Here we work with order of magnitudes of the times instead of their exact and accurate figures. This is because the accurate times estimation is possible only when a detailed engineering design of all the components of the proposed system is carried out and some of them have been implemented to measure their time response behaviors. On the other hand the order of magnitudes can give us a fair feasibility assessment of the proposed scheme quite early in the planning stage (i.e. the present moment) so that we can decide whether to invest resources to carry our a more detailed engineering design, or not. Here we deliver that feasibility assessment which is useful for planning purpose.

Here we will demonstrate that insertion of SRN in the search engine does not significantly perturb the time response and throughput capacity of the entire system. Fig. 7.19 shows the time response analysis of a typical distributed search engine. The response time of this system has three components, as illustrated by equation (7.4) and repeated here- $T = t_{\text{qryprocess}} + t_{\text{index}} + t_{\text{docserver}}$, where $t_{\text{qryprocess}}$, t_{index}, $t_{\text{docserver}}$ are the response times of query processor, index and document server sub-systems respectively.

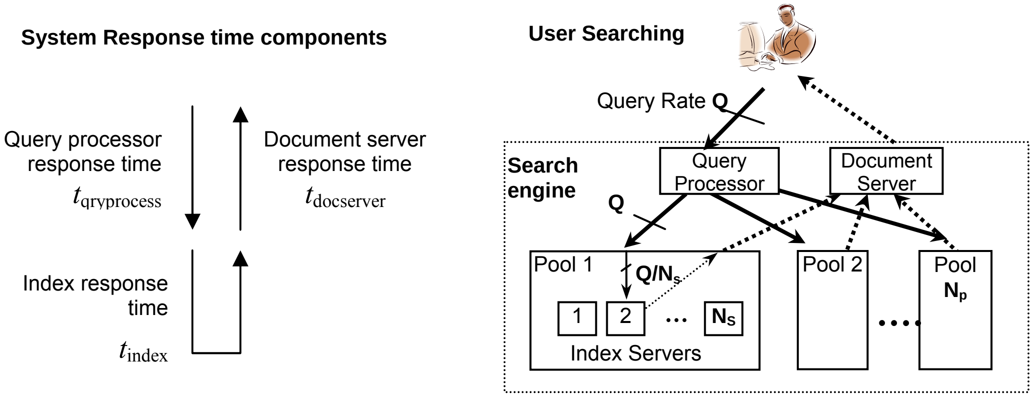

System Response time components
Query processor response time $t_{\text{qryprocess}}$
Document server response time $t_{\text{docserver}}$
Index response time t_{index}
User Searching
Query Rate Q
Search engine
Query Processor
Document Server
Q
Pool 1
Q/N$_s$
1
2
N$_S$
Index Servers
Pool 2
Pool N$_p$

Fig. 7.19 Time response analysis of a typical search engine

On the other hand, Fig. 7.20, shows the response time analysis of the proposed distributed search engine that incorporates SRN.

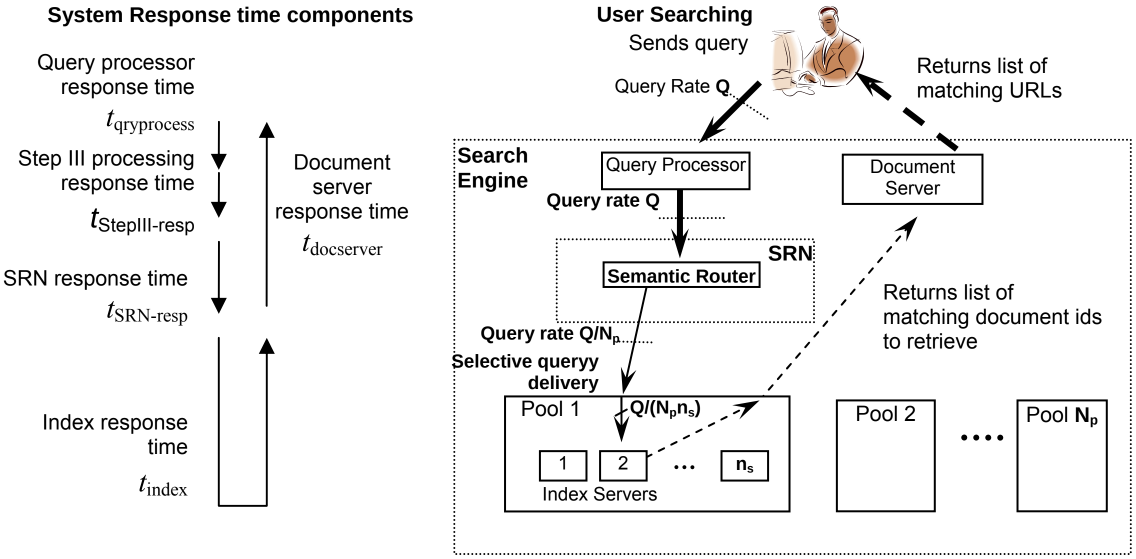

System Response time components
Query processor response time $t_{\text{qryprocess}}$
Step III processing response time $t_{\text{StepIII-resp}}$
SRN response time $t_{\text{SRN-resp}}$
Index response time t_{index}
Document server response time $t_{\text{docserver}}$
User Searching
Sends query
Query Rate Q
Returns list of matching URLs
Search Engine
Query Processor
Document Server
Query rate Q
SRN
Semantic Router
Returns list of matching document ids to retrieve
Query rate Q/N$_p$
Selective queryy delivery
Pool 1
Q/(N$_p$n$_s$)
1
2
n$_s$
Index Servers
Pool 2
Pool N$_p$

Fig. 7.20 Time response analysis of proposed search engine which incorporates SRN

Incorporation of the SRN increases the response time of the search engine by an additional amount ($t_{\text{StepIII-resp}}$ + $t_{\text{SRN-resp}}$), where $t_{\text{StepIII-resp}}$ is the response time of the step III processing which is incorporated in the query processor sub-system (refer Fig. 3.2 in section 3.3, Chapter III) and $t_{\text{SRN-resp}}$ is the response (or flow) time of the SRN. Thus the response time T_{prop} of the proposed design is:

$$T_{prop} = t_{\text{qryprocess}} + t_{\text{StepIII -resp}} + t_{\text{SRN -resp}} + t_{\text{index}} + t_{\text{docserver}} \tag{7.11}$$

In the following sections we will show that this response time T_{prop} is not significantly different than the response time T of a typical search engine, as given by equation (7.4). We will also show that it is possible to design the SRN with sufficient throughput capacity to handle the query traffic rate, so that the SRN does not become the bottleneck.

For simplicity we assume that the additional response time needed to carry out step III is similar to its execution time (refer section 2.11.3, Chapter II for the rationale). Therefore this time is considered equal to the one given by equation (4.1) in section 4.5.1.3, Chapter IV. Therefore the average response time is given as:

$$t_{\text{StepIII -resp}} \approx T_{\text{StepIII -exe}} = {}^{n_{byte}}\!/_{u} \tag{7.12}$$

Where n_{byte} is the average number of bytes in the basis vector terms in the tensor model which is considered = 40 bytes based on our experiences from [13]. Whereas u is the loop unroll factor which is taken as = 1 here. Therefore $t_{\text{StepIII-resp}}$ = 40 clock cycles, which is equivalent to 10 nanoseconds.

For simplicity we model the entire SRN as a single queue M/M/n system. This queue system had been explained in Chapter II, section 2.11. Therefore we can assert that the flow time $t_{\text{SRN-resp}}$

required to route a message through the SRN includes two components: the time $t_{\text{SRN-wait}}$ that the message have to wait for the SRN to actually begin the routing process and the time $t_{\text{SRN-exe}}$, which is the actual execution time that the SRN takes to service (route) the message. Therefore the three variables: $t_{\text{SRN-resp}}$, $t_{\text{SRN-wait}}$, $t_{\text{SRN-exe}}$ are related to each other by:

$$t_{\text{SRN-resp}} = t_{\text{SRN-wait}} + t_{\text{SRN-exe}} \tag{7.13}$$

We shall show that the time $t_{\text{SRN-exe}}$, is indeed small, therefore we can design a SRN to have small response time $t_{\text{SRN-resp}}$ and high throughput capacity. This is based on the insights gained from queuing theory, as presented earlier in Chapter II, section 2.11. There, we had shown how we can deliver the required throughput capacity and response time by choosing *n*, the number of processors (i.e. routers in case of SRN) in a system, as long as the execution time is small compared to the required response time.

To establish this we will show that $t_{\text{SRN-exe}} << T$, so that it is possible to chose a small value of *n*, so that we get a $t_{\text{SRN-resp}} \approx t_{\text{SRN-exe}}$. This $t_{\text{SRN-resp}} << T_{\text{prop}}$ and therefore response time of the search engine with SRN is $T_{\text{prop}} = T + t_{\text{SRN-resp}} \approx T$. In other words, as $t_{\text{SRN-exe}}$ is very small compared to response time *T* of a typical search engine that does not use SRN, therefore we can deploy a very small number of concurrent routers (acting as processors or servers) to get the required SRN throughput capacity and yet manage to leave the response time of the search engine unperturbed. We will establish this for different kinds of SRN and router implementations as explained in the following sections.

7.10.2 Alternative SRN implementations

We considered a distributed index system model where there are ~1000 index nodes (i.e. N_{P} =1000, Fig. 7.20). In this system two alternative SRN implementations are possible. One implementation involves a single large router. This single semantic router will forward queries to any one of the 1000 index server pools (destinations). Therefore this semantic router will have 1000 rows in its semantic routing table and will need at most 1000 key comparisons, due to the chosen semantic routing table mechanism as explained earlier in Chapter V, section 5.1.7.

The other SRN implementation involves 200 small routers, connected to each other in P2P fashion, as modeled in the SRN simulation presented in section 7.7. Each of these small semantic routers will have 5 rows in its semantic routing table.

For each of these SRN implementations, the descriptor comparator in the semantic routers can be implemented by two methods. In one design, the meaning descriptor comparison can be carried out using the proposed comparator accelerator hardware as explained in Chapter IV, section 4.5.4. In the other design, we can use the meaning comparison software code, as presented in Fig. 7.4, in section 7.5.2. The time response estimate calculations are presented in the following sections for all these four cases (i.e. 2 SRN designs * 2 router implementations). The assumptions behind these calculations are also presented.

7.10.3 Semantic router hardware and timing related assumptions

All semantic router related the time calculations assumes the high-level design of the semantic router as presented in Chapter VI, section 6.7.2. Within the SRN, a message needs *h* number of hops to reach the final destination. For each routing hop, the key needs to be transmitted over the network. Therefore, for each hop, the time needed to execute a message in the SRN has two major components: (1) t_{net}, the time that the message spends in the network that connects the semantic router nodes; and (2) $t_{\mathrm{msgprocess}}$, the time spent by the semantic router nodes to process the message. In section 6.7.2.20 of Chapter VI, we had shown that the message processing inside the semantic routers involves transferring the message data through the system interconnect and comparing the message key against routing table row keys and deciding the next destination address. The time response analysis of this system is briefly repeated here for sake of continuity.

For simplicity, we assume that the system bus interconnect is the bottleneck because each message may have large size ~100KB and for large messages, the message data transfer time to the memory is significantly larger than the message protocol processing time. Hence the transfer time is in the order of $t_{\text{I/O}}$, the time taken by the interconnect system. Similarly, we assume that the time to decide the route has the order of t_{comp}, the time taken to compare the message's key against all the keys in the semantic routing table and $t_{\text{min-dest-id}}$ is the time necessary to identify the minimum value and corresponding addresses from the destination column of the routing table. Therefore we can assert the followings:

$$t_{\text{msgprocess}} \sim (t_{\text{I/O}} + t_{\text{comp}} + t_{\text{min-dest-id}}) \tag{7.14}$$

$$t_{\text{SRN-exe}} = h \cdot (t_{\text{net}} + t_{\text{msgprocess}}) \approx h \cdot (t_{\text{net}} + t_{\text{I/O}} + t_{\text{comp}}) \tag{7.15}$$

7.10.4 Response time analysis of SRN having large router

7.10.4.1 Analysis for semantic routers using hardware comparator

In this case, a big semantic router having 1000 routing table rows is sufficient to send the message to one of the 1000 destination index nodes using only one routing hop, hence number of semantic routing hops $h = 1$. If we assume that the query key has a size of 100Kbytes (as explained in section 7.6.3), and the network protocol overhead is around 20%, then t_{net}, the time spend in the network is in order of 100KB/(0.8*10Gbps) = 100 microseconds for one networking hop.

Whereas message transfer time $t_{\mathrm{I/O}}$ will be for a two way transfer – the first one from the network interface sub-system to the buffer RAM during the message receiving and the second one from buffer RAM to network interface card during message transmission (refer section 6.7.2, Chapter VI. This involves 2 memory access operations over the system bus and memory interconnect (which is assumed to have a speed of 25.6GByte/sec [152]). Considering a 20% protocol overhead, the message transfer time $t_{\mathrm{I/O}}$ for a 100KByte sized key would take $100*10^3*(1/0.8)*2*1/(25.6*10^9) \approx 9.76$ microseconds. The time $t_{\text{min-dest-id}}$ is calculated using equation (6.2) presented in section 6.7.2.2 of Chapter VI. For our design we assume each modules packs in $q = 64$ comparators. Therefore to achieve a router with $n = 1000$ routing table rows, we need r to be = 16. This gives the $t_{\text{min-dest-id}} = \lceil \log_2 q \rceil + \lceil \log_2 r \rceil = \lceil \log_2 32 \rceil + \lceil \log_2 32 \rceil =$ 5+5 =10 clock cycles = 2.5 nanoseconds.

For semantic descriptors having say 1000 basis vectors, the expected value of the number of common basis vectors, c, is small in order of $1/(N_{\mathrm{P}}) = 10^{-3}$ (as explained earlier in Chapter II, section 4.5.2.3). In such case, the expected execution time is around 42.39 clock cycles (refer Table 7.3), which is less than 10.6 nanosecond for 4Ghz clock. The setup time for the comparator is ignored because it only one time setup cost and thus does not occur for every message processing. For the routing table lookup which involves 1000 key comparisons, the time to compare all 1000 pair of keys concurrently will be in order of 10.6 nanosec itself. This is because we choose to concurrently compare against all semantic routing table rows keys, as explained in section 6.7.2, Chapter VI.

Thus the estimated execution time $t_{\text{SRN-exe}}$ is ~ $(100 + 9.76 + 10.6*10^{-3} + 2.5*10^{-3}) = 109.77$ microseconds. This is also the average service period of each processor in the M/M/n queue model as explained Chapter II, section 2.11. We assume an indeterministic execution time due to following reason. In section 7.6.1.3 we have seen that the execution time of the comparison task varies with the number of common basis vector. The number of common basis vector is a random variable, hence the execution time is also a random variable and it is assume to obey exponential distribution.

We know from [1], that the long term query arrival rate $\lambda = 3500$ per second. So the utilization figure $\lambda * t = 3500 * 109.77 * 10^{-6} = 0.384$. From Table 2.1 of Chapter II, section 2.11, we can see that the average waiting time is 0.6666 times the execution time for an utilization figure of 0.40 (which is the nearest to 0.384), if we use only one semantic router hardware replica (i.e. number of servers/processor = 1).

This means that when one single router replica is used then average $t_{\text{SRN-resp}} = t_{\text{SRN-wait}} + t_{\text{SRN-exe}} = (1+0.6666) * t_{\text{SRN-exe}} = 1.6666 * 109.77 * 10^{-6} = 182.94$ micro seconds and there is 60% chance that the message will not have to wait. If two semantic router replicas are used to share the traffic load, then the $t_{\text{SRN-resp}}$ would be $= (1+0.1905) * 109.77 * 10^{-6} = 130.68$ micro seconds and there would be 77.14% chance that the message would not have to wait. This load sharing by replicas is explained in Fig. 7.21. Here the traffic distributor will send an incoming message to a randomly selected router replica, so that each replica encounters only a fraction of the message arrival rate.

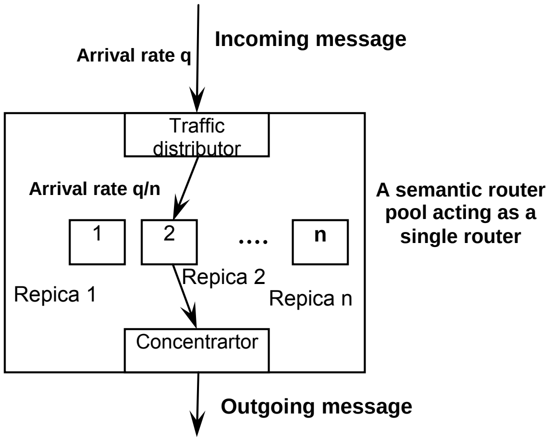


Fig. 7.21 Semantic Router replication technique for load sharing

7.10.4.2 For semantic routers using software comparator

On the other hand, the software implementation of the semantic key comparator takes $1000*7.7*10^6$ clock cycles (refer Table 7.3) which is ~1.925 seconds (for 4Ghz clock), to compare 1000 key comparisons. In this case the t_{net} and $t_{\text{I/O}}$ components remain same as the hardware comparator implementation case. Thus the execution time $t_{\text{SRN-exe}}$ is ~ (100 microsec + 9.76 microsec + 1.925 sec) ≈ 1.925 seconds. The $t_{\text{min-dest-id}}$ time is built in the 1.925 seconds time, because comparison simply requires maintaining the latest minimum value in a register which does not need a significant time.

The utilization figure $\lambda * t = 3500 * 1.925 = 6737.5$ which is >> 1. This means that to maintain the required throughput more than 6738 semantic router replicas are necessary to share the load as shown in Fig. 7.21. This need for large number of replicas is not attractive arrangement. Even with such arrangement, the execution time will be always be at least 1.925 seconds irrespective to the number of semantic routers deployed. Thus this scheme is not acceptable.

7.10.5 Response time analysis of SRN with small routers

7.10.5.1 For semantic routers using hardware comparator

If the SRN with small routers is used, as shown by the SRN simulation (Fig. 7.14 and Fig. 7.15), then on average, there would be around 2 routing hops (we take the nearest higher integer $h = 2$ instead of the value of 1.89) . Each router will have to carry out only 5 key comparisons in this case. In this SRN design, there will be around 5 copies of each query, because the message overhead is around 5 per query (refer Fig. 7.16). We take the nearest integer value of 5 instead of 4.8. This means each of the 200 routers will encounter 3500*5/200 = 87.5 queries per second. Out of these only $(5/200)^{th}$ fraction of the queries will have a c value which is 100%. On average, one out of 5 rows in the semantic routing table will have a match, therefore average c value will be 5/(200*5) = 0.5%. The comparator execution time t_{comp} for this $c = 0.05\%$, is around 26.32 cycles (= 6.58 nanoseconds). The $t_{\text{min-dest-id}}$ is still 10 clock cycles = 2.4 nanoseconds. The t_{net} is 100 microseconds and the $t_{\text{I/O}}$ is 9.76 microseconds as the earlier case. So the total execution time $t_{\text{SRN-exe}}$ for two semantic routing hop = 2* (6.58 nanosec + 2.5 nanosec + 100 microsec + 9.76 microsec) = 219.54 micro seconds.

The utilization figure $\lambda * t = 87.5 * 219.54 * 10^{-6} = 0.019$. The minimum utilization that is tabulated in Table 2.1 of Chapter II, section 2.11, is only 0.1. If we take that value, then we get an average waiting time as 0.1111 times of the execution time, if we do not use any additional replicas for load sharing (i.e. number of processor = 1). This indicates that the average $t_{\text{SRN-resp}}$ is less than $(1+ 0.1111) * 219.54 * 10^{-6} = 243.93$ micro seconds and there would be more than 90% chance that the message would not have to wait.

7.10.5.2 For semantic routers using software comparator

Whereas the software based comparator will require (2*5*7.7*10^{6}*0.25*10^{-9}=19.25 milliseconds for key comparisons + 2*109.76 microseconds for network and I/O transfer) = 19.47 millisecond. In this case the $\lambda*t$ = 87.5 * 19.47 *10^{-3} = 1.703. This means that at least 2 semantic router replicas are necessary to maintain the necessary throughput. As waiting time table does not provide estimates for utilization greater than 0.95, so we use an approximation method to get a rough estimate of the average execution time. Suppose we consider using 8 replicas of semantic routers for load sharing, then the average utilization is (87.5/8)*19.47 *10^{-3}) = 0.213. Waiting time is available from Table 2.1 for utilization value of 0.3 which is greater than 0.213. We take that conservative value for our purpose. The estimated waiting time for utilization of 0.3 and 1 processor case is 0.4286 times of the execution time. Thus the estimated average $t_{\text{SRN-resp}}$ = (1+0.4286) * 19.47 = 27.82 milli seconds, and there is 70% chance that a message would not have to wait.

7.10.6 SRN does not significantly perturb search engine performance

The entire search is completed in a typical search engines in order of 200 milli seconds [5]. Therefore the query delivery network (Semantic Routed Network) in the proposed model should also have response time ($t_{\text{SRN-resp}}$) which should be significantly small compared to 200 millisec. In the Table 7.7 below, we summarize the response time estimate of the proposed search engine which uses SRN. These estimates have been carried out in the earlier sections.

Here we examine how these response times of the proposed search engine compare with this 200 millisec time response of existing search engines. From the this table it is clear that both SRN designs which implements the semantic routers with hardware based descriptor comparators, is acceptable because insertion of SRN does not significantly increase the search engine response time from 200 millisec. This is because SRN's and step III's combined response time is in order of hundred micro seconds which is an insignificant value compared to 200 millisec.

Table 7.7 Time response of the proposed search engine design using SRN

SRN implemented with-	**Descriptor comparator Implementation**	**Step III & SRN response =** $t_{\textbf{StepIII-resp}}$ **+** $t_{\textbf{SRN-resp}}$	**# of load sharing replicas needed for each semantic router**	**Utilization factor = arrival rate/ service rate**	**Response time of the proposed search engine with SRN**	**Comment**
A single large single semantic router	Hardware	182.94 microsec + 10 nanosec	1	0.384	(200 millisec + 182.95 microsec)	Acceptable but not scalable
	Software	> 1.925 sec	> 6738	< 1	> (1.925 sec + 200 millisec)	Not acceptable
200 small semantic routers	Hardware	< 243.93 microsec + 10 nanosec	1	0.019	(200 millisec +243.94 microsec)	Acceptable & scalable
	Software	27.82 millisec + 10 nanosec	8	0.213	(200 + 27.82) millisec	May be acceptable

From a more practical engineering viewpoint we will interpret this as follows. We consider the hardware based single semantic router SRN case (row 1 in Table 7.7). There is a quite a difference between the required response time 200 millisec and the 182.94 microsec response time of the available comparator circuit. This significant difference leaves enough room to absorb the time overheads that will get added on top of this basic 182.94 microsec. These overheads will be incurred when the proposed comparator circuit is implemented as a module and many of such modules are integrated as a deployable product. It may appear that the parallel design as presented in section 6.7.2, Chapter VI, is an over-engineered design, but it may not be so under certain circumstances. That kind of parallelism may be still necessary if the overheads are large and we need to cut down the core execution time to leave enough room to accommodate the overheads. We believe that the best design will be a hybrid scheme of a mix of sequential and parallel designs to get an optimum balance between lower response time and lower hardware complexity.

These SRN implementations satisfy the throughput requirements when the required number of router replicas is deployed (as indicated in the 4th column of the above table). In all cases the utilization is less than 1 indicating the SRN has sufficient throughput (or service) capacity which is greater than the message arrival rate. The SRN design with small routers and software based comparator may be also acceptable. However, the most preferable option is the SRN with small routers using hardware comparator, because that design has much more capacity than what is needed at this moment. This additional capacity may be useful when the infrastructure is scaled up by adding many more index pools, in addition to the 1000 index nodes, that are currently present.

7.11 Estimated power consumption of the proposed semantic router

Here we show that the semantic table lookup mechanism, as proposed in section 6.7.2, Chapter VI, is feasible from point of view of power consumption and packing density. Based on our past hardware design experiences [131] and new hardware design related work, we have estimated that the power consumption of a descriptor similarity comparator having 10,000 slices, 32 CAM units and 16 multipliers will be within 10 Watts. The power consumption estimation of the comparator is beyond the scope of this thesis, however we will use this figure of 10W to show that it is possible to pack 32 such comparators in a comparator module, and 32 such modules in a 19 inch rack system as used in data centers.

Each comparator module packing 32 comparators will consume at least 10*32 = 320 Watts, which is in addition to the power consumption of around 200Watts for rest of the server motherboard. This server need not have a powerful host processor so an estimate of 200 Watt consumption is reasonable. This total of 200+320= 520 Watt power requirement is well within the capacity of a blade server form factor. A 19-inch data center rack can physically accommodate 32 comparator modules having either blade server or 1U form factors. The total power consumption of 32 comparator modules is 32 * 520 = 16.64 Kilo watts, which is well within the 20 Kilo Watt power density limit for a 19-inch rack.

7.12 Suitability of the proposed comparison technique

In the previous sections, we have substantiated the followings:

1) The proposed tensor model incorporates generative meaning composition therefore performs better than vector models in representing and comparing meanings.

2) The tensor model can represent many aspects of meaning comparison and interpretation that takes place within human mind and thus can be used to solve many problems in information retrieval.

3) Some of the key assumptions behind the tensor model is congruent with the theories in cognition science, thus the tensor model is realistic.

4) The proposed dot product computation approach yield correct results.

5) The setup and execution time of the proposed comparator is significantly smaller compared to those of the efficient software implementation and existing hardware designs.

6) The proposed approach is suitable in terms of memory requirement and satisfaction of the timing constraints for use in semantic routers.

All these taken together indicate that the proposed tensor model is physiologically more realistic compared to vector models and can be used to design a semantic router.

7.13 Suitability of the index self-organization technique

In the past sections we also substantiated the followings:

1) The proposed semantic routing table compaction algorithm enables small semantic routers to carry out the routing in a better manner compared to the large routers.

2) The proposed self-organizing scheme enables automatic generation of a small world semantic routed network topology.

Thus all these aforementioned arguments together indicate that the proposed network organization techniques can enable self-organization of a meaning based index.

7.14 Summary

Systematic, meaning based distribution of index entries in a distributed index system, leads to a resource efficient index structure. To create and operate this index structure a meaning based message routing network is needed. By using the proposed protocols, algorithms and techniques as presented in this chapter, it is possible to design a semantic routed network that route messages based on their meanings. Furthermore, the proposed network self-organization technique will allow automatic creation of this optimum distributed index system using the semantic routed network. In addition, by adopting the proposed semantic routing table compaction algorithm it is possible to built this semantic routed network by small routers instead of larger ones without trading off routing (and index) performances. To create and operate the aforementioned semantic routed network and semantic routers, a method is needed to represent and compare meanings. The proposed tensor model can be a good candidate for this purpose. The proposed information processing architecture enables high speed computation of the tensor based meaning comparison and needs a memory space which can be implemented using existing devices. Thus this architecture is suitable to be used in the semantic routers which can then be used to materialize the semantic routed network and the distributed index system.

CHAPTER VIII

CONCLUSION

8.1 Open research problems

This dissertation covered a variety of problems and delved deeper to explore several of them. However, there is couple of unexplored problems which should be investigated further. The significant ones are mentioned below:

1. Design and evaluation of alternative techniques to generate concept tree from a given text in addition to the method suggested in Chapter III. This will involve application of natural language processing and machine learning techniques and human user based experimentations.

2. Evaluating the need to integrate existing methods like, term vector models, latent semantic indexing, etc., along with the proposed tensor model to improve performance of meaning comparison, and design of a method to materialize their integration. This will need user based experiments on a variety of text document corpus. This can be carried out only after the previous task has been completed.

3. Design of hash functions that can be parallelized at circuit level to implement extremely fast and efficient hash function circuitry that is needed in the proposed meaning comparator architecture.

4. Design of a pipelined scheme for the proposed comparator architecture.

5. Power consumption and silicon area estimates of the alternative architectures as proposed in Chapter IV, table 4.1.

6. Finally, a more detailed design of the semantic router using the aforementioned artifacts. Here, a challenge would be to maintain high I/O throughputs between network interface to memory and memory to the comparator hardware. This can be best done if all these three – network interface, memory and comparator are closely integrated.
7. Apply the index distribution technique in an actual search system and evaluate the improvements in performance.

8.2 Contributions

There is a significant demand for meaning based search systems. The key challenges in building such systems are scalability and ability to provide relevant search results. Existing schemes for distributing the search indexes are not resource efficient. Therefore an alternative efficient index distribution and organization scheme is needed. This scheme requires a meaning based message delivery network and an appliance called semantic router to materialize this network. Constructing this semantic router will require a superior method to represent and compare meanings of text documents and objects, compared to existing vector meaning representation models. In addition, novel networking organization and operation mechanisms (algorithms and protocols) are also necessary.

This dissertation presented the following foundational technologies necessary to enable superior meaning representation, comparison and meaning based message delivery network:

1. An algebraic theory, design of a data structure and related algorithms for representing composite meaning in a psychologically realistic manner. This technique will enable more precise meaning based searching.

2. An efficient technique to compare two data structures for similarity of their meanings. This will enable implementation of fast response semantic routers, Semantic Routed Network and index server systems that enable more precise meaning based searching.
3. Design of an overlay networking scheme that deliver messages based on their meanings. This enables implementation of a meaning based distributed index.
4. Techniques to construct a scalable and self-organizing meaning based index.

We showed that the proposed meaning comparison techniques can compare composite meanings ~10^5 times faster than an equivalent software code and existing hardware designs. Whereas, the proposed index organization approach can lead to 33% savings in number of servers and power consumption in a model search engine having 700,000 servers. Therefore, using all these techniques, it is possible to design a Semantic Routed Network which has a potential to improve relevance of search results and search response time, while saving resources.

REFERENCES

[1] A. Patriquin, “March Search Market Share: Record query growth and the Yahoo/Microsoft search deal by the numbers,” Available: http://blog.compete.com/2009/04/13/search-market-share-march-google-yahoo-msn-live-ask-aol-2/, Apr. 2009.

[2] J. C. Perez, “Google joins crowd, adds semantic search capabilities,” *Computer World*, Mar. 2009.

[3] P. Lenssen, “Checking Google’s Index Size,” *Blogoscoped*, 2006. Available: http://blogoscoped.com/archive/2006-12-27-n44.html [Apr. 2009].

[4] L. A. Barroso, J. Dean, and U. lzle, “Web search for a planet: The google cluster architecture,” *IEEE Micro*, vol. 23, no. 2, 2003.

[5] A. Agarwal, “Single Google query uses 1000 machines in 0.2 seconds,” *Digital Inspiration*, Available: http://www.labnol.org/internet/search/google-query-uses-1000-machines/7433/, [Feb. 2009].

[6] D. MacKay, “Google Searches, energy cost, carbon footprint, and cups of tea,” Available: http://withouthotair.blogspot.com/2009/01/google-searches-energy-cost-carbon.html, Jan. 2009, [Apr. 2009].

[7] Wikipedia, Information retrieval, Available: http://en.wikipedia.org/w/index.php?title=Information_retrieval&oldid=274343961, [Apr., 2009].

[8] B. Rosario, “Latent Semantic Indexing: An overview,” School of Information Management & Systems, U.C. Berkeley, Available: http://people.ischool.berkeley.edu/~rosario/projects/LSI.pdf, Spring, 2000,

[9] G. Salton, and C. Buckley, “Term-weighting approaches in automatic text retrieval,” *Information Processing & Management*, vol. 24, no. 5, 513–523, 1988.

[10] B. Botelho, “Gartner predicts data center power and cooling crisis,” 2007, Available: http://searchdatacenter.techtarget.com/news/article/0,289142,sid80_gci1260874,00.html, [Mar. 2009].

[11] Wikipedia, “Bag of words model,” Available: http://en.wikipedia.org/wiki/Bag_of_words_model, [Mar. 2010].

[12] J. Mitchell, M. Lapata, “Vector-based models of semantic composition,” *Proc. 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, Columbus, OH, 2008.

[13] A. Biswas, S. Mohan, A. Tripathy, J. Panigrahy, R. Mahapatra, "Semantic key for meaning based searching," *Proc. 3rd IEEE International Conference on Semantic Computing*, Berkeley, CA, USA, 2009.

[14] Pubmed, http://www.ncbi.nlm.nih.gov/pubmed/ [Mar 2010]

[15] A. Biswas, S. Mohan, R. Mahapatra, "Semantic technologies for distributed search P2P networks," *Handbook of Research on P2P and Grid Systems for Service-Oriented Computing: Models, Methodologies and Applications*, N. Antonopoulos, G. Exarchakos, M. Li and A. Liotta Eds., IGI Publisher, Hershey, PA, USA, 2009.

[16] A. Biswas, S. Mohan, and R. Mahapatra, "Semantic technologies for searching in e-Science grids," *Semantic e-Science,* H. Chen, Y. Wang and K.H. Cheung Eds., Springer AoIS Book Series, New York, 2009.

[17] E. Saffran, "The organization of semantic memory: In support of a distributed model," *Brain and Language*, vol. 71, no. 1, pp. 204–212, 2000.

[18] C. Bai, I. Bornkessel-Schlesewsky, L. Wang, Y.-C. Hung, M. Schlesewsky and P. Burkhardt, "Semantic composition engenders an N400: evidence from Chinese compounds," *Neuro Report*, vol. 19, no. 6, pp. 695, 2008.

[19] M.M. Piango, "The neural basis of semantic compositionality," In session hosted by the Yale Interdepartmental Neuroscience Program, Yale University, 2006.

[20] G. Murphy, "Comprehending complex concepts," *Cognitive Science*, vol. 12, no. 4, pp. 529–562, 1998.

[21] P.W. Culicover and R. Jackendoff, "The simpler syntax hypothesis," *Trends in Cognitive Science*, vol. 10, no. 9, pp. 413-418, 2006.

[22] P.W. Culicover and R. Jackendoff, "Simpler syntax," *Oxford linguistics*, Oxford University Press, Oxford, UK, 2005.

[23] G. Kuperberg, "Neural mechanisms of language comprehension: Challenges to syntax," *Brain Research*, vol. 1146, pp. 23–49, 2007.

[24] H.S. Kirshner, "Language studies in the third millennium," *Brain and Language*, vol. 71, no. 1, pp. 124-128, 2000.

[25] P. Hagoort, "On Broca, brain, and binding: A new framework," *Trends in Cognitive Sciences*, vol. 9, no. 9, pp. 416–423, 2005.

[26] Y. Grodzinsky, "The neurology of syntax: Language use without Broca's area," *Behavioral and Brain Sciences*, vol. 23, no. 01, pp. 1-21, 2001.

[27] A. Caramazza and R.S. Berndt, "Semantic and syntactic processes in aphasia: A review of the literature," *Psychological Bulletin*, vol. 85, no.4, pp. 898-918, 1978.

[28] A.D. Friederici, B. Opitz and D.Y. Cramon, “Segregating semantic and syntactic aspects of processing in the human brain: An fMRI investigation of different word types,” *Cerebral Cortex*, vol. 10, pp. 698-705, 2000.

[29] J. Brennan, and L. Pylkknen, “Semantic composition and inchoative coercion: An MEG study,” *Proc. 21st Annual CUNY Conference on Human Sentence Processing*, University of North Carolina, Chapel Hill, 2008.

[30] Z. Ye and X. Zhou, “Involvement of cognitive control in sentence comprehension: Evidence from erps,” *Brain Research*, vol. 1203, pp. 103–115, 2008.

[31] M. Ekiert, “The bilingual brain,” *Working Papers in TESOL and Applied Linguistics*, vol.3, no.2, 2003.

[32] K.H.S. Kim, N.R. Relkin, and J. Hirsch, “Distinct cortical areas associated with native and second languages,” *Nature*, vol. 338, pp.171-174, 1997.

[33] E. Zurif, “Syntactic and semantic composition,” *Brain and Language*, vol. 71, no. 1, pp. 261–263, 2000.

[34] D. Bickerton, *Language & species*, The University of Chicago Press, Chicago & London, 1990.

[35] R. Jackendoff, “Compounding in the parallel architecture and conceptual semantics,” *The Oxford handbook of compounding*, Ed. R. Lieber, R. Stekauer, Oxford: Oxford University Press, 2009.

[36] A.M. Collins and E.F. Loftus, “A spreading-activation theory of semantic processing,” *Psychological Review*, vol. 82, no. 6, pp. 407-428, 1975.

[37] J.R. Anderson, “A spreading activation theory of memory,” *Journal of Verbal Learning and Verbal Behavior*, vol. 22, 261-295, 1983.

[38] J.R. Anderson and P.L. Pirolli P L, “Spread of activation,” *Journal of Experimental Psychology: Learning, Memory, & Cognition*, vol. 10, pp.791-799, 1984.

[39] R.A. Rodriguez, “Aspects of cognitive linguistics and neurolinguistics: conceptual structure and category-specific semantic deficits,” *Estudios Ingleses de la Universidad Complutense*, vol. 12, pp. 43–62, 2004.

[40] Wikipedia, Semiotics, http://en.wikipedia.org/wiki/Semiotics, Accessed on Mar 12, 2010.

[41] Wikipedia, Machine learning, Available: http://en.wikipedia.org/wiki/Machine_learning, [Mar. 2010].

[42] R. Rajapske and M. Denham, “Fast access to concepts in concept lattices via bidirectioanl associative memory,” *Neural Computation*, vol. 17, pp. 2291-2300, 2005.

[43] R. Rajapske and M. Denham, “Text retrieval with more realistic concept matching and reinforcement learning,” *Information Processing and Management*, vol.42, pp. 1260-1275, 2006.

[44] C. Andersen, “A computational model of complex concept composition,” M.S. thesis, Department of Computer Science, University of Texas, Austin,1996.

[45] Wikipedia, “Latent Dirichlet allocation,” Available: http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation, [Mar 2010]

[46] D. Widdows, “Semantic vector products: Some initial investigations,” *Quantum Interaction: Papers from the Second International Symposium*, Oxford, 2008.

[47] D. Widdows, “A mathematical model for context and word-meaning,” *Lecture Notes In Computer Science*, pp. 369-382, 2003.

[48] A. Augello, G. Vassallo, S. Gaglio and G. Pilato, “Sentence induced transformations in "conceptual" spaces,” Proc. *IEEE International Conference on Semantic Computing*, pp. 34-41, 2008.

[49] K. Coursey, R. Mihalcea and W. Moen, “Using encyclopedic knowledge for automatic topic identification,” *Proc. 13th Conference on Computational Natural Language Learning*, pp. 210–218, Boulder, Colorado, 2009.

[50] Coursey, K and Mihalcea, Rada, “Topic identification using graph centrality,” *Proc. Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 117-120, 2009.

[51] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *Proc. 6th Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004.

[52] D. Widdows, “Geometric ordering of concepts, logical disjunction, and learning by induction,” *Compositional Connectionism in Cognitive Science*, AAAI Fall Symposium Series, Washington, DC, 2004.

[53] D. Widdows, “Orthogonal negation in vector spaces for modeling word meanings and document retrieval,” *Proc. 41st Annual Meeting of the Association for Computational Linguistics*, 2003.

[54] G.L. Murphy and D.L. Medin, “The role of theories in conceptual coherence,” *Psychological Review*, 1985.

[55] H. Ogata, W. Fujibuchi, S. Goto and M. Kanehisa, “A heuristic graph comparison algorithm and its application to detect functionally related enzyme clusters,” *Nucleic Acids Research*, vol. 28, no. 20, pp. 4021-4028, 2000.

[56] P. Foggia, C. Sansone and M. Vento, “A performance comparison of five algorithms for graph isomorphism,” *Proc. 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pp. 188-199, 2001.

[57] K.E. Wolff, “A first course in formal concept analysis,” *Proc. StatSoft '93*, Ed. F. Faulbaum, pp. 429-438, Gustav Fischer Verlag, 2004.

[58] J. Qi, L. Wei and Y. Bai, “Composition of concept lattices,” *Proc. 7th International conference on machine learning and cybernatics*, Kunming, 2008.

[59] P. Gerrans and V. E. Stone, “Generous or parsimonious cognitive architecture? Cognitive neuroscience and theory of mind,” *British Journal of Philosophical Science*, vol. 59, pp. 121–141, 2008.

[60] Wikipedia, “Resource description framework,” Available: http://en.wikipedia.org/wiki/Resource_Description_Framework, [Apr. 2009].

[61] Wikipedia, “RDFa,” Available: http://en.wikipedia.org/wiki/RDFa, [Apr. 2009].

[62] W3C, “HTML 5: A vocabulary and associated APIs for HTML and XHTML,” *Editor's Draft 9*, http://dev.w3.org/html5/spec/Overview.html, [Aug. 2009].

[63] Wikipedia, “Microformat,” Available: http://en.wikipedia.org/wiki/Microformat, [Apr. 2009].

[64] Wikipedia, “HTML5,” Avaialable: http://en.wikipedia.org/wiki/HTML_5, [Apr. 2009].

[65] J. T. L. Wang, K. Zhang, K. Jeong and D. Shasha, “A system for approximate tree matching,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 4, pp.559 - 571, 1994.

[66] D. Chen, J. G. Burleigh, M. S. Bansal and D. Fernández-Baca, “PhyloFinder: An intelligent search engine for phylogenetic tree databases,” *BMC Evolutionary Biology*, vol. 8, no. 90, 2008.

[67] D. R. Robinson and L. R. Foulds, “Comparison of phylogenetic trees,” *Mathematical Biosciences*, vol. 53, pp. 131-147, 1981.

[68] V. Krishna, N. Ranganathan and A. Ejnioui, “A tree-matching Chip,” *IEEE Transactions On Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 2, 1999.

[69] L. Knecht, “PubMed: Truncation, automatic explosion, mapping, and MeSH headings,” *NLM Technical Bulletin*, pp.302, May-Jun. 1998.

[70] Medical subject headings, U.S. National Library of Medicine, Available: www.nlm.nih.gov/mesh/, [Apr 2009].

[71] E. Alvarez-Lacalle, B. Dorow, J.-P. Eckmann and E. Moses, “Hierarchical structures induce long-range dynamical correlations in written texts,” *Proc. National Academy of Sciences of the United States of America*, 2006.

[72] Wikipedia, “Named entity recognition,” Available: http://en.wikipedia.org/wiki/Named_entity_recognition, [Feb. 2010].

[73] M. Ruiz-Casado, E. Alfonseca, P. Castells, “Automatic assignment of Wikipedia encyclopedic antries to WordNet synsets,” *Advances in Web Intelligence*, pp. 380-386, 2005.

[74] Wikipedia, “Information extraction,” Available: http://en.wikipedia.org/wiki/Information _extraction, [Feb. 2010].

[75] D. Balasuriya, N. Ringland, J. Nothman, T. Murphy and J.R. Curran, “Named entity recognition in Wikipedia,” *Proc. Workshop on The People's Web Meets NLP: Collaboratively Constructed Semantic Resources*, Suntec, Singapore. 2009.

[76] H.W. Chun, Y. Tsuruoka, J.D. Kim, R. Shiba, N. Nagata, T. Hishiki, J. Tsujii, “Extraction of gene-disease relations from Medline using domain dictionaries and machine learning,” *Pacific Symposium on Biocomputing*, 2006.

[77] M. Huang, X. Zhu, Y. Hao, D. G. Payan, K. Qu and M. Li “Discovering patterns to extract protein-protein interactions,” *Bioinformatics*, vol. 20. pp. 3604–3612, 2004.

[78] T.C. Rindflesch, L.Tanabe, J.N.Weinstein and L.Hunter, “EDGAR: Extraction of drugs, genes, and relations from the biomedical literature,” *Proc. Pacific Symposium on Biocomputing*, pp. 514-525, 2000.

[79] C. Ramakrishnan, K. J. Kochut and A. P. Sheth “A Framework for schema-driven relationship discovery from unstructured text,” *Proc. International Semantic Web Conference*, pp. 583–596, 2006.

[80] W. Wong, W. Liu and M. Bennamoun, “Acquiring semantic relations using the web for constructing lightweight ontologies,” *Proc. 13th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2009.

[81] M. Balakrishna, and M. Srikanth, “Automatic ontology creation from text for National Intelligence Priorities Framework (NIPF),” *Proc. Conference on Ontology for the Intelligence Community*, 2008.

[82] Pedro, V., Niculescu, S. and Lita, L. Okinet: Automatic extraction of a medical ontology from Wikipedia. *WiKiAI08: A workshop of AAAI2008*, 2008.

[83] Roberson, S., Dicheva, D.: Semi-automatic ontology extraction to create draft topic maps. *Proc. 45th ACM Southeast Conference*, Winston-Salem, NC, March 2007, pp. 23–24, 2007.

[84] D. Mladenić, M. Grobelnik, “Evaluation of semi-automatic ontology generation in real-world setting,” *Proc. 29th International Conference on Information Technology Interfaces*, 2007.

[85] A. Maedche, E. Maedche and S. Staab, “Learning ontologies for the semantic web,” *Proc. Semantic Web Worskhop*, 2001.

[86] M. A. Hearst, “Automatic acquisition of hyponyms from large text corpora,” *Proc. 14th International Conference on Computational Linguistics*, pp. 539-545, Nantes, France, July 1992.

[87] P. Buitelaar and P. Cimiano, “Ontology learning and population: Bridging the gap between text and knowledge,” *Series information for Frontiers in Artificial Intelligence and Applications*, IOS Press, 2008.

[88] P. Buitelaar, P. Cimiano, and B. Magnini, “Ontology learning from text: Methods, evaluation and applications,” *Series information for Frontiers in Artificial Intelligence and Applications*, IOS Press, 2005.

[89] R. Navigli and P. Velardi, “Learning domain ontologies from document warehouses and dedicated web sites,” *Computational Linguistics*, vol. 30, no. 2, MIT Press, pp. 151-179, 2004.

[90] Wikipedia, “Automatic summarization,” http://en.wikipedia.org/wiki/Automatic_summarization, [Mar. 2010].

[91] D. Marcu, “From discourse structures to text summaries,” *Proc. of ACL’97/EACL’97 Workshop on Intelligent Scalable Text Summarization*, pp. 82–88, Madrid, Spain, July 1997.

[92] D. Marcu, “The rhetorical parsing, summarization, and generation of natural language texts,” PhD thesis, Department of Computer Science, University of Toronto, Dec. 1997.

[93] K. Ono, K. Sumita and S. Miike, “Abstract generation based on rhetorical structure extraction,” *Proc. International Conference on Computational Linguistics*, vol. 1, pp. 344–348, Kyoto, Japan, 1994.

[94] K. S. Jones, “What might be in a summary?”, *Proc. Information Retrieval ’93*, Eds. G. Knorz, J. Krause and C. Womser-Hacker, Von der Modellierung zur Anwendung, pp. 9–26, Universit¨atsverlag Konstanz, Konstanz,1993.

[95] UNL Wiki, “Tools”, Available: http://www.undl.org.br/wiki/index.php?title=Tools, [Mar. 2010].

[96] Wikipedia, “Universal Networking Language,” Avaliable: http://en.wikipedia.org/wiki/Universal_Networking_Language, [Mar. 2010].

[97] U. Priss, “Lattice-based Information Retrieval,” Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.2145, 2000.

[98] S. Joseph, “Neurogrid: Semantically routing queries in peer-to-peer networks,” *Proc. International. Workshop on Peer-to-Peer Computing*, pp. 202-214, 2002.

[99] F. M. Cuenca-Acuna, C. Peery, R. P. Martin and T. D. Nguyen, “Planetp: Using gossiping to build content addressable peer-to-peer information sharing communities,” Proc. *12th IEEE International Symposium on High Performance Distributed Computing*, 2003.

[100] E. Michlmayr, S. Graf, W. Siberski, and W. Nejdl, “Query routing with ants,” *Proc. 1st Workshop on Ontologies in P2P Communities* in *2nd Annual European Semantic Web Conference* Greece, 2005.

[101] I. Brunkhorst, H. Dhraief, A. Kemper, W. Nejdl, and C. Wiesner, “Distributed queries and query optimization in schema-based p2p-systems,” *Proc. Databases, Information Systems and. Peer-to-Peer Computing*, pp. 184-199, 2003.

[102] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, “Content-based addressing and routing: A general model and its application,” Technical Report CU-CS-902-00, University of Colorado, Computer Science department, Jan. 2000.

[103] D. Faye, G. Nachouki, and P. Valduriez, “Semantic query Routing in SenPeer, a P2P data management system,” *Lecture Notes in Computer Science*, vol. 4658, pp. 365-374, 2007.

[104] S. Montanelli and S. Castano, “Semantically routing queries in peer-based systems: The h-link approach,” *Knowledge Engineering Review*, vol. 23, no. 1, pp. 51-72, 2008.

[105] P. Fraigniaud and P. Gauron, “D2b: A de bruijn based content-addressable network,” *Theoritical Computer Science*, vol.355, no.1, pp. 65-79, 2006.

[106] N. T. Borch, “Improving semantic routing efficiency,” *Proc. 2nd International Workshop on Hot Topics in Peer-to-Peer Systems*, pp. 80-86, 2005.

[107] N. T. Borch and L. K. Vognild, “Searching in variably connected P2P networks,” *Proc. International Conference on Pervasive computing and communications*, 2004.

[108] W. Acosta, and S. Chandra, “Improving search using a fault-tolerant overlay in unstructured P2P systems,” *Proc.International Conference on Parallel Processing*, Sept. 2007.

[109] “Gnutella protocol specification,” Available: http://wiki.limewire.org/index.php?title=GDF, [Apr 2009].

[110] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker, “A scalable content-addressable network,” *Proc. Annual conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, SanDiego, California.,2001.

[111] D. Zeinalipour-Yazti, V. Kalogeraki and D. Gunopulos, “Information Retrieval in Peer-to-Peer Networks,” Available: http://www.cs.ucr.edu/~csyiazti/papers/cise2003/cise2003.pdf, 2003.

[112] J. Li, S. Vuong, “A scalable semantic routing architecture for grid resource discovery, parallel and distributed Systems,” *Proc. 11th International Conference on*, pp. 29- 35, vol. 1, pp. 20-22, 2005.

[113] C. Tempich, S. Staab and A. Wranik, “REMINDIN': Semantic query routing in peer-to-peer networks based on social metaphors,” *Proc. 13th International World Wide Web Conference*, pp. 640-649, 2004.

[114] A. Kothari, D. Agrawal, A. Gupta and S. Suri, “Range addressable network: A P2P cache architecture for data ranges,” *Proc. 3rd Iinternational Conference on Peer-To-Peer Computing*, Washington, DC, pp.14, 2003.

[115] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan and J. Lilley, “The design and implementation of an intentional naming system,” *Proc. 17th ACM symposium on Operating systems principles*, vol. 33, pp. 186-201, New York, NY, 1999.

[116] S. Waterhouse, M.D. Doolin, G. Kan, Y. Faybishenko, “Distributed search in P2P networks,” *IEEE Internet Computing,* vol. 6, no. 1, pp. 68-72, 2002.

[117] R. Albert, and A.L. Barabasi, “Statistical mechanics of complex networks,” *Reviews of Modern Physics*, vol. 74, pp. 47-97, 2002.

[118] D. J. Watts, “Six Degrees: The Science Of A Connected Age,” W.W. Norton & Company, 2003.

[119] D. J. Watts and S.H. Strogatz, “Collective dynamics of ‘small-world’ networks,” *Nature*, vol. 393, pp. 440-442, 1998.

[120] Wikipedia, “Clustering coefficient,” Available: http://en.wikipedia.org/wiki/Clustering_coefficient, [Mar. 2010].

[121] J. Klienberg, “The small-world phenomena: an algorithmic perspective,” *Proc. 32nd ACM Symposium on Theory of Computing*, pp. 163-170, 2000.

[122] P. Fraigniaud, C. Gavoille and C. Paul, “Eclectism shrink even small worlds,” *Distributed Computing*, vol. 18, no. 4, pp. 279-291, 2006.

[123] Wikipedia, “Geographic routing,” Available: http://en.wikipedia.org/wiki/Geographic_routing, [Mar. 2010].

[124] Wikipedia, “Taxicab geometry,” Available: http://en.wikipedia.org/wiki/Taxicab_geometry, [Mar. 2010].

[125] Wikipedia, “Poisson distribution,” Available: http://en.wikipedia.org/wiki/Poisson_distribution, [Mar. 2010].

[126] Wikipedia, “Exponential distribution,” Available: http://en.wikipedia.org/wiki/Exponential_distribution, [Mar 2010].

[127] Wikipedia, “Kendalls notation,” Available: http://en.wikipedia.org/wiki/Kendall%27s_notation, [Mar 2010].

[128] E. Page, “Tables of waiting times for M/M/n, M/D/n and D/M/n and their use to give approximate waiting times in more general queues,”*Journal of Operations Research Society*, vol. 33, pp. 453-473, 1982.

[129] A. Broder, M. Mitzenmacher, “Network applications of Bloom Filters: A survey,” *Internet Mathematics*, vol.1, no.4, pp.485-509, 2002.

[130] A.F. Webster and S.E. Taveres, “On the design of S-boxes,” *Proc. Annual International Cryptology* Conference, pp. 523-534, 1985.

[131] S. Mohan, A. Biswas, A. Tripathy and R. Mahapatra, “A parallel architecture for meaning comparison,” *Proc. 24th IEEE International Parallel and Distributed Processing Symposium*, 2010.

[132] Amazon, www.amazon.com, [Mar 2010].

[133] Gene Ontology, http://www.geneontology.org/, [Apr 2009].

[134] Disease Ontology, http://diseaseontology.sourceforge.net/, [Apr 2009].

[135] N. Bottini et al. “A functional variant of lymphoid tyrosine phosphatase is associated with type I diabetes,” *Nature Genetics*, vol. 36, pp.337– 338, 2004.

[136] Wikipedia, “Dewey decimal classification,” http://en.wikipedia.org/wiki/Dewey_Decimal_Classification, [Mar. 2010].

[137] F. Irgens, “Tensors,” *Continuum Mechanics*, Springer, New York, 2004.

[138] Wikipedia, “Fowler-Noll-Vo hash function,” Available: http://en.wikipedia.org/wiki/Fowler-Noll-Vo_hash_function, [Mar. 2010].

[139] A. Biswas, S. Mohan, A. Tripathy, J. Panigrahy, R. Mahapatra, “Representation and comparison of complex concepts for Semantic Routed Network,” *Proc. 10th International Conference on Distributed Computing and Networking*, Hyderabad, 2009.

[140] T. Kumaki, K. Iwai and T. Kurokawa, “A flexible multiport content- addressable memory,” *Systems and Computers in Japan*, vol. 37, no 11, 2006.

[141] H.J. Mattausch, et al. “Area-efficient multi-port SRAMs for on-chip data-storage with high random-access bandwidth and large storage capacity,” *IEICE Transaction on Electronics*, vol. E84-C, no. 3, pp. 410-417, 2001.

[142] A. Kirsch and M. Mitzenmacher, “Less hashing, same performance: Building a better Bloom Filter,” *Proc. 14th Annual European Symposium on Algorithms*, Zurich, 2006.

[143] Wikipedia, “Uniform resource identifier,” Available: http://en.wikipedia.org/wiki/Uniform_Resource_Identifier, [Mar. 2010].

[144] Wikipedia, “Heap (data structure),” Available: http://en.wikipedia.org/wiki/Heap_(data_structure), [Mar. 2010].

[145] A. Biswas, S. Mohan, R. Mahapatra, “Search co-ordination by Semantic Routed Network,” *Proc. 18th International Conference on Computer Communication and Network*, San Francisco, CA, Aug 2009.

[146] A. Biswas, S. Mohan, R. Mahapatra, “Optimization of semantic routing table,” *Proc. 17th International Conference on Computer Communication and Network*, US Virgin Islands, 2008.

[147] S. Deerwester, T. Dumais, R. Harshman, “Indexing by latent semantic analysis,” *Journal of the American Society of Information Science*, vol. 41, pp. 391-407, 1990.

[148] P. Jaccard, “Étude comparative de la distribution florale dans une portion des Alpes et des Jura,” *Bulletin del la Société Vaudoise des Sciences Naturelles*, vol. 37, pp. 547-579.

[149] D. G. Perera and K. F. Li, “On-chip hardware support for similarity measures,” *Proc. IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pp. 354-358, 2007.

[150] M. Freeman, M. Weeks and J. Austin, “Hardware implementation of similarity functions,” *Proc. IADIS International Conference on Applied Computing*, pp. 329-332, 2005.

[151] D. G. Perera and K. F. Li, “Parallel computation of similarity measures using an fpga-based processor array,” *Proc. IEEE 22nd International Conference on Advanced Information Networking and Applications,* pp. 955-962, 2008.

[152] Wikipedia, “Intel quickpath interconnect,” Available: http://en.wikipedia.org/wiki/Intel_Quickpath_Interconnect, [Mar 2010].

VITA

Name: Amitava Biswas

Address: HRBB 301, TAMU 3112, Texas A&M University,
College Station, TX 77843-3112

Email Address: amitavabiswas@ieee.org

Education: B.Tech.(Hons.), Indian Institute of Technology, Kharagpur, 1993.
M.B.A., Indian Institute of Management, Ahmedabad, 1997.
M.S., Concordia University, Montreal, 2005.
Ph.D., Texas A&M University, 2010.