Aston University

Some pages of this thesis may have been removed for copyright restrictions.

If you have discovered material in Aston Research Explorer which is unlawful e.g. breaches copyright, (either yours or that of a third party) or any other law, including but not limited to those relating to patent, trademark, confidentiality, data protection, obscenity, defamation, libel, then please read our Takedown policy and contact the service immediately (openaccess@aston.ac.uk)

Self-Adapting Parallel Metric-Space Search Engine for Variable Query Loads

Khalil Badar Ali AL Ruqeishi

Doctor Of Philosophy

Aston University

Jun 2015

©Khalil Badar Ali AL Ruqeishi, 2015

Khalil Badar Ali AL Ruqeishi asserts his moral right to be identified as the author of this thesis

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without appropriate permission or acknowledgement.

Aston University

Self-Adapting Parallel Metric-Space Search Engine for Variable Query Loads

Khalil Badar Ali AL Ruqeishi
Doctor Of Philosophy, 2015

Thesis Summary

This research focuses on automatically adapting a search engine size in response to fluctuations in query workload. Deploying a search engine in an Infrastructure as a Service (IaaS) cloud facilitates allocating or deallocating computer resources to or from the engine. Our solution is to contribute an adaptive search engine that will repeatedly re-evaluate its load and, when appropriate, switch over to a different number of active processors.

We focus on three aspects and break them out into three sub-problems as follows: Continually determining the Number of Processors (CNP), New Grouping Problem (NGP) and Regrouping Order Problem (ROP). CNP means that (in the light of the changes in the query workload in the search engine) there is a problem of determining the ideal number of processors p active at any given time to use in the search engine and we call this problem CNP. NGP happens when changes in the number of processors are determined and it must also be determined which groups of search data will be distributed across the processors. ROP is how to redistribute this data onto processors while keeping the engine responsive and while also minimising the switchover time and the incurred network load.

We propose solutions for these sub-problems. For NGP we propose an algorithm for incrementally adjusting the index to fit the varying number of virtual machines. For ROP we present an efficient method for redistributing data among processors while keeping the search engine responsive. Regarding the solution for CNP, we propose an algorithm determining the new size of the search engine by re-evaluating its load. We tested the solution performance using a custom-build prototype search engine deployed in the Amazon EC2 cloud. Our experiments show that when we compare our NGP solution with computing the index from scratch, the incremental algorithm speeds up the index computation 2–10 times while maintaining a similar search performance. The chosen redistribution method is 25% to 50% faster than other methods and reduces the network load around by 30%. For CNP we present a deterministic algorithm that shows a good ability to determine a new size of search engine. When combined, these algorithms give an adapting algorithm that is able to adjust the search engine size with a variable workload.

Keywords: Metric-Space Search engine, query workload, computer resources

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor Dr Michal Konečný for his invaluable support and guidance throughout this process. I would not have been able to complete the work of this Thesis if not for him. I am also incredibly thankful for his time and patience in correcting my thesis and paper, and for his enlightening discussions that brought me closer to the solution. His time and effort, which was invested in me by him, indeed really helped me on my path towards becoming a researcher. I would also like to express my gratitude to Dr Radu Calinescu from York University for his guidance and support during my PhD. His advice has been an important influence on my work. Similarly, I would like to thank my colleagues and friends in the computer science department for their friendship and for providing unforgettable moments. I would like to thank Dr Veronica Gil-Costa from Yahoo! Research Latin America for her help to validate my work. I would like to thank Mr Alexander Brulo for his help and support during my study. I would like to thank my colleague Dr Shahzad Mumtaz for his help and support during my PhD. In particular, I would like to thank my parents and friends for their unconditional love and support. Lastly, but not least, I want to thank my wife and my children for their support and patience all of these years.

Contents

1 Introduction **1**

1.1 Research Questions . . . 3
1.2 Contribution . . . 3
1.3 Structure of the Thesis . . . 4
1.4 Publications and Software . . . 4

2 Background **5**

2.1 Search Engine Distributed Architecture . . . 5
2.2 Similarity Search . . . 6
2.2.1 Metric Spaces . . . 8
2.2.2 Similarity Queries . . . 8
2.3 Metric Space Indexing Approaches . . . 8
2.3.1 Sparse Spatial Selection (SSS) Approach . . . 9
2.3.2 List of Clusters (LC) Approach . . . 9
2.4 Parallel Query Processing in Distributed Metric Space . . . 13
2.5 Distributed Metric Space Index Problem (DMP) . . . 15
2.5.1 Distributed Metric Space Index for Search Engine . . . 17
K-means Clustering . . . 18
Query Vectors . . . 19
Km-Col Algorithm . . . 19
2.6 Search Engine and Cloud Architecture . . . 20
2.6.1 A Search Engine in Cloud . . . 21
2.7 Feedback Controller of Search Engine Simulator . . . 23
2.8 Datasets . . . 24
2.9 Realistic User Behaviour . . . 25

3 Research Problem Analysis **27**

3.1 Self-adapting Distributed Metric Space Index Problem (SDMP) . . . 27

3.2 Analysis of SDMP 28
3.2.1 Regrouping Order Problem (ROP) 29
3.2.2 New Grouping Problem (NGP) 30
3.2.3 Continually Determining the Number of Processors (CNP) 31

4 Dynamic Update of Distributed Metric Space Index 32
4.1 Distributed LC-clusters onto Processors 32
4.2 Experiments Design 35
4.2.1 Search Engine Simulator 36
4.3 Experiments 37
4.3.1 D-P is the Fastest 37
4.4 Additional Observations 38
4.5 Conclusions 40

5 Adapting Distributed Metric Space Index 41
5.1 Recomputing G-groups 41
5.1.1 Computing H-groups 42
5.2 Experimental Evidence Supporting Hypotheses 44
5.2.1 The Number of H-groups 46
5.2.2 Search Performance of TT-S 48
5.2.3 Comparing TT-A and TT-R 48
5.3 Switch-over Performance 50
5.3.1 Experiments 51
5.3.2 Results of Experiments 51
5.4 Conclusions 52

6 Determining Number of Processors 56
6.1 CNP 56
6.1.1 Feedback Controller of Simulator 57
6.1.2 CNP Solution 57
6.2 SDMP 61
6.3 Experiments Design and Results 62
6.3.1 Search Scenarios 63
Scenario 1 64
Scenario 2 64
Scenario 3 64

6.4 Conclusions . . . 64

7 Validation of Search Engine Simulator 70

7.1 Validation of Simulator . . . 70

7.2 Experiments and Results . . . 71

7.2.1 Performance Metric Experiments . . . 71

7.2.2 The Level of Scalability Experiments . . . 72

8 Conclusion 75

8.1 Summary of Achievements . . . 75

8.1.1 Strengths of Research . . . 77

8.2 Future Work . . . 77

A A Process to Run Proposed Search Engine Simulator 80

A.1 Amazon EC2 configuration . . . 80

A.2 Shell scripts . . . 81

A.3 Search engine simulator . . . 83

Bibliography 86

List of Figures

2.1 Simplified search engine architecture adopted from [11]. 6
2.2 Parallel processing architecture Adopted from [26]. 7
2.3 Example clusters built using the LC-cluster building algorithm. On the right, the LC-clusters c_1, c_2 and c_3 adopted from [12, 14]. 11
2.4 For $R(q_1, r)$ we need to consider the current bucket (c, r_c, I) and the rest of the centers. For $R(q_2, r)$ we consider only the current bucket. For $R(q_3, r)$ we can avoid considering the current bucket (Adopted from [12]). 12
2.5 The role of the metric space in both the index building and search processors. Also showing how the index is distributed among the processors. . . . 15
2.6 Searching using distributed metric space index 16
2.7 K-Means clustering algorithm illustration (adopted from [37]). 18
2.8 Search engine deployed in cloud 22
2.9 Feedback Controller of search engine simulator 23
2.10 Example of Realistic Query Sequence 25

4.1 Re-distributing LC-clusters from 3 to 6 processors using D-P 35
4.2 The results of switch-over performance experiments. 39

5.1 Impact of increasing the number of H-groups ($= w * p$) on performance. . . 47
5.2 TT-S and TT-R produce similar throughput, measured separately for increasing p' (E3) and increasing p/p' (E4). 49
5.3 TT-A and TT-R lead to a similar maximum throughput after switchovers with various p' and with various ratios. 50
5.4 TT-A is faster than TT-R in switchovers with with various p' and various ratios. 53

5.5 TT-S and TT-A produce similar result of TT-R that show D-P is the faster redistributed method. The black line in the figures show time of step2 of switch-over when run one of transition types (Under the line) and time of step 3 when use the ROP solution (D-P) (above the line). 54
5.6 TT-S and TT-A produce network load like TT-R that show D-P and D-I have less load than D-S. 55

6.1 Feedback controller of search engine simulator 57
6.2 Computing p' when query workload is increasing 59
6.3 Computing p' when query workload is decreasing 60
6.4 Query rate in the Sogou log (adapted from [43]). 62
6.5 In **Scenario 1**, search engine show good response time when the value of *lookahead-time* was suitable in E1, however, a small *lookahead-time* lead to high response times in E2. 66
6.6 The search engine increases size too quickly when *lookahead-time* is too large in E3 **Scenario 1**. 67
6.7 In **Scenario 2**, search engine show good response time when the value of *lookahead-time* was suitable in E1, however, a small *lookahead-time* lead to high response times in E2. 68
6.8 The algorithm does not cope well with very sharp changes of the workload in **Scenario 3**. 69

7.1 Comparing the number of distance evaluation performed for the queries Sets using our simulator and a third party implementation 72
7.2 Scalability of the search engine simulator. 74

A.1 Clustering package classes 84
A.2 Network load classes 85
A.3 Search engine package classes 86

List of Tables

2.1 Instances details 22

4.1 Mapping the LC-Clusters when switching over ($p = 3 \rightarrow p' = 6$), (the bold numbers indicate clusters that have to be relocated) 34

4.2 Mapping the LC-Clusters when switching over ($p = 6 \rightarrow p' = 3$), (the bold numbers indicate clusters that have to be relocated) 34

4.3 Experiments of switch-over performance 37

5.1 Experiments of switch-over performance using TT-A and TT-S 51

6.1 Elements of Feedback Controller of Search engine simulator 57

1 Introduction

Since 1993, the Internet has been growing rapidly and so too have the number of new users [1]. As more data such as images, video and audio are available on the Internet, a data search has become important to many users [2, 3, 4, 5, 6]. More than 4.64 billion pages were contained within the Indexed Web by Monday, 13 April, 2015[1]. As Jonassen mentions in [1], in June 2011, the sites of Facebook, Microsoft and Yahoo! have more than half a billion unique visitors each, and the sites of Google had already passed 1 billion mark.

The growth in the amount of data available for search, rapidly increasing the numbers of distributed data centers housing thousands of computers. These increases have driven companies to build large data centers and increase the number of processors which consume enormous amounts of electricity and require a huge infrastructure as support. For example, Google has invested around $1 billion on building and running their data centers in 2011 [1][2] and $7.3 billion in 2013 [1][3].

[1] http://www.worldwidewebsize.com,The size of the World Wide Web, visited April 13, 2015

[2] http://www.datacenterknowledge.com/,Google Spent $951 Million on Data Centers in 4Q, January 2012,visited April 13, 2015

[3] http://www.datacenterknowledge.com/,Google Spent $7.3 Billion on its Data Centers in 2013, FEBRUARY 2014,visited April 14, 2015

The search engine is a popular term for an information retrieval system. It is used to retrieve from large collections of information of an unstructured nature that satisfies a user's query [44]. The objective of a search engine is to answer queries well and fast using a large data collection, in an environment that is constantly changing. Such a goal implies that a search engine needs to cope with Web growth and change, as well as growth in the number of users and variable searching patterns (user model). For this reason, the system must be scalable. Scalability is the ability of the system to process an increasing workload as we add more resources to the system and reducing it when workload decreased. Scalability is not the only important aspect, the system must also provide high capacity, where capacity is the maximum number of users that a system can sustain at any given time, given both response time and throughput goals. Finally, the system must not compromise quality of answers, as it is easy to output bad answers quickly.

The main two objectives of the search engine are providing results matching to a user's query and reducing the time and resources needed to assign this information to the user. Using smart indexing data techniques with modern technology such as cloud should help a search engine to achieve these objectives.

A typical search engine distributes its search index into multiple processors to achieve a sufficiently high throughput [14, 15, 23, 24, 25, 33, 7]. However, the workload of a search engine typically fluctuates. Therefore, it is desirable that a search engine adapts its size to avoid wasting resources when the workload is low and to avoid unacceptable delays when the workload is high. If the engine is deployed in an Infrastructure as a Service (IaaS) cloud, the cloud facilitates allocate or deallocate computer resources to or from the engine.

Such an adaptive search engine repeatedly determines the number of processors to use, appropriately regroups the search data to form a new search index, and re-deploys the data onto the processors according to the new index.

Traditionally, the users used the keyword-based search as the main tool for finding information in the Web. Also online applications use keywords as one of the essential components for searching. For example, YouTube (Google) uses keyword search for finding a gigantic archive of videos and Flickr (Yahoo!) does the same for finding pictures. Google, Microsoft and Yahoo! use search to filter and organize news, media and private emails.

In addition, search is often conducted as a similarity search. Similarity search has received much attention due to increasing interest in retrieving multimedia data [13]. Modern search engine systems have to manage collections of billions of objects. It is implemented using a large pool of computing servers which share the engines large index. Each server holds the part of the index relative to a disjoint sub-collection of documents.

The metric space index used to represent these objects are very large data structures, the form of which can have a big impact on the quality and the speed of search engine algorithms.

1.1 Research Questions

Our research focusing on similarity search with metric space indices and due to restricted resources only on small-scale distributed search engines. The main question addressed in this thesis is:

RQ *How should a distributed search engine adapt its size when its query workload changes?*

Seeking to answer this question, the research is organized as an exploratory, iterative process consisting of observation or literature study in order to define a problem or to state a hypothesis, proposal of a solution, qualitative evaluation against the baseline and publication of the results. In order to adapt the search engine size when query workload changes, we need to determine a new number of processors and regroup search data to be redistributed onto the processors. Thus, the research and the contributions within this thesis are divided into the following three main directions:

RQ-A *How to continually keep determining a suitable search engine size ?*

RQ-B *How to regroup the distributed data to changed requirements of the search engine?*

RQ-C *How to redistribute new groups of data among new processors when adapting the search engine size ?*

1.2 Contribution

We contribute solutions for the above research questions. The main contributions of the research are as follows:

1. We propose an algorithm to determine the number of processors when the workload changes (RQ-A solution, Algorithm 6.1.1 in Chapter 6).
2. We provide an algorithm to re-group the search data when search engine changes size (RQ-B solution, Algorithm 5.1.2 in Chapter 5). (This chapter is joint work with the supervisor based on our joint paper) .

3. We provide an efficient method to re-distribute data among processors when search engine changes size (RQ-C solution, D-P method in Chapter 4).

4. We create a search engine simulator and tools for deploying it on Amazon EC2.

A solution to the problem of self-adapting distributed search engines sizes in the cloud have been provided in this research as explained in details in Chapter 6.

1.3 Structure of the Thesis

In Chapter 2, we present the background material on classification and information theory which is necessary in order to understand the contributions of the thesis. We also briefly discuss cloud technology.

In Chapter 3, we outline and analyse the research questions and also discuss criteria of success in finding answers to the questions.

In Chapter 4, we discuss methods of distributing (or re-distributing) newly re-grouped data onto processors while keeping the search engine responsive and also the design and results of our experiments to validate and evaluate the different methods.

In Chapter 5, we propose an algorithm for regrouping distributed data to changed requirements of the search engine and evaluate its effectiveness.

In Chapter 6, we look at an algorithm for continually updating the search engine size and the design and results of our experiments to validate and evaluate our algorithm.

In Chapter 7, we conclude this thesis, reviewing the material presented. We suggest several interesting future directions which have arisen during the course of this research.

1.4 Publications and Software

The work presented in this thesis has resulted in one publication:

Al Ruqeishi, Khalil, and Michal Konečný. "Regrouping Metric-Space Search Index for Search Engine Size Adaptation." Similarity Search and Applications. Springer International Publishing, 2015. 271-282.

Software

To support the experimental studies in this thesis which were developed together constitute a search engine simulator deployable on the Amazon Elastic Compute Cloud (Amazon EC2) platform http://duck.aston.ac.uk/khalil/thesis.

2 Background

In order to ease the understanding of our work and contributions, this chapter gives an overview of the technical background and related work. In Section 2.1, we present the search engine distributed architecture we assume in research. In Section 2.2, we discuss the similarity search. We present metric space indexing algorithms in Section 2.3. Parallel query processing is discussed in Section 2.4 including index partitioning. In Section 2.5, we introduce distributed metric space index problem. In Section 2.6, we briefly discuss cloud architectures and efficient deployment of a search engine in a cloud. Finally, we present the datasets used in this research in Section 2.8.

2.1 Search Engine Distributed Architecture

Search engines are committed to answering a large number of different queries and handling complex information in real time. According to Levene [11] a search engine includes five parts: database, indexer, search index, query engine, and search interface as shown in Figure 2.1. The database stores and categorizes the objects. These objects are scanned by the indexer, which will create the search index. The query engine processes the queries by retrieving relevant objects from the search index and combining these objects to provide

a ranked list of results. The search interface displays the results [21].

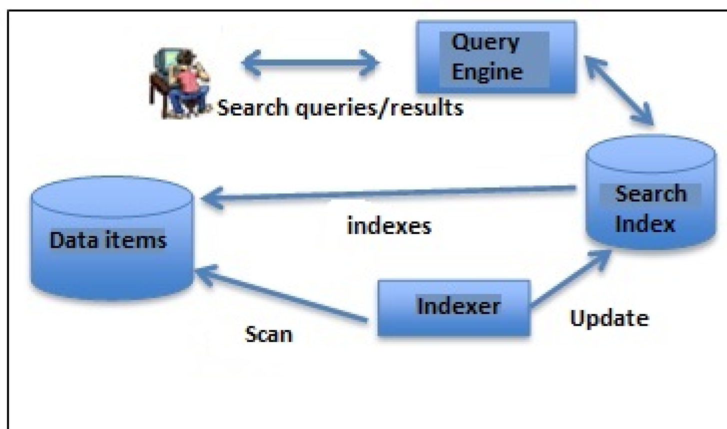


Figure 2.1: Simplified search engine architecture adopted from [11].

As explained in [22], we can categorize search engine mechanisms based on two key functions supported by the three key components: database, an indexer, and a query processor. These functions are as follows:

- index processing: identifying and storing documents for indexing, transforming documents into index terms or features, taking index terms and building data structures that enable fast searching.

- query processing: supporting the creation and refinement of query, displaying results, generating ranked lists of documents for a user's query, monitoring and measuring effectiveness (quality of results) and efficiency (response time and throughput).

As in [14, 15, 24, 25, 26, 27, 28, 29, 30, 31] we assume a parallel query processing architecture. In this architecture, there is a receptionist machine called query broker which receives queries from users and distributes query processing onto processors, then collects, merges and orders the results on the basis of their relevance as shown in Figure 2.2. Moreover, the broker has a cache of query results (called query log) which holds the results of most frequently submitted queries from users, which are used for successive submissions of the same query. A search engine is composed of one or more broker machines and a collection of p processors where p is typically several tens for a small search engine and thousands for a large search engine.

2.2 Similarity Search

As more data such as images, video and audio are available on the Internet, huge data centers become important to hold these data objects. These data objects need to be

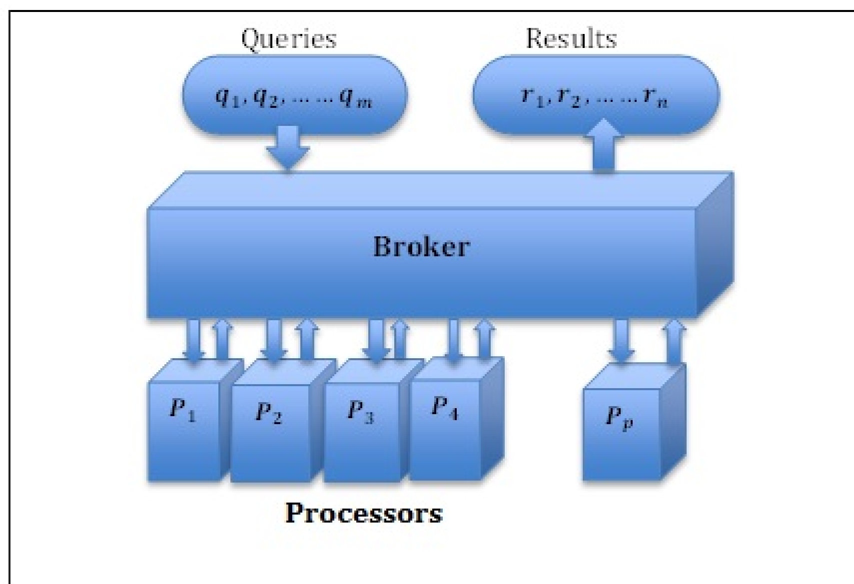


Figure 2.2: Parallel processing architecture Adopted from [26].

structured and manipulated effectively in order to enable users to access specific data.

Most of the traditional data centers made up of a simple attributed data and more complex in nature which make too difficult to measure the relations between the data objects when the data increased. Thus, to deal with increased data, data reduction approaches are employed using high dimensional vectors. A vector representation enables these approaches to easily measure the distance between two data objects [45].

Searching for objects that are close or similar to a given query object is called similarity searching or proximity searching [12]. Similarity searching has received much attention due to increasing interest in retrieving multimedia data such as images, video and audio [13]. It is known that employing a data structure for indexing objects will make it faster to answer queries [14]. Many applications include a search engine for searching multimedia objects annotated with unstructured text, such as in molecular biology, fingerprint matching, voice recognition and multimedia databases. All these applications share a common framework, which is in essence, to find close objects, under some suitable similarity measure, among a large set of objects.

Similarity is represented by using a distance function, which satisfies the triangular inequality, over a set of objects called a metric space [12]. Metric space, as a very general data abstraction, is applicable to a wide variety of multimedia data types [19].

2.2.1 Metric Spaces

A metric space (X, d) is composed of a universe of valid objects X and distance function $d : X \times X \rightarrow \mathbb{R}$ which will denote a measure of distance between objects [12, 13, 14, 15]. A finite subset $U \subset X$, with size $n = |U|$, is used to represent the database i.e. the collection of objects to be searched. In a metric space, the properties of the distance function $d : X \times X \rightarrow \mathbb{R}$ are typically characterized as:

- $d(x, y) \geq 0$ and if $d(x, y) = 0$ then $x = y$ (strictly positiveness),
- $d(x, y) = d(y, x)$ (symmetry),
- $d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality).

2.2.2 Similarity Queries

A similarity query can be one of three basic types:

- Range query $R(q, r)$:

  This query retrieves all the objects $u \in U$ within distance r to q i.e. $R(q, r) = \{u \in U | d(q, u) \leq r\}$.

- Nearest neighbor query $NN(q)$: This query retrieves the object in U which is nearest to q, that is, $NN(q) = \{u \in U | \forall v \in U, d(q, u) \leq d(q, v)\}$

- k-nearest neighbor query $kNN(q, k)$:

  This query retrieves the k closest objects to q in U. More precisely, $kNN(q, k) \subseteq U$ such that $|kNN(q, k)| = k$ and, for every $u \in kNN(q, k)$ and every $v \in U \setminus kNN(q, k)$, it holds that $d(q, u) \leq d(q, v)$.

We mainly focus in our research on k-nearest neighbor queries. While search engines typically receive **k-nearest neighbor** (kNN) queries, i.e. "find k nearest objects to a specified object q for a small k" [14], search engines would usually translate such queries to **range queries** $R(q, r)$, i.e. "find all objects within distance r from q", because range queries are easier to distribute and process. Our engine also adopts this approach.

2.3 Metric Space Indexing Approaches

According to [12, 13, 14, 15] metric space indexing strategies can be classified into two main approaches :

- Pivot-based approaches: select a number of objects from the metric space X as pivots $(pt_1, pt_2, \ldots\ldots, pt_s \in X)$ and categorize each object $u \in U$ according to their distance to these pivots. The distances between u and the pivots and between the query q and the pivots are used together with the triangle inequality to filter out objects in the database, without actually evaluating their distances to the query q [12, 13, 14, 15] .

- Cluster-based approaches: A set of centers $c_1, c_2, \ldots\ldots, c_k$ is chosen to divide the collection of objects into groups called clusters and each object in the space is associated to its closest center c_i. Thus similar objects typically fall into the same cluster. The key idea is to divide the space into zones as compact as possible [12, 13, 14, 15], to reduce the numbers of items to search exhaustively.

These indexing approaches have been proposed in [12, 17, 18]. A good survey on metric spaces can be found in [13]. Next, two concrete approaches will be described as examples of the above two general approaches:

2.3.1 Sparse Spatial Selection (SSS) Approach

The Sparse Spatial Selection (SSS) approach is pivot-based. It selects a group of objects as pivots from the database and then computes the distance between the objects of database and the pivots. The SSS construction approach can be divided in two steps: the pivots selection process and the distances computation process. To select the pivots set, let (X, d) be a metric space, $U \subseteq X$ an object collection, and M the maximum distance between any pair of objects i.e $M = \max\{d(x, y)/x, y \in X\}$. The set of pivots is initialized by only the first object of the collection. Then for each object $x_i \in U$, x_i is chosen as a new pivot if its distance to every pivot in the current set of pivots is equal or greater than αM , where α is a constant whose optimal values are between 0.38 and 0.5 [20, 33]. One of the good features of the pivot selection technique is being dynamic and adaptive when the database is growing. The set of pivots adapts itself when a new element is added to the database [38].

2.3.2 List of Clusters (LC) Approach

List of clusters (LC) is a simple and effective technique used to cluster finite subsets of metric spaces.

The index is built by choosing a set of objects (centers) $c \in U$ with radius r_c where each center maintains a bucket I that keeps track of the objects contained within the

cluster (c, r_c, I_c). Each bucket I_c contains the nearest k points to the respective center c. Thus the radius r_c is the maximum distance between the center c and its k nearest neighbors. The index has been computed as specified below:

- Select the first center $c_1 \in U$ randomly to determine a LC-cluster (c_1, r_1, I_1), where I_1 is the set $kNN_U(c_1, k)$ of k nearest neighbours of c_1 in U and r_1 is the distance between center of c_1 and the most distant point in I_1.

- Then, select the next center c_2 such that the center is the farthest from c_1 from the remaining set $U_1 = U \setminus (I_1 \cup \{c_1\})$. This second center c_2 determines a new LC-cluster (c_2, r_2, I_2) where I_2 is the set $kNN_{U_1}(c_2, k)$ of k nearest neighbours of c_2 in U_1 and r_2 is the distance between center of c_2 and and the most distant point in I_2.

- The above steps are carried out until there are no remaining centers in U_{n-1}.

This process is specified in detail as Algorithm 2.3.1 below.

Algorithm 2.3.1: Build (U, k) adopted from [12]

Input:
U: a set of data points,
k: number of data points in each cluster.

Output: List of LC-clusters: (c_1, r_1, I_1),(c_2, r_2, I_2),......,$(c_{i-1}, r_{i-1}, I_{i-1})$;

```
1: i=1
2: U_1=U
3: Select center c_1 ∈ U randomly.
4: while (U_i ≠ ∅) loop
5:    Compute the radius r_i to enclose the k nearest neighbors of c_i in U_i.
6:    I_i = {u ∈ U_i \ {c_i}, d(c_i, u) ≤ r_i}
7:    U_{i+1}=U_i \ (I_i ∪ {c_i})
8:    Select c_{i+1} ∈ U_{i+1} such that Σ_{k=0}^{i} d(c_{i+1}, c_i) is as large as possible.
9:    i = i + 1
10: end loop
```

Figure 2.3 illustrates this algorithm using three clusters with centers c_1, c_2 and c_3 in the order of construction. Space can be divided into clusters using two ways: taking a fixed radius for each cluster or using a fixed size. In [14, 23, 24, 25] the authors use clusters with fixed size of k objects to promote a good load balance across processors and we also adopt this approach to create 10000 clusters for our experiments. The algorithmic complexity is $\mathcal{O}(n^2/p^*)$ time for fixed radius partitions and $\mathcal{O}(n^2/m^*)$ time for fixed size partitions, where p is the expected bucket size and m^* is the a fixed number of elements inside each cluster.

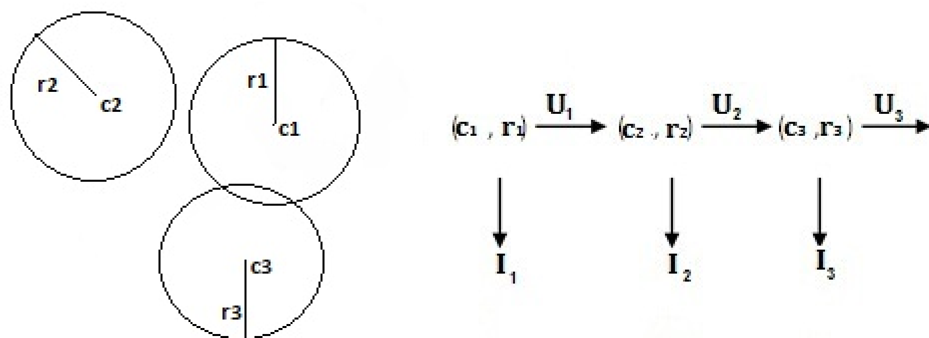


Figure 2.3: Example clusters built using the LC-cluster building algorithm. On the right, the LC-clusters c_1, c_2 and c_3 adopted from [12, 14].

This first step described above generates locally correlated clusters. Next, we turn to how to use these clusters to search for the results of a range (q, r). The search algorithm is shown in Algorithm 2.3.2. The idea is that, during the processing of a search query (q, r), if the first cluster center is c and its radius r_c, then start measuring the distance $d(q, c)$ and adding the center c of the cluster (c, r_c, I) to the result set if $d(q, c) \leq r$. Then, we scan exhaustively I from the cluster (c, r_c, I) only if the query ball (q, r) intersects with the cluster (c, r_c, I). However, if the query ball (q, r) is totally and strictly contained in the cluster (c, r_c, I), we only consider this cluster and ignore others because all the points inside the query ball have been inserted into I. Moreover, if the query ball does not cross with any clusters from U then that cluster is ignored. Figure 2.4 illustrates various situations that can occur between a range query (q, r) and a cluster with center c and radius r_c. In our research, the value of r was 100 for all the range query (q, r) either search query or training query that used to create the Index as explain in details in Section 2.5.1.

As stated in [12], a cluster-based model such as LC is much more efficient than other known approaches to index high dimensional spaces because it takes up little memory space and it is simple to develop and use. In addition, the LC (cluster based) approach is shown to deal better with high dimensional metric spaces than the SSS (pivot based) approach as pivots require more memory [12, 16]. Marin et al in [33] achieve the best performance by combining LC and SSS indexing methods in their hybrid index. However, we use pure LC-cluster because it's simpler and the improvement of hybrid over LC is not very significant.

Algorithm 2.3.2: Search (L, q, r)

Input:
(q, r) — a range query,
C: list of LC clusters.

Output: L: list of results

1: L = empty list
2: for all (c, r_c, I) in C (in the order given) repeat
3: compute $d(c, q)$
4: if $d(c, q) \leq r$ then add c to L
5: if $d(c, q) \leq r_c + r$ then add all $a \in I$ with $d(a, q) \leq r$ to L
6: if $d(c, q) > r_c - r$ then break from the loop
7: end for
8: return L

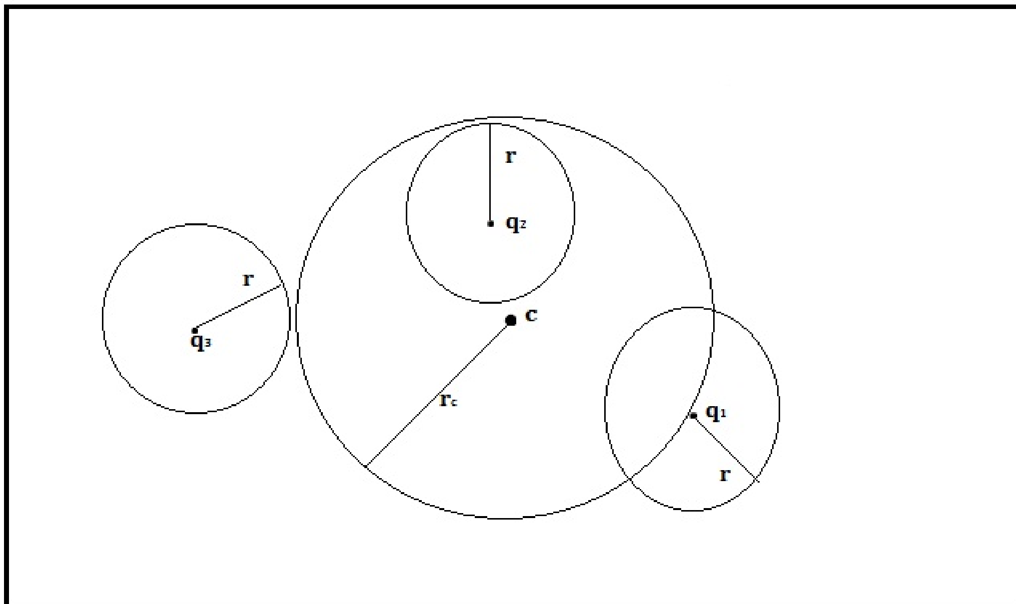


Figure 2.4: For $R(q_1, r)$ we need to consider the current bucket (c, r_c, I) and the rest of the centers. For $R(q_2, r)$ we consider only the current bucket. For $R(q_3, r)$ we can avoid considering the current bucket (Adopted from [12]).

2.4 Parallel Query Processing in Distributed Metric Space

According to [23], distributed metric space query processing was first studied in [32]. In[32], the authors proposed the following four query processing methods for a local index:

- Concurrent processing: Submit query to all p processors, collect top k results from each processor and select the best k among p processors.
- Selective processing: Visit each processor sequentially and in each visit determine if there are better results than the current top-k. This method aims to reduce the number of intermediate results.
- Two-phase processing: Visit $m < p$ processors and collect $m \cdot k$ results. Then select the global top k and contact the remaining $p - m$ processors to determine if they are able to produce better results than global top k for each query.
- Probabilistic processing: Perform iterations by asking top-k/p results in each iteration. As objects are distributed randomly onto processors there is a high probability that the best top k will be determined in few iterations.

This work was extended in [15] for the LC-based approach, studying various forms of parallelization.

This study will pay attention to the centralized model like the broker model as shown in Figure 2.2. This processing architecture assumes that the broker receives queries from users and distributes their processing onto the processors. The broker sends each query to a circularly selected processor which becomes the **ranker** for the query. The processors produce query answers and pass the results back to the broker [14, 23, 24, 25].

In [15] the authors studied the following forms of parallelizing the LC approach:

- Local index Local centers (LL): After distributing the vectors that correspond to the search objects onto the processors, The LC clusters in each processor are constructed locally. The ranker sends each query to all processors and a sequential LC search algorithm will be applied in each processor and the local top k results sent back to the ranker.
- Local index Global centers (LG): This uses a similar method to LL in order to index objects locally in each processor. The ranker performs a parallel computation to determine the cluster or center that must be visited by calculating the query plan. Query plan will be the set of LC-Clusters that intersect with the query.

- Global index Global centers (GG): In this case the LC clusters are built for the whole database of search objects and the clusters are distributed onto processors in such a way that each whole cluster is placed in a single processor. At end of this process the whole LC clusters and the index plan is distributed in such a way that the clusters are distributed onto p processors as shown in Figure 2.5 and as explained below :

  1. Index Planner: Index Planner is responsible for computing clusters and distributing them to the processors. It sends each processor not only its LC-clusters, but also an **index plan**, which is a map indicating for each LC-cluster on which processor it is. The index plan is used by the processor when it acts as a ranker for a query to determine which processors to contact regarding the query.
  2. Broker: As above, the broker receives queries from users and distributes query processing onto processors (Rankers). In our simulator, we used broker as the end users that sends queries to the processors.
  3. Ranker(Processor): Upon receiving a query, the ranker processor calculates the distance among the query and all of the centers across processors and formulates a *query plan*, namely the set of LC-clusters that intersect the ball of the range query (q, r). The ranker sends the query and its query plan to the processor p_i that contains the first cluster to be visited, namely, the first LC-cluster that intersects the query ball. Then p_i processes all LC-clusters that intersect (q, r). For each such cluster, p_i compares (q, r) against the data points stored inside. The processor p_i then returns to the ranker the objects that are within (q, r) and passes the query and its plan to the next processor specified in the plan. This continues until all the processors in the query plan have returned their results to the ranker. The ranker sorts the query answers and passes the best k back to the broker as shown in Fig. 2.6. Each processor acts both as a processor in charge of processing a subset of LC-clusters and as a potential ranker.
  4. Processor: Processors processes all clusters that intersect with (q, r) and returns to the ranker all objects in the clusters. This continues until all the processors in the query plan have returned their results. Each processor acts both as a processor in charge of processing its LC-clusters and as a ranker of different subset of all queries to the ranker.

[15, 14, 23, 24, 25] concluded that GG achieves better performance than the local index as it reduces the average number of processors contacted per query. The GG method will

be adopted for this study. An attractive feature of schemes without a global index is that they lend themselves to Peer-to-Peer (P2P) processing, which naturally supports resizing in response to load variations. For example, [34] presents a distributed metric space index as a P2P system called M-index. Nevertheless, M-index is based on a pivot partitioning model, which has a high space complexity and P2P cannot use GG. For further related work using P2P metric space indexing see e.g. [39, 40, 41, 42].

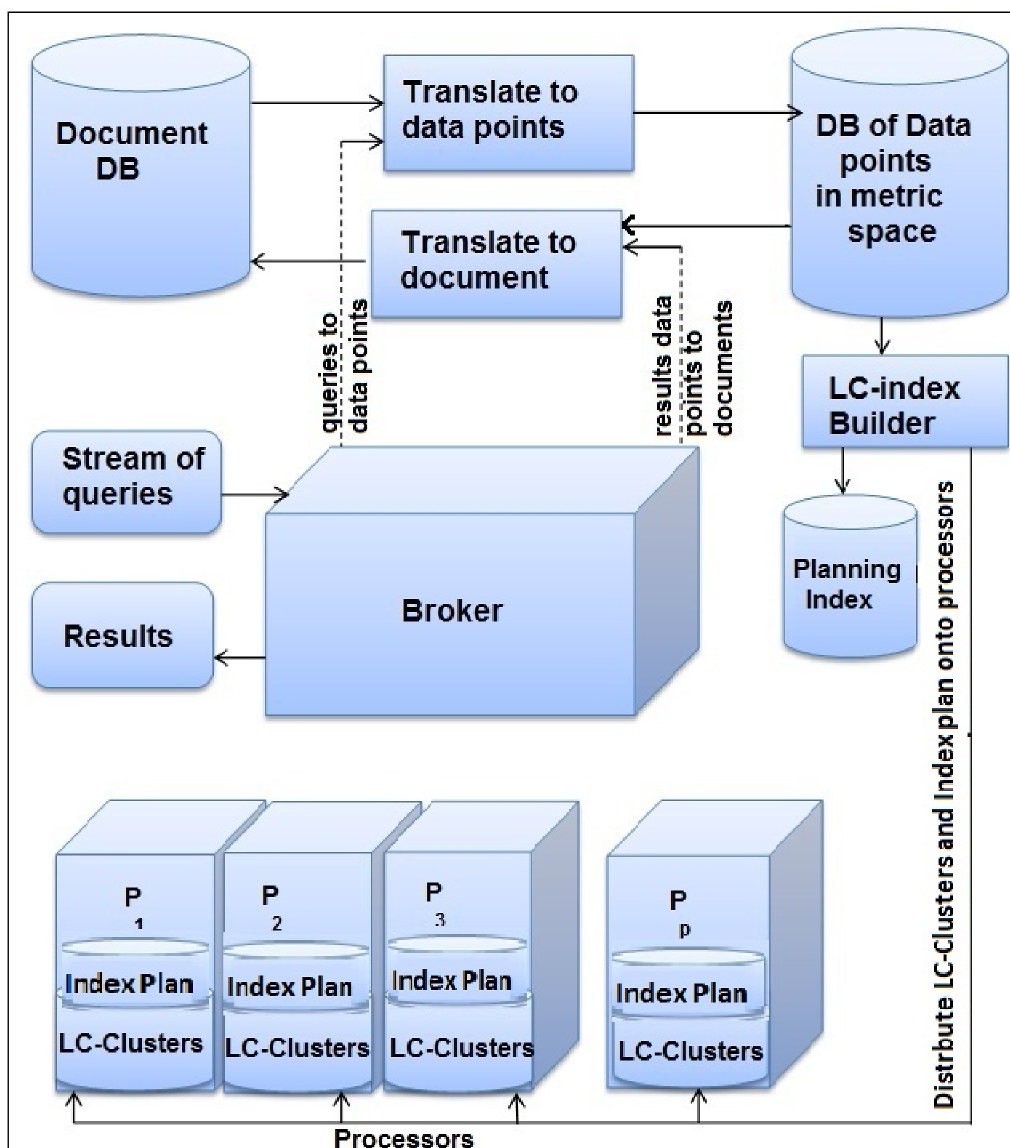


Figure 2.5: The role of the metric space in both the index building and search processors. Also showing how the index is distributed among the processors.

2.5 Distributed Metric Space Index Problem (DMP)

The previous discussion on local versus global distributed indexing concluded that the global index strategy helps the broker to get the right top k results quickly. However, it

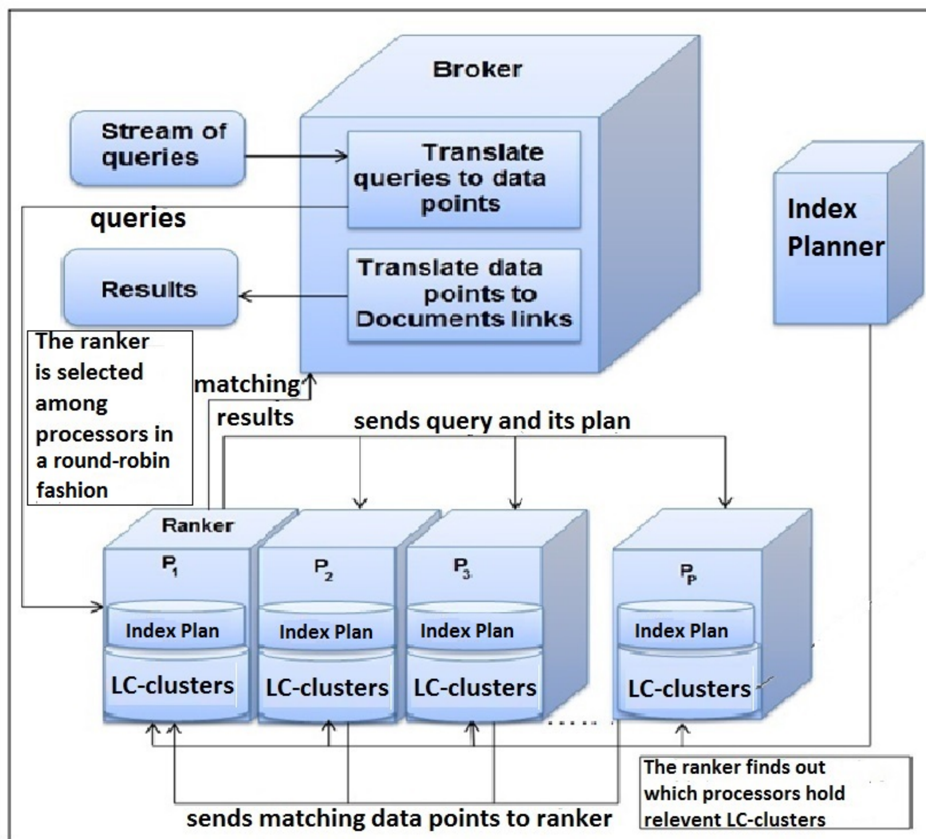


Figure 2.6: Searching using distributed metric space index

remains to consider the problem of how to distribute the global metric space index among processors to improve search performances when a large stream of queries hit the search engine. We call this the Distributed Metric space index Problem (DMP). More formally, a solution to DMP is an algorithm with the following:

- **Input:** C a set of LC-clusters, $p > 0$ the number of processors.
- **Output:** A partition/grouping of C where the number of groups equals p (Each group is be deployed on a different processor) .

The algorithm is measured by the following criteria:

- Performance of the search engine in terms of maximum throughput and average and maximum response times.
- Balance query processing by using a small fraction of the distributed processors to process a query and all processors dealing with a similar amount of queries.

2.5.1 Distributed Metric Space Index for Search Engine

In [14], Marin at al propose a solution to DMP supporting a large-scale search engine. The authors proposed two types of algorithms for DMP: query-based algorithms and query-independent algorithms. Query-based algorithms, namely Km-Col and Km-Row, partition a set of LC-clusters based on a set of sample queries Q.

Km-Col introduces several levels of groupings. We adopt the following notation for these levels:

- Data points U: points in a metric space, representing the objects of the search
- LC-clusters C: partition of U, grouping nearby data points
- H-groups H: partition of LC-clusters U, grouping clusters whose centers are near in the metric d_Q derived from Q.
- G-groups: partition of H, one group per processor

LC-clusters are computed using the List of Clusters (LC) algorithm (Section 2.3.2) , with the natural metric on the data points. H-groups are computed from LC-clusters using K-means with the query-vector metric d_Q. Due to the nature of K-means, H-groups are of varying sizes. G-groups are computed from H-groups using a procedure we call Group-Balanced, which attempts to balance their sizes.

As explained above, H-groups are computed from LC-clusters using K-means with the query-vector metric d_Q.

K-means Clustering

K-means clustering is a method of grouping items into k groups. The grouping is achieved by minimizing the sum of distances between objects and the corresponding centroid. A centroid is an element of the group that tends to be near its centre. The main idea of the K-means Algorithm is explained in 2.5.1 :

Algorithm 2.5.1: K-means (k, X, d)

Input:
k —number of groups,
X —set of objects to be clustered,
d— a metric on X.
Output: Partition Y of X with $|Y| = k$

1: Select k initial " means " randomly.
2: Create k groups by allocating each of the objects into the nearest mean.
3: Compute the actual mean of each group and replace the previous means with the new means.
4: Repeat the steps 2 and 3 until there are no significant changes in the groups.

This approach has the advantage of being simple. Its aim is to partition data points into k groups in which each data point belongs to the group with the nearest mean. The above steps are illustrated in an example shown in Figure 2.7.

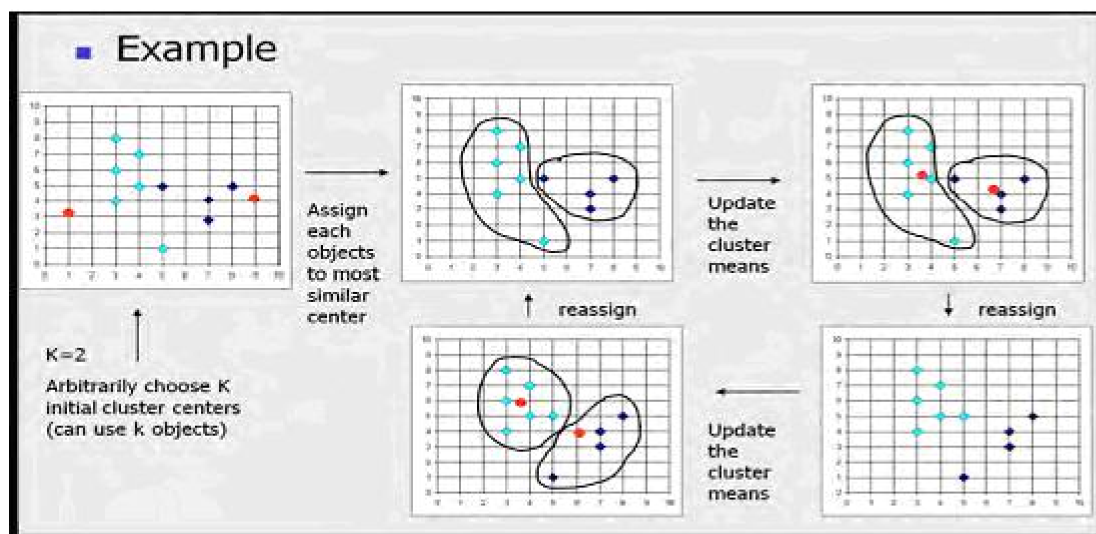


Figure 2.7: K-Means clustering algorithm illustration (adopted from [37]).

A limitation of the K-means method is a lack of discrete data analysis capability. This method requires specification of the number of groups in advance. However, it is a simple and easy to implement method of clustering. Note that the groups computed by K-means are not necessary of similar sizes i.e. some groups can have more items than

others. The work by Marin et al [14] is the basis of our research and some components of the query-based algorithms will be used in other chapters.

Query Vectors

Puppin et al [26] proposed the so-called query-vector model for partitioning search data with a focus on search performance. This model takes into account a set of sample queries obtained from a log of queries over a certain period. In effect, the search engine is trained to perform well on queries that have been observed in recent past.

Let us briefly explain the concept of query vectors. Let $Q = \{q_1, q_2, \ldots\ldots q_m\}$ be the set of sample queries and let $C = \{c_1, c_2, \ldots\ldots c_n\}$ be the centres of LC-clusters. The query vector for c_i using Q, denoted QV (Q, c_i), is the m-dimensional vector $\{b_1, b_2, \ldots\ldots b_m\}$, where $b_j = 1$ *iff* the handling of query q_j requires data c_i to be visited, otherwise $b_j = 0$. The Euclidean metric on these query vectors translates to a new metric over the data items C. The metric d_Q makes pairs of points that are near in the natural metric seem far away from each other if they are close to many queries from Q, and conversely the metric d_Q makes pairs of faraway points seem almost identical if they are not near any of the queries from the set Q. In Km-Col, query vectors are used to measure distance between LC-clusters.

Km-Col Algorithm

This algorithm uses query vectors to group the LC-clusters into $p \cdot w$ H-groups, where $w > 1$. One uses $p \cdot w$ instead of just p in order to have roughly the same number of LC-cluster assigned to each processor, which will lead to good load balance of workload among the processors. This is achieved by assigning the resulting H-groups to G-groups as explained in the following steps and specified in Algorithm 2.5.2:

1. Group the n LC-Clusters into $p \cdot w$ H-groups using the K-means algorithm where the distance used is d_Q, i.e. the Euclidean distance between query vectors.

2. Then, apply Group-Balanced (Algorithm 2.5.3 originally a part of Km-Col):

   2.1 sort the H-groups $\{H_1, H_2, \ldots\ldots, H_{p \cdot w}\}$ in non-increasing order by the number of LC clusters.

   2.2 assign $\{H_1, H_2, \ldots\ldots, H_{p \cdot w}\}$ to the G-groups $\{g_0, \ldots\ldots, g_p\}$ in such a way that the H-group which contains the biggest number of LC-clusters is assigned to the first G-group, H-group that contains the second biggest number of LC-clusters to the second G-group, and so on until we reach p-th G-groups.

2.3 Starting from the H-group that contains the $(p+1)$-th biggest number of LC-clusters and continuing in decreasing order until the last H-group which contains the least number of LC-clusters, each one is assigned to the G-group with the least number of LC-clusters.

Algorithm 2.5.2: Km-Col $(w, d_Q)(C, p)$

Tuning Parameters:
Integer $w > 0$,
d_Q — metric on C.
Input:
C a set of LC-clusters ,Integer $p > 0$.
Output:
G a set of groups with $|G| = p$ that partition C.
Extra Output: H a set of H-groups that partition C.

1: H = K-means$(C, p \cdot w, d_Q)$
2: G = Group-Balanced(H, p)
3: return G and H;

Algorithm 2.5.3: Group-Balanced (H, p)

Input:
$p > 0$,
H a set of H-groups that partition C.
Output:
G a set of groups with $|G| = p$ that partition C.

1: H_{sorted} = sort-by-decreasing-size(H)
2: **for** $i = 0; i < p; i++$ **do**
3: $G[i].insert_all(H_{sorted}[i])$
4: $G_size[i] = H_{sorted}[i].size()$
5: **end for**
6: **for** $i = p \ldots (p.w - 1)$ **do**
7: $smallest_G_i = find_smallest(G)$
8: $G[smallest_G_i].insert(H_{sorted}[i])$
9: $G_size[smallest_G_i] = G_size[smallest_G_i] + H_{sorted}[i].size()$
10: **end for**
11: return G;

2.6 Search Engine and Cloud Architecture

The term cloud computing probably comes from the cloud patterns which are used to present the Internet in various flow charts and schemes. Cloud can be defined in many ways as an umbrella term to describe a category of sophisticated on-demand computing services [35]. A Cloud Computing Architecture is a structure of cloud resources, services,

middleware and software components and the relationships between them [36]. The cloud computing architecture has its 'front end' which is the side interacting with a computer user, client or any application (i.e. web browser, etc.) required to access the cloud computing systems, and its 'back end' which comprises a network of various computers, servers and data storage systems which implement Internet-accessible on-demand services. The cloud computing architecture is relatively simple, but requires intelligent management of connections between servers and assigning tasks, as well as monitoring and measurement systems which track and control the use of resources and determine whether resources are allocated to the appropriate user. It is the easy availability of computing resources any time and anywhere which is the key feature of cloud computing.

In our context, a search engine is a type of distributed application and cloud is a platform to deploy it on. Cloud technology with its massive storage capabilities and information management can effectively support the performance that is sufficient for some types of engines and other applications. Cloud offers the facilities to increase and decrease the number of processors and facilitate a flexible allocation of the processors in order to meet changing workload demands. It simplifies the deployment of cloud services including search engines by protecting users from the essential infrastructure and implementations details and promotes redistribution of data on processors that can process the queries in a search engine. This improves the resource utilisation. This research aims to use cloud facilities to manage search engine resources efficiently. Using cloud resources to run a search engine will help to increase and decrease resources depending on the query workload as shown in Figure 2.8 .

2.6.1 A Search Engine in Cloud

One of the main services model delivered by Cloud is Infrastructure-as-a-Service (IaaS). IaaS has become a solution to reduce costs and improve resource efficiency. IaaS is characterized by the concept of resource virtualization. Virtualization enables the running of multiple Operating System (OS) instances - called virtual machines (VMs)- on the same physical server [8]. For example, Amazon EC2 runs instances on its physical infrastructure using open-source virtualization middleware [9, 10].

All the experiments of this research have been executed in the Amazon EC2 IaaS cloud using different types of virtual machine instances. The details of Amazon EC2 instances types are shown in Table 2.1.

We measure the throughput of the search engine simulator by finding the maximum throughput. Maximum throughput is the highest output throughput achieved when flood-

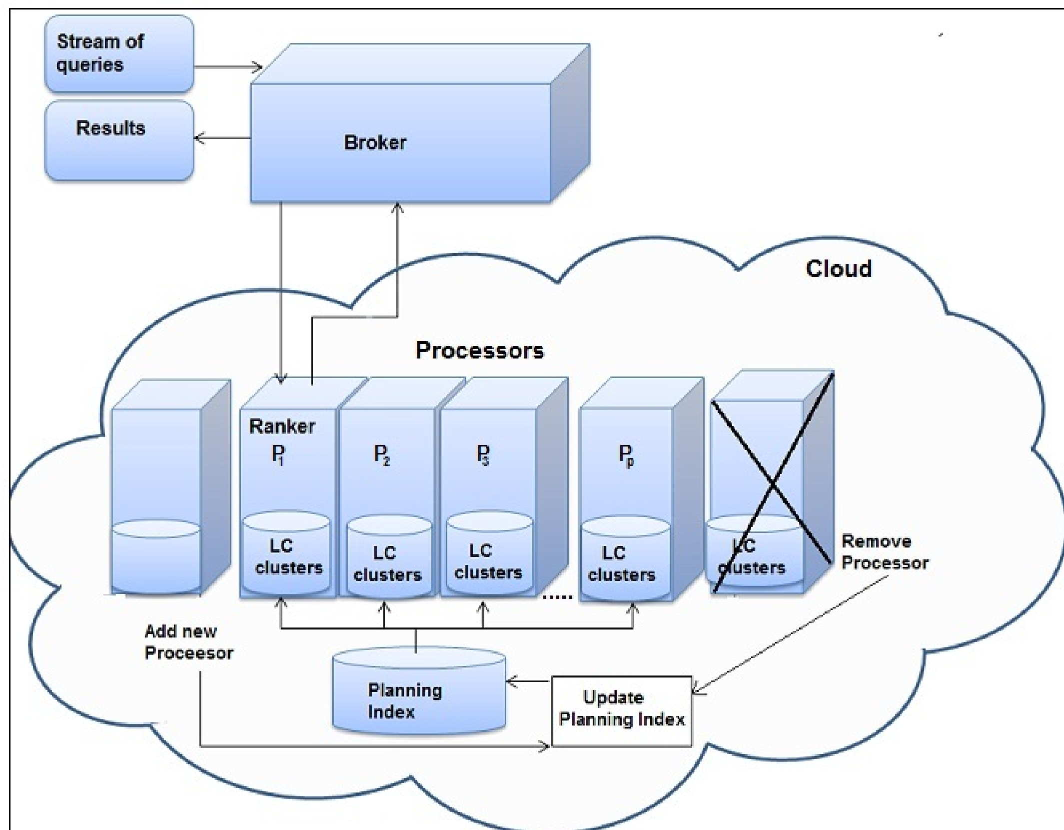


Figure 2.8: Search engine deployed in cloud

Table 2.1: Instances details

Instance Family	Type	Processor Arch	vCPU	Memory (GiB)	Instance Storage(GB)
General purpose	m1.medium	32-bit or 64-bit	1	3.75	1 x 4
General purpose	m3.medium	32-bit or 64-bit	1	3.75	1 x 4
General purpose	m3.large	32-bit or 64-bit	2	7.5	1 x 32

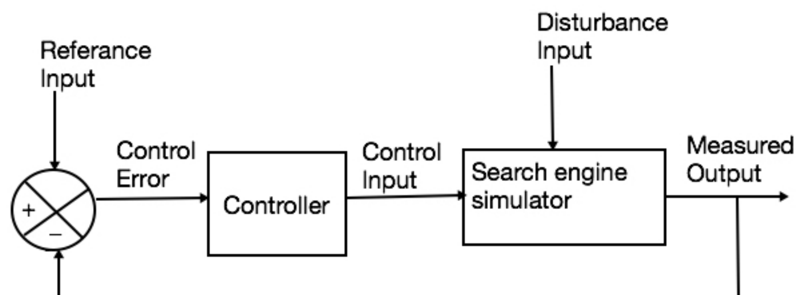


Figure 2.9: Feedback Controller of search engine simulator

ing the input with queries. We run our simulator in Amazon EC2 however EC2 have unstable network characteristics that can degrade the performance and drastically unstable throughput [54]. We have observed that the network stack efficiently facilitates the queuing of queries until the engine is able to accept them. The steps of obtaining the throughput are explained in chapter 5.

2.7 Feedback Controller of Search Engine Simulator

Feedback control can help increasingly complex computer systems adapt to changes in workloads. Control theory offers a principled way for designing feedback loops for dealing with unpredictable workload changes in systems [47].

In the previous chapter, we mentioned that we propose an algorithm to determine the number of processors there are when the workloads changes. Thus a software feedback controller has been implemented in the search engine simulator to determine the number of processors there should be when the workload changes. The feedback controller is used to measure system's outputs (CPU utilization) to achieve the specified goal.

In [48], Abdelzaher at al presented a survey of feedback performance control in computing systems. In [49], a feedback control real-time scheduling (FCS) framework has been proposed to provide performance guarantees for real-time systems.[50] presented issues regarding feedback control in a cloud computing infrastructure. Also there are many previous works on feedback-controlled adaptive resources (e.g., [51, 52, 53]), however none of the previous works match the objectives of this research i.e. adapting cloud resources when the query load changes in the search engine.

We adapt elements of feedback control from [47] to design our feedback controller in the simulator. In particular, we use the process shown in Figure 2.9, incorporating the following elements:

- **Reference input** is the desired value of the measured outputs such such as the Maximum and Minimum of CPU utilization of all processors.
- **Measured output** is a measurable characteristic of the system such as overall CPU utilization of processors, query workload and average response time.
- **Control error** is the difference between the reference input and the measured output.
- **Control input** is a parameter that affects the behaviour of the system such as the new number of processors.
- **Controller** determines the setting of the control input needed to achieve the reference input. The controller computes values of the control input based on current and past values of control error.
- **Disturbance** input is any change that affects the way in which the control input influences the measured output such as the query workload.

We used these elements to design the feedback controller of our search engine simulator as explained in Chapter 6

2.8 Datasets

In our research experiments, we have used the following two datasets:

- SIFT [1].This dataset consists of three subsets: learning, database and query. The learning set is extracted from Flickr images and the database and query descriptors are from the INRIA Holidays dataset [16]. The following table summarizes the number of descriptors in the dataset:

Vector dataset : SIFT , descriptor dimensionality d 128
Learning set size: 100,000 vectors
Database set size: 1,000,000 vectors
Queries set size: 10,000 vectors

  This dataset was used in our experiments before we attained access to CoPhIR dataset.

[1]: http://corpus-texmex.irisa.fr

- CoPhIR [2]. This is the largest publicly available collection of high-quality images metadata for research purposes. According to Bolettieri in [46], the XML data of CoPhIR collection consists of 245.3 GB disk space for 106 millions images.

  All of the experiments reported in the thesis use data derived from CoPhIR.

2.9 Realistic User Behaviour

In realistic scenarios for search engines the user queries tend to be highly skewed and the bias changes unpredictably. In our research, we aim to run our search engine simulator in a similar environment by using queries that mimic a realistic user behaviour.

Veronica Gil-Costa from Yahoo! Research Latin America Labs helped us to create queries that mimic a more realistic user behaviour. The Realistic Queries Sequence has been derived from two files:

- User Sequence: text queries from a real user query log of the Yahoo! search engine.
- Queries Set: a randomly selected set of 10,000 objects from the CoPhIR Dataset.

The Realistic Queries Sequence has been produced using the following steps:

1. Take each query from Queries Set and match it with a query (text query) from the User Sequence. Each object from Queries Set is linked to a single text query from the User Sequence i.e. IMG1 $\mapsto$ Q1, IMG2 $\mapsto$ Q2, IMG3 $\mapsto$ Q3.
2. Reproduce the order of the queries from the User Sequence using the image queries provided by Queries Set. Notice that some queries in the User Sequence can be repeated, e.g:

Figure 2.10: Example of Realistic Query Sequence

User Sequence	Realistic Query Sequence
Q1 $\mapsto$	IMG1
Q1 $\mapsto$	IMG1
Q2 $\mapsto$	IMG2
Q3 $\mapsto$	IMG3
Q3 $\mapsto$	IMG3

We used Realistic Query Sequence in all our experiments. In our experiments, we were comparing the Realistic Query Sequence and Random Query Sequence. We used Realistic Query Sequence to get a more realistic simulation. The performance does not differ very

[2] http://cophir.isti.cnr.it/

much between both sequences. The results did not change because we did not use Broker cache in our simulation. If a broker cache would be used, the sequences would probably differ more as cache impacts performance when there are repeating queries. Using Broker cache will give better throughput and will reduce the processors overload.

3 Research Problem Analysis

In Section 3.1, we introduce the main research problem. Then, we analyse the problem and explain three sub-problems and measurable success criteria in Section 3.2.

3.1 Self-adapting Distributed Metric Space Index Problem (SDMP)

In [14], the authors have proposed a unified framework that facilitates an understanding of DMP as explained in Chapter 2 and offers algorithms enabling efficient similarity search in large-scale Web search engines. However, this work does not address the following issues:

1. Dealing with *varying* query workload by varying the number of processors.
2. Redistributing C, the set of LC-clusters, when the set of active processors changes.

We focus on a variation of DMP with the addition of the above issues, called Self-adapting Distributed Metric Spaces Index (SDMP). DMP is the base for understanding the principles of SDMP and finding a solution for SDMP can give further insights to DMP. This research aims to find solution to SDMP by developing an algorithm with the following specification:

- Assume $\mathcal{P}$ is the set of processors available for use by the search engine.
- **Initial input:** G_0 a partition of C allocated onto a subset of processors $\mathcal{P}_0 \subseteq \mathcal{P}$.
- **Continual input:**
  - Set of queries Q_T over any given time interval T in the past when the engine was active.
  - Response time of all queries $\rho_T : Q_T \to \mathbb{R}^+$.
  - Average load $\mathcal{L}_T(P)$ for each processor $P \in \mathcal{P}$ over T during which P was active.
- **Continual effect:** The algorithm repartitions C and redistributes it to different processors within $\mathcal{P}$ at different times. $\mathcal{P}_t \subset \mathcal{P}$, the set of active processors at time $t \geq 0$, changing depending on the G_t (allocation of the LC-clusters to processors at time t).

A SDMP algorithm will be measured by the following criteria:

- Reduced cost of the resources required for searching and increased processor utilisation compared with non-adaptive algorithms. The utilisation is measured using the average load $\mathcal{L}_T(P)$ for all processors $P \in \mathcal{P}$ over various time period T. The cost is measured using $\int_T p_t \mathrm{d}t$ where $p_t = |\mathcal{P}_t|$.
- Maintaining a good search performance compared with non-adaptive algorithms. Search performance is measured using the average and maximum response time of all queries.

3.2 Analysis of SDMP

In other words, the SDMP solution may be regarded as an adaptive search engine which will repeatedly re-evaluate its load and, when appropriate, switch over from p active processors to a different number of active processors. Each switch-over comprises the following steps:

1. Determine the new number of processors p' based on the recent load.
2. (Re-)compute H-groups and G-groups (i.e. the index plan) for p' processors.
3. Distribute the index plan and the relevant LC-clusters onto each processor.

4. Pause search .

5. Switch to new LC-clusters and plan, de/activating some processors.

6. Resume search.

In order to have a better understanding of SDMP, we focus on three aspects and break them out into three sub-problems that correspond to the first three steps of a switch-over. Solutions to these sub-problems can produce key answers and insights for solving the overall problem. The three sub-problems are as follows:

- Continually determining the Number of Processors (CNP): In the light of the changes in the query workload in the search engine, there is a problem of determining the ideal number of processors p active at any given time to use in the search engine (Step 1 of switch-over).

- New Grouping Problem (NGP): When a change in the number of processors is determined, we have to decide the groups G that will be distributed across the processors (Step 2 of switch-over).

- Regrouping Order Problem (ROP): When we have new groups G we will need to plan how to redistribute the LC-clusters in the groups G onto processors, while minimising the switchover time and the incurred network load (Step 3 of switch-over).

However, a solution to the problem associated with continually determining the number of processors (CNP) can be judged only in relation to some solution to new grouping problem (NGP). In turn, solving NGP should take into account the chosen method for ROP so that the new groups can be efficiently deployed. Thus, solutions of these problems will be considered in reverse order.

3.2.1 Regrouping Order Problem (ROP)

Find an algorithm with the following:

- **Input:**

  $G = \{g_1, g_2, \ldots, g_p\}$, a partition of C, allocated onto processors $\mathcal{P} = \{P_1, P_2, \ldots\ldots, P_p\}$ and $G' = \left\{g'_1, g'_2, \ldots, g'_{p'}\right\}$ another partition of C. (Groups G' are typically those calculated by a solution to the NGP problem).

- **Output:** Allocation of the sets $\left\{g'_1, g'_2, \ldots, g'_{p'}\right\}$ onto processors $\mathcal{P}' = \{P'_1, P'_2, \ldots\ldots, P'_{p'}\}$ where $\mathcal{P}$ and $\mathcal{P}'$ typically share a number of common processors.

A list of steps, each of which is of one of the following types:

- Copy some LC-clusters from P_i to P'_j
- Update the LC planning index with the new location of LC-clusters in P'_j
- Delete some LC-clusters from a processor P_i.

These steps must lead from G on P to G' to P'.

A ROP algorithm will be measured by the following criteria:

- Network load: moving LC-clusters among the processors as little as possible, measured as a weighted average of the amount of transferred data and the number of transactions. For example, the algorithm should check for similar groups and keep them in the same processor.
- Switch-over performance: the time it takes to distribute the index new plan and the relevant LC-clusters onto each processor (switch-over step 3).

Solving ROP well will be more or less important depending on the type of search engine. For example, the ROP will be less relevant when the data allocated in the processors is small, such as a small search engine. On the other hand, deployment of the search engine in a cloud will increase the weight of ROP because moving a large amount of LC-clusters among rented processors in the cloud has less predictable (and usually higher) costs than doing the same on one's own hardware.

3.2.2 New Grouping Problem (NGP)

Find an algorithm with the following:

- **Input:**

  $G = \{g_1, g_2, \ldots, g_p\}$ a partition of C , Q a sample set of queries and integer $p' > 0$.

- **Output:**

  $G' = \left\{g'_1, g'_2, \ldots, g'_{p'}\right\}$ a partition of C.

A NGP algorithm will be measured by the following criteria:

- Switch-over performance: time to compute G' groups and time to deploy it using ROP solution.

- Search performance (maximum throughput) of the search engine in the cloud after reorganizing the LC-clusters according to the groups G'. The search results are independent of performance or configuration of the search engine i.e. number of processors p or G-groups.

3.2.3 Continually Determining the Number of Processors (CNP)

Find an algorithm with the following:

- **Initial input:** $G = \{g_1, g_2, \ldots, g_p\}$, a partition of C, allocated onto processors $\{P_1, P_2, \ldots\ldots, P_p\}$.
- **Continual input:**
  - Set of queries Q_T over any given time interval T in the past.
  - Response time of all queries $\rho_T : Q_T \to \mathbb{R}^+$.
  - Average load $\mathcal{L}_T(P)$ for each processor $P \in \mathcal{P}$ over T.
- **Continual effect:**

  Produce a function to measure overall load for current processors to decide when to change p and defining a new value of p.

A CNP algorithm will be measured by the following criteria:

- When the algorithm is connected with certain solutions to ROP and NGP (as described below), an efficient solution for SDMP would be obtained.

A solution to SDMP will be obtained from solutions to CNP, GNP and ROP as follows: The CNP produced a function p_t. Wherever the value of p_t changes, NGP is applied to calculate new groups from the current groups $G = \{g_1, g_2, \ldots, g_p\}$. Then the result of NGP $G' = \left\{g'_1, g'_2, \ldots, g'_{p'}\right\}$ is passed onto ROP to compute the allocation of these groups onto processors $\mathcal{P}' = \{P'_1, P'_2, \ldots\ldots, P'_{p'}\}$ and a list of steps that lead from an old to a new allocation. When these steps are executed the allocation of new groups is realized and allocated in the new processors until the value of p_t change again. In the following chapters we will study these three problems one by one.

4 Dynamic Update of Distributed Metric Space Index

In the previous chapter we looked at three sub-problems of SDMP and explained how to evaluate potential solutions. In this chapter we focus on the first sub-problem, i.e. Regrouping Order Problem (ROP). At the beginning, we briefly expand on ROP problem in Section 4.1. Then, in section 4.2 we present the results of experiments comparing several potential solutions to ROP.

4.1 Distributed LC-clusters onto Processors

Adapting the search engine size with varying workloads means that sometimes it switches over from using p processors to a different number of active processors p'. As soon as the search engine starts, the index planner computes G-groups of LC-clusters and assigns processors for these groups and then distributes these groups onto the assigned processors. When the workload changes, the search engine needs to switch over to the new number of processors. Before switching, we need to plan the reorganizing of the search data by re-distributing the LC-clusters among new processors.

The Index Planner will re-compute G-groups and create a new plan for the new number

of processors and then assign processors for each of the new G groups and re-distribute them onto the processors. Processors need the plan of G-groups when they search. This plan holds the centres of all LC-clusters and the location of each LC-cluster in the processors. As explained in the previous chapter, the input of ROP is $G = \{g_1, g_2, \ldots, g_p\}$, a partition of C, is allocated onto processors $\mathcal{P} = \{P_1, P_2, \ldots\ldots, P_p\}$ and $G' = \left\{g'_1, g'_2, \ldots, g'_{p'}\right\}$, another partition of C, calculated by a solution to the NGP problem.

We consider the following three methods to redistributing LC-clusters during searching and redistributing the LC clusters according to the given G-groups G' to a new set of processors $\mathcal{P}'$:

D-S Distributed from Scratch; The Index Planner sends all the LC-clusters to the assigned processors to replace their old LC-clusters, if any exist.

D-I Distributed using Index Planner; The Index planner compares between G' and G and sends only the missing LC-clusters to each processor. The processors will remove any old LC-clusters that are no longer assigned to them.

D-P Distributed using Processors; The Index Planner pre-computes the instructions for the processors and the processors swap LC-clusters among themselves according to the new plan.

In the D-P method, the Index Planner pre-computes instructions for processors instead of the processors doing it themselves because we expect the processors will be busy with searching and we do not want to give them more load by forcing them to calculate the LC-clusters that need to be sent to other processors. The D-S method loads the network more and requires more time as it re-distributes the whole dataset for every switch-over. It is important to maximise the reuse of previous LC-clusters when assigning processors to G-groups. For this purpose, D-I and D-P reuse most of the previous LC-clusters. Thus, they are likely to be more efficient than D-S.

In D-I and D-P, we compare the LC-clusters in the old and new plan before assigning new processors to the new G-groups. Then, we assign each currently active processor to the new G-group that has the maximum number of LC-clusters matching those LC-clusters that are currently in the processor. The steps are as follows:

1. Find the group g'_i that has the maximum number of LC-clusters matching those LC-clusters in the group g_1 on the processor p_1.
2. Assign g'_i to p'_1, where $p'_1 = p_1$.

3. Repeat steps 1 and 2 as many as times as possible.

4. If $p > p'$, the LC-clusters on the remaining G groups will be redistributed to the $\mathcal{P}'$ processors according to their location in G'.

5. If $p < p'$, the remaining G' groups will be assigned arbitrarily to the remaining processors in $\mathcal{P}'$

6. Produce the instructions for moving the LC-clusters to get from the current assignment to the new assignment given by the mapping of G' onto $\mathcal{P}'$.

   We list all the LC-clusters that are missing from or do not belong to each g'_i. The missing LC-clusters from g'_i will be received either from Index Planner or from other processors, depending on the redistribution method.

When either increasing or decreasing the search engine size, we try to reduce the network bandwidth by reducing the number of LC-cluster transfers onto processors. Examples for both types of Switch-over are shown in tables 4.2 and 4.1. In these two tables, we show the number of LC-clusters that match between G on $\mathcal{P}$ and G' on $\mathcal{P}'$. Table 4.2 shows a decrease in the search engine size from 6 to 3 processors where the number of LC-clusters that are left in the same processors are 8614 and 1386 need to be transferred to other processors. On the other hand, when the search engine increased in the size from 3 to 6 processors, 9649 LC-clusters were left in the same processors and 351 transferred to the new processors, as shown in Table 4.1 and Figure 4.1.

	p_1	p_2	p_3
p'_1	8946	0	0
p'_2	0	472	0
p'_3	0	0	231
p'_4	0	0	**165**
p'_5	0	0	**96**
p'_6	0	**54**	**36**

Table 4.1: Mapping the LC-Clusters when switching over ($p = 3 \rightarrow p' = 6$), (the bold numbers indicate clusters that have to be relocated)

	p_1	p_2	p_3	p_4	p_5	p_6
p'_1	7721	0	0	**55**	0	0
p'_2	0	444	0	**325**	**41**	**222**
p'_3	0	0	449	**82**	**412**	**249**

Table 4.2: Mapping the LC-Clusters when switching over ($p = 6 \rightarrow p' = 3$), (the bold numbers indicate clusters that have to be relocated)

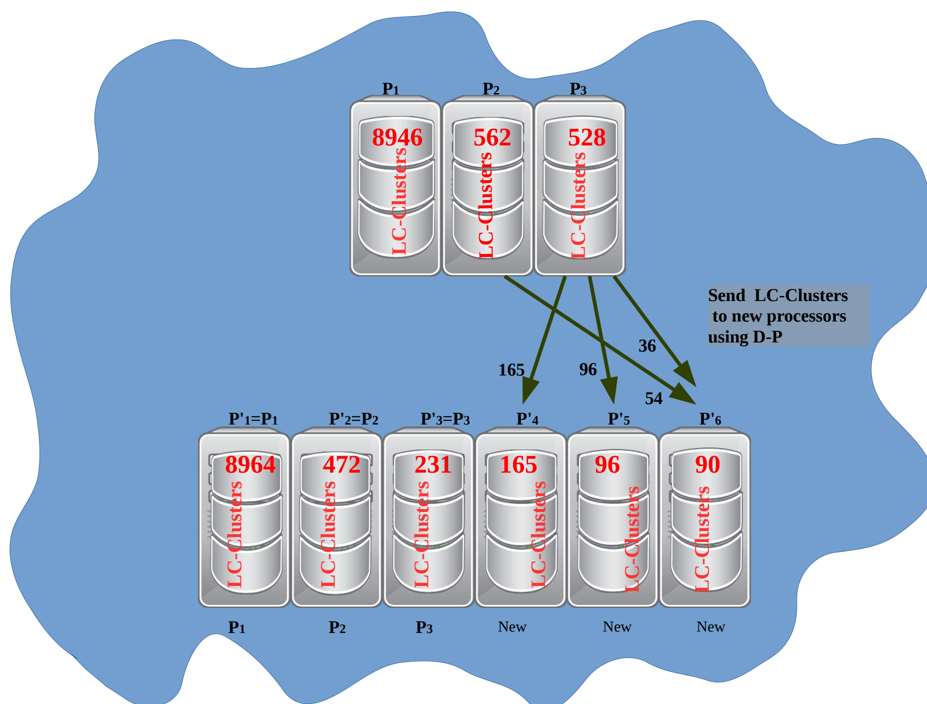


Figure 4.1: Re-distributing LC-clusters from 3 to 6 processors using D-P

As we mentioned earlier, in the D-I method, the index planner will send to each processor only the missing LC-clusters, and in the D-P method, the processors will swap LC-clusters among themselves depending on the new plan. This way we reduce the time for switch-over and also the number of LC-clusters which will be sent to the processors.

For this purpose, we formulated the following hypothesis:

ROP-H D-P is faster and does not load the network significantly more than D-S and D-I.

Experimental evidence supporting this hypothesis and a measurement of the scale of difference in speed and network load is provided in the following section.

4.2 Experiments Design

In these experiments, we aim to compare the switch-over performance of the redistribution methods D-S, D-I and D-P. Switch-over performance is measured using :

- the time from the point when G' groups have started to deploy onto $\mathcal{P}'$ until the point when the new processors become active (see switch-over step 3 in Section 3.2).
- network load.

We have to mention that grouping data and switch-over will be done in the background of processors during the searching. For this purpose, we will use the faster redistribution

methods to reduce the impact of switch-over on throughput performance. The performance is influenced by the following parameters:

1. **Search engine size evolution (SE)**: We consider only a one switchover at a time and write it as $p \rightarrow p'$. E.g., 5→8 encodes a single switchover from 5 to 8 processors. In our experiments, we use a small and large ratio p/p' of increasing or decreasing transitions sharing the same p'. More specifically, we use the four switch-overs shown in Table 4.3.

2. **Dataset (D)**: The parameter dataset D represents the set of objects that needs to be searched. In our experiments, we used a number of varying size datasets from the CoPhIR Dataset[1] (see Chapter 2). In our experiments, we used a randomly selected set of 1,000,000 objects (2.27 GB) from the CoPhIR. Each object comprises 282 floating-point number co-ordinates .

As explained in Chapter 1, the experiments to measure switch-over performance have been run on Amazon EC2 instances. The maximum number of instances used in our experiments was 14. All search processors and broker ran on m3.medium instances. while the Index planner ran on a m3.large instance.

When the search engine starts, the Index Planner computes 10000 LC-clusters using the LC algorithm. Each of these clusters have 100 objects. The Index Planner computes G-groups of LC-clusters for processors using the Km-Col algorithm. When the search engine changes its size, the Index Planner re-computes G-groups for a new number of processors and executes one of the ROP solutions and then co-ordinates the execution of the instructions to physically move the LC-clusters to the new processors (i.e.in switch-over step 3 introduced in Section 3.2).

4.2.1 Search Engine Simulator

We used the search engine simulator to conduct all the experiments in our research. We assume the search engine simulator outlined in Figure 2.6. Search engine simulator has been implemented by using Java language and Eclipse software and it contains the following four parts:

1. Index Planner: Index Planner node is responsible for computing G-groups and distributing them to the processors.

[1]http://cophir.isti.cnr.it/

2. Broker: As we stated in the Section 2.1, Broker receives queries from users and distributes query processing onto processors, then collects, merges and orders the results on the basis of their relevance. However, in our simulator we used broker as the end users that sends queries to the processors and receives the results from the Rankers.

3. Ranker(Processor): calculates query plan and sends the query and its query plan to the processors in order to get the result.

4. Processor: processes all clusters that intersect with (q, r) and returns to the ranker all objects in the clusters.

For further details about how to run a proposed search engine simulator in the cloud, please read the Appendix A.

4.3 Experiments

In these experiments, we check switch-over performance of D-S, D-I and D-P for different p, p'. We run four experiments for each redistribution method as shown in Table 4.3.

Table 4.3: Experiments of switch-over performance

Code	Redistribution method	$p \rightarrow p'$	Switching type
$E1$	D-S,D-I,D-P	8→12	small increase
$E2$	D-S,D-I,D-P	4→12	large increase
$E3$	D-S,D-I,D-P	6→4	small decrease
$E4$	D-S,D-I,D-P	12→4	large decrease

The switch-over time and network load has been measured in each experiment to support the hypothesis as explained below.

4.3.1 D-P is the Fastest

Distributing LC-clusters using D-P speed up the switching time as stated in ROP-H. The results of these experiments, as shown in Figure 4.2, support this hypothesis. The time of switch-over is less when using D-P as we use processors to re-distribute LC-clusters to other processors instead of using index planner. The Network load becomes slightly higher than D-I because D-P sends instructions to swap LC-clusters over to the processors. As explained above, this is because we expect processors will be busy with searching and we do not want to give them more load by forcing them to calculate the LC-clusters that need to be sent to other processors.

The experiments show that in this context D-P is approximately 50% faster than D-S and 10% to 25% faster than D-I. In addition, D-P and D-I reduce the network load to around 30% less than D-S. As we stated in Subsection 3.2.1, the aim was to moving LC-clusters among the processors as little as possible.

Figure 4.2(b) switchover network load, which is the amount of data transferred among processors during a switchover. We compare the amount of data transferred among processors when the search engine size enlarges and shrinks using the three distribution methods. D-P and D-I reduce the number of LC-cluster transferred among processors during switchover comparing with D-S, however the amount of data in D-P was more than D-I because each processor needs to check which LC-clusters will move to other processors.

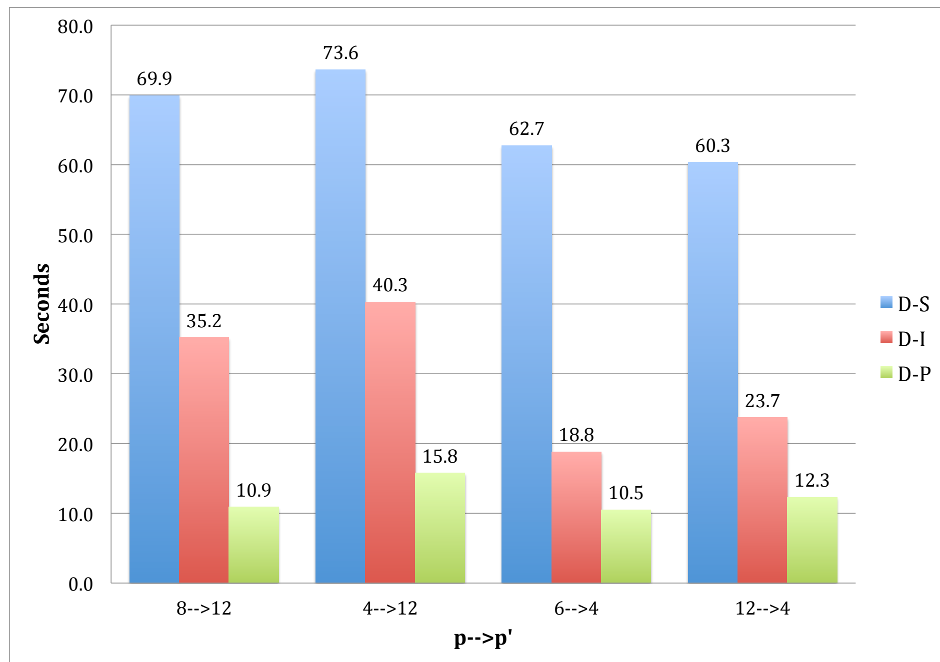
4.4 Additional Observations

Shrinking Search Engine Size is Faster than Enlarging

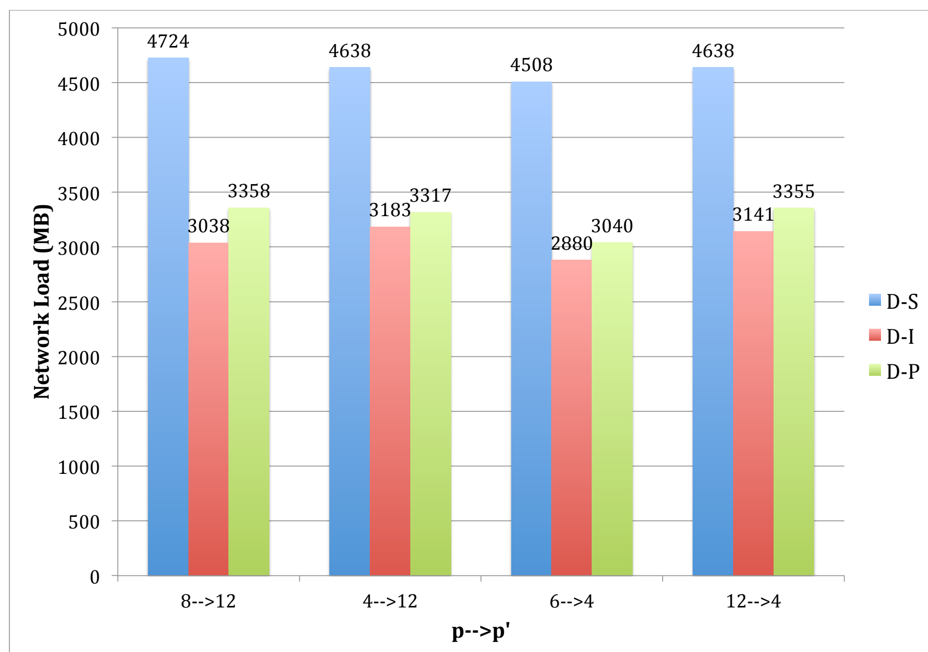
In the experiments, we compare the switch-over time when the search engine size enlarges and shrinks using the three distribution methods. The Switch-over time is faster when the search engine size shrinks from 12 to 4 processors than when it enlarges from 4 to 12 processors. This applies to all the methods shown in Figure 4.2. In D-I and D-S methods, the Index Planner needs to connect with fewer numbers of processors in order to deploy the LC-clusters when shrinking the size of the search engine than it does when it is enlarging. In D-P method, when shrinking the search engine size, the number of processors that send the LC-clusters is more than the number of processors that sends the LC-clusters when it is enlarging.

Ratio between p and p' affect Switching Time

In the experiments, we compare the switch-over time using two different ratios between p and p': 2/3 and 1/3, when the search engine size enlarges or shrinks. The results show that when the engine size enlarges and the ratio was 2/3 between p and p' switch-over requires less time than when ratio was 1/3. However, when the search engine is shrinking, switch-over was a little bit faster when the ratio was 1/3 rather than 2/3. This applies to all the methods shown in Figure 4.2. Moreover, that was not the case with the network load as the ratio between p and p' does not seem to correlate to many differences. The reason, as we expected, was that the number of processors that sends the LC-clusters or the number of processors that needs to be connected by Index planner affected switching time as explained in Subsection 4.4. However, we need to do more in future work to prove



(a) LC-clusters redistribution time



(b) Network load of different distribution methods

Figure 4.2: The results of switch-over performance experiments.

these observations.

4.5 Conclusions

As explained above, the main goal of the experiments is to find an efficient method to redistribute G-groups among processors when switching over from p to p' and to reduce the network load. D-P was found to be the fastest among the three methods we considered. Moreover, D-P and D-I required fewer LC-clusters to be redistributed among the processors than D-S. We can conclude from the experiments that D-P is a suitable solution to ROP and we will use it for redistributing LC-clusters among processors. In particular, we will apply D-P with NGP solutions when measuring their impact on switchover performance.

5 Adapting Distributed Metric Space Index

An adaptive search engine is required to determine a suitable number of processors to use at any given time. When the number of processors is determined, it is necessary to compute the G-groups that will be distributed across the processors. Adapting G-groups to a new number of processors requires either creating new groups from scratch or the altering existing groups.

In this chapter, we propose an algorithm for the New Grouping Problem (NGP) and evaluate its effectiveness using a prototype cloud-based search engine we developed for this purpose as explained in Chapter 1. This chapter is organized as follows. Section 5.1 describes our algorithm for recomputing G-groups. Section 5.2 presents the design and results of our experiments to validate and evaluate our algorithm. Section 5.3 presents experiments to study how different NGP solutions suit different ROP methods.

5.1 Recomputing G-groups

In this chapter, we present an algorithm for NGP and experimental evidence of how different ways of implementing NGP impact the search performance after the switch-over.

5.1.1 Computing H-groups

We compute G-groups from H-groups in the same way as in the KmCol algorithm as explained in Chapter 2. We therefore focus on the computation of H-groups for p' processors from H-groups for p processors. We introduce the following three methods (called **transition types**):

TT-R: Compute H-groups from scratch using K-means, like KmCol.

TT-S: Reuse the H-groups from previous configuration.

TT-A: Increase the number of H-groups using Adjust-H (Algorithm 5.1.1).

Algorithm 5.1.1: Adjust-H(d)(H, new_size)

Tuning Parameters: d — a metric on C
Input:
H — a set of H-groups partitioning C,
new_size — the target number of H-groups (new_size $> |H|$)
Output:
updated H with $|H| =$ new_size

1: H_{sorted} = sort_by_decreasing_size(H)
2: **while** size(H_{sorted}) $\neq$ new_size **loop**
3: largest_group = H_{sorted}.getFirst()
4: new_groups = K-means(d)(largest_group, 2) // split
5: H_{sorted}.insert_sorted(new_groups)
6: H_{sorted}.delete(largest_group)
7: **end loop**
8: return H_{sorted}

Notice that the number of H-groups will never be decreased by TT-A. This is appropriate because, as we show in Section 5.2, reducing the number of H-groups does not improve search performance.

Adjust-H takes as parameters the number new_size ($= p' \cdot w$) and the old H-groups. On line 1, it starts by arranging the H-groups in an ordered collection, with the largest group first. On lines 2–7, the number of H-groups is increased by repeatedly splitting the largest H-group into two using K-means, until there are new_sizes many of them. Thanks to the following observation, we do not need to study the effect of repeated TT-A on search performance:

Proposition 1 (Repeated TT-A is equivalent to a single TT-A)**.** *For any set H and sequence $|H| < p_1 < p_2 < \ldots < p_n$, it holds:*

$$\textit{Adjust-H}\big(\ldots \textit{Adjust-H}(\textit{Adjust-H}(H, p_1), p_2), \ldots, p_n\big) = \textit{Adjust-H}(H, p_n)$$

Proof. A repeated execution of Adjust-H results in successive executions of the loop that forms the algorithm. There are no commands to change the H-groups between the successive executions of the loop. Thus the result of the repeated loop executions is the same as running the loop only once with new_size set to the final value p_n. □

To pursue our goal to speed up switchovers while keeping a good search performance, we will test the search performance implications of the three transition types TT-R, TT-S and TT-A. Based on preliminary observations, we formed the following hypotheses:

NGP-H1 The time it takes to compute H-groups grows significantly with the number of these H-groups.

NGP-H2 Increasing the number of H-groups does not reduce search performance. Equivalently, when reducing p, TT-S does not lead to a worse search performance than TT-R.

NGP-H3 Computing a number of H-groups and then splitting them up using TT-A does not impair search performance when compared to computing the same number of H-groups directly using TT-R.

We provide experimental evidence supporting these hypotheses in Section 5.2.

Using these hypotheses, on the assumption that they are correct, we propose the algorithm Regroup (Algorithm 5.1.2) to decide which of the three transition types to use.

The algorithm parameters can be tuned using the w_{min} and w_{init}. TT-R uses w_{init} to compute H-groups from scratch, while w_{min} is used by TT-A to re-compute H-groups. Due to hypothesis NGP-H2, the values of these tuning parameters do not significantly affect search performance. We therefore use the fairly low values $w_{\text{init}} = 2$ and $w_{\text{min}} = 1.5$ in our experiments in order to reduce the time it takes to compute the H-groups. At the beginning, if a new Q is provided, it is necessary to update the metric d_Q and re-compute the H-groups from scratch (TT-R, lines 2 and 3). If the number of H-groups is smaller than $p' * w_{\text{min}}$, the number of H-groups is increased (TT-A, line 5). If there is no change in Q and $p > p'$, then H is reused (TT-S). Finally, on line 7, new G-groups are computed from the H-groups, using Group-Balanced, an algorithm borrowed from Km-Col (see Section 2.4.1).

Algorithm 5.1.2: Regroup($w_{\text{init}}, w_{\min}$)($p', H, d_Q, Q$)

Tuning Parameters: w_{init}, $w_{\min} \geq 1$
Input:
p' — new number of processors,
H — a set of H-groups partitioning C (optional, needed if Q absent),
d_Q — a metric on C (optional, needed if Q absent),
Q — sample set of queries (optional, needed if H absent)
Output:
G — a partition of C with $|G| = p'$, updated H and d_Q

```
1: if Q is provided then
2:    d_Q := Query-Vector-Metric(C, Q)
3:    H := K-means(d_Q)(p' * w_init, C) // TT-R
4: elseif |H| < p' * w_min then
5:    H := Adjust-H(d_Q)(H, p' * w_min) // TT-A
6: end // TT-S: the if block not executed
7: G := Group-Balanced(H, p')
8: return G, H, d_Q
```

5.2 Experimental Evidence Supporting Hypotheses

In the experiments, the three transition types are compared in terms of their effect on *search performance* and the time it takes to compute H-groups for the new number of processors (a component of *switch-over performance*). The performance is influenced by the following parameters:

1. **Search engine size evolution (SE):** We consider only a one switchover at a time and write it as $p \to p'$ as explained in Chapter 4.

2. **Dataset (D):** As explained in Chapter 4, a dataset represents the set of objects that needs to be searched. In our experiments, we used a randomly selected set of 1,000,000 objects from the CoPhIR Dataset. Each object comprises 282 floating-point number coordinates.

3. **Sample queries (Q):** As explained in Section 2.5, the set defines the metric d_Q which is used to partition LC-clusters into H-groups. In our experiments, we used as Q a randomly selected set of 1,000 objects from the CoPhIR Dataset.

4. **Query profile (QP):** Query profile simulates how users send queries to the search engine. It is determined by a sequence of queries and the timing when each query occurs. In our experiments, we use the Realistic Query Set as explained in Section 2.9. We fire the queries at a constant query rate. This rate is not a parameter of

the experiment because it is determined automatically in the process of measuring maximum throughput as described below.

Search performance is measured using *maximum throughput*. This is defined as follows: The current output throughput (queries/s) of a search engine is the rate at which answers to queries are sent to clients. This is equal to the input throughput, i.e. the rate at which the queries are arriving, except when the queries are accumulating inside the engine. Maximum throughput is the highest output throughput achieved when flooding the input with queries. We have observed that the network stack efficiently facilitates the queuing of queries until the engine is able to accept them.

As we explained in 2.6.1, EC2 produces unstable throughput caused by virtualization. We have observed that from different runs that the maximum query throughput is unstable when we run the same number of processors at different times. Thus we propose the following approach to measure the maximum query throughput in the search engine. We use the same types of Amazon EC2 instances (as explained in chapter 4) for the search engine nodes (Fig. 2.6).

In each experiment, we used the following steps to obtain sufficiently reliable throughput measurements despite significant performance fluctuations of the Amazon cloud platform:

- Conduct two speed tests: an initial and a final test. The two tests are identical. Each test comprises 4 repetitions of a fixed task based on distributed searching.
- If the speed variation of throughput within these 4 repetitions is over 5%, the cloud is not considered sufficiently stable.
- Also if the initial and final speed measurements differ by over 2%, the cloud is not considered sufficiently stable.
- The average of the speed measurements in the initial and final tests is used to calibrate the maximum throughput measurements obtained in the experiment to account for longer-term variations in the cloud performance.

When the stability tests failed repeatedly, we relaxed the thresholds and if the initial and final speed measurements differed by over 2% we took the average of the measurements obtained from 3 repetitions of the experiment. This happened in approximately 75% of our experiments.

We observed that in many experiments, the throughput fluctuates at the beginning and then stabilises. To discount the initial instability, we run each search experiment

as a sequence of blocks of 100 queries and we waited until there were four consecutive blocks with a performance variation of the throughput below 30%. We discounted the preceding blocks that had a higher variance. We computed the 95% confidence intervals for throughput experiments as per the following steps:

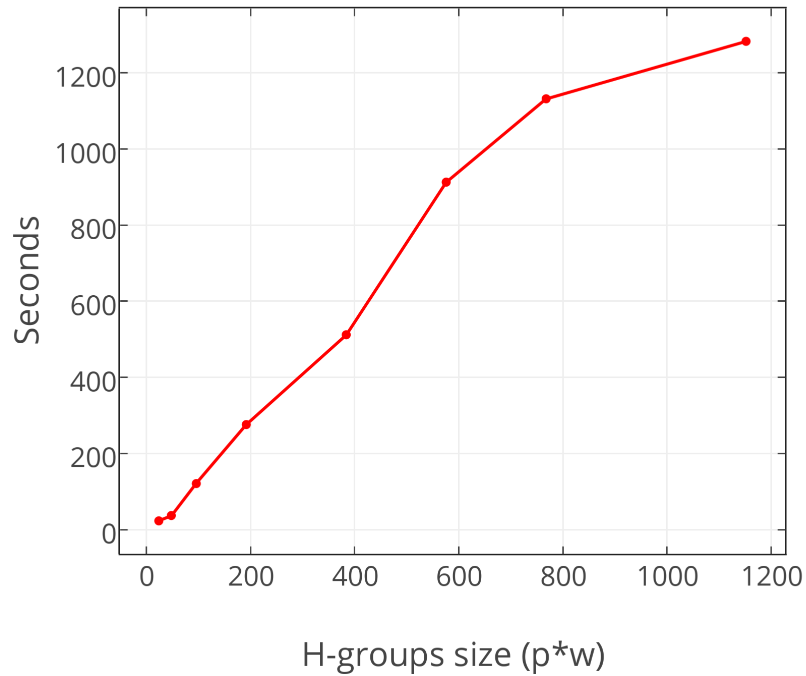
- based on three runs of an experiment (each accepted by relaxed stability criteria):
  - compute mean and margin for the throughput of each run (h_1, h_2, h_3 the 3 throughput measurements) where margin is $\sigma = 2\sqrt{\frac{\sum_{k=1}^{n}(x_k-\mu)^2}{n}}$ and $L = \mu + -$ margin.
  - calibrated measurements and speed margin.
    - $hL_i = h_i * sstd/sL_i$.
    - sL_i = key formula applied to results of the 8 speed tests.
    - $sstd$ = standard speed test result, used for calibration only.
- compute mean and margin of the three margins, hLa = mean of hL_1, hL_2, hL_3.
- compute margin of the centres of three margins hL_i, hm = margin of centres of hL_1, hL_2, hL_3.
- compute final throughput range $hLam = hLa + -hm$ where hm accounts for variations due to non-determinism and hLa accounts for variations due to Amazon cloud.

Moreover, we sorted the result of each query to get only 30 objects for each query. The full code for our experimental search engine and the experiments described in this section are available on http://duck.aston.ac.uk/ngp.

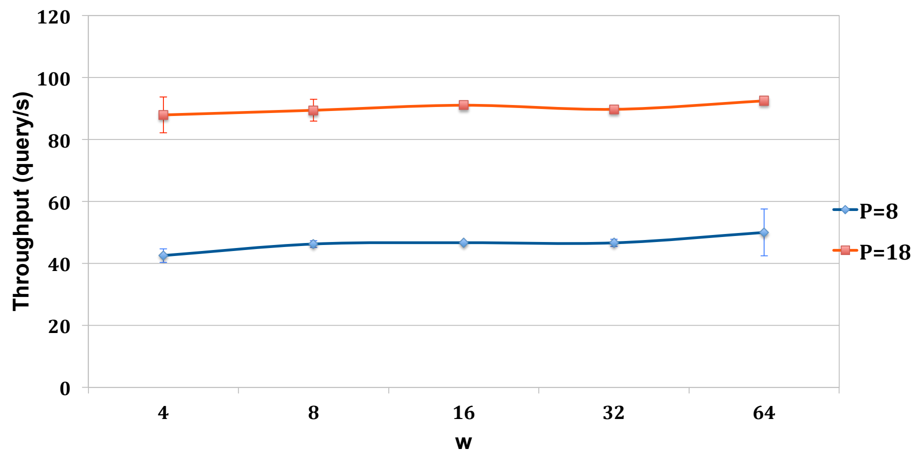
5.2.1 The Number of H-groups

Experiment E1. To test hypothesis H1, we computed different numbers of H-groups and observed how the computation time grows with size while the remaining parameters are fixed. The results shown in the Figure 5.1(a) confirm hypothesis H1.

Experiment E2. In a similar setup as experiment E1, we checked whether the extra computation time spent creating more H-groups translates to improved search performance, in contradiction to hypothesis H2. We have done this for $p = 8$ and $p = 18$ and the same values of w as for E1. The results of $E2$ in Fig. 5.1(b) show that the throughput is not significantly affected by w, confirming H2.



(a) TT-R computation time grows



W	4	8	16	32	64
	42.56	46.28	46.70	46.66	50.02
p=8	(40.35-44.76)	(45.17-47.40)	(46.45-46.96)	(45.51-47.81)	(42.48-57.56)
	87.95	89.45	91.07	89.75	92.48
p=18	(82.19-93.70	(85.94-92.95)	(90.56-91.58)	(89.72-89.79)	(91.16-93.81)

(b) Throughput is not significantly affected

Figure 5.1: Impact of increasing the number of H-groups ($= w * p$) on performance.

5.2.2 Search Performance of TT-S

Experiments E3 and E4. Reusing H-groups for $p' < p$ (TT-S) is much faster than re-computing H-groups (TT-R). The alternative phrasing of hypothesis H2 states that this speed up does not come at a cost to the search performance. Here we report on experiments that confirm hypothesis H2 in the alternative phrasing: The same switchover $p \to p'$ is performed using TT-R and independently using TT-S and the resulting search performance is measured.

These two experiments differ in the set of switchovers considered as follows:

- E3 varies p' and fixes the ratio p/p'.
- E4 varies the ratio p/p' and fixed p'.

The results of these experiments shown in Figures 5.2(a) and 5.2(b) support H2: TT-S does not lead to worse search performance as compared to TT-R when switching over to a smaller number of processors , in fact the opposite seems to be true.

5.2.3 Comparing TT-A and TT-R

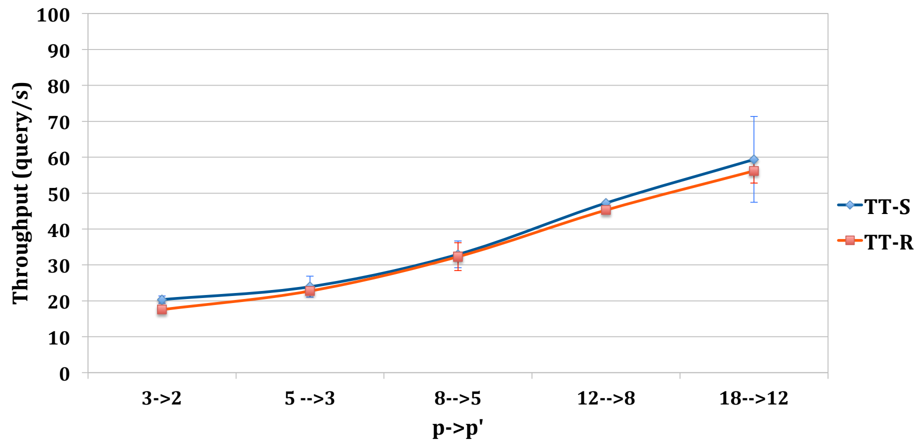
Experiments E5 and E6. In this section, we test hypothesis H3 by comparing the results of experiments that measure the search performance after computing H-groups using TT-R and TT-A. Moreover, we capture the computation time of the transitions to measure the speed-up of TT-A over TT-R.

As with E3 and E4, the experiments differ in the set of switchovers considered as follows:

- E5 varies the ratio p/p' and fixed p'.
- E6 varies p' and fixes the ratio p/p'.

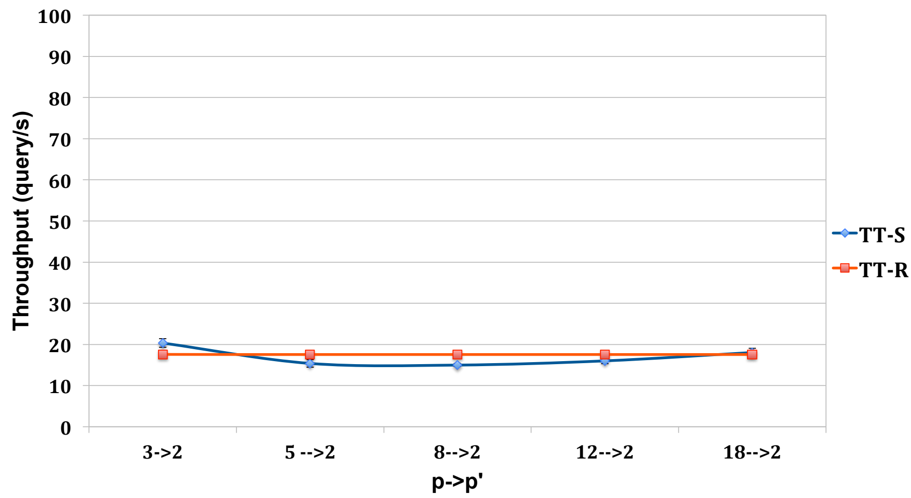
The results of experiments $E5$ and $E6$ in Figures. 5.3(a) and 5.3(b) support hypothesis H3, namely they show that the maximum throughputs after TT-A is similar to, sometimes even better than the maximum throughput after TT-R. Plots in Figures. 5.4(a) and 5.4(b) show that in this context the speed-up of TT-A versus TT-R is 2–10 times.

In the next section, we applied the ROP methods with TT-A and TT-S transition types to measure switchover performance.



p-->p'	3-->2	5-->3	8-->5	12-->8	18-->12
TT-S	20.34	23.93	32.94	47.24	59.37
	(19.46-24.33)	(21.01-26.86)	(29.23-36.64)	(46.95-47.52)	(47.42-71.32)
TT-R	17.55	22.73	32.29	45.27	56.19
	(16.54-18.56)	(21.75-23.72)	(28.43-36.16)	(44.93-45.60)	(52.78-59.59)

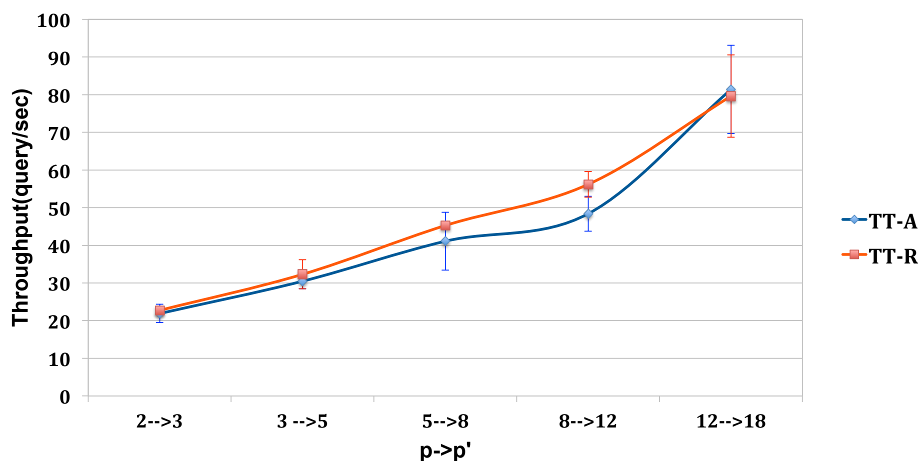
(a)



p-->p'	3-->2	5-->2	8-->2	12-->2	18-->2
TT-S	20.34	15.39	14.98	16.01	18.04
	(19.46-24.33)	(14.46-16.31)	(14.61-15.36)	(15.31-16.71)	(17.03-19.05)
TT-R	17.55	17.55	17.55	17.55	17.55
	(16.54-18.56)	(16.54-18.56)	(16.54-18.56)	(16.54-18.56)	(16.54-18.56)

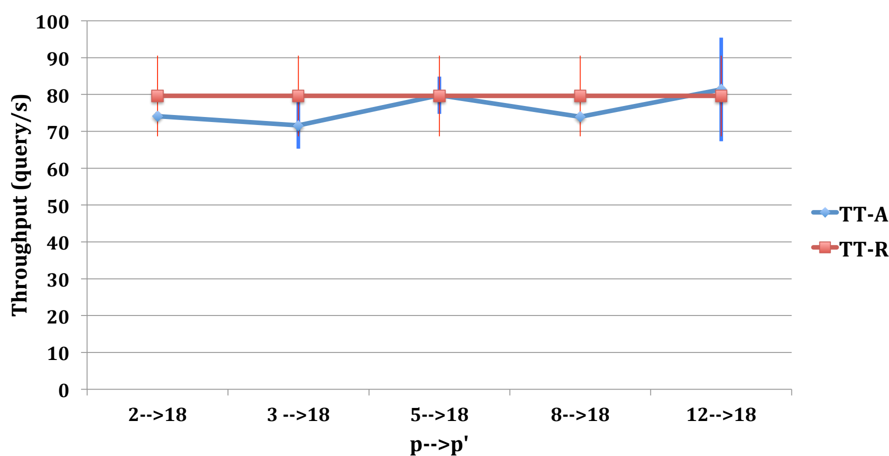
(b)

Figure 5.2: TT-S and TT-R produce similar throughput, measured separately for increasing p' (E3) and increasing p/p' (E4).



p-->p'	2-->3	3-->5	5-->8	8-->12	12-->18
TT-A	21.89 (19.46-24.33)	30.47 (28.48-32.46)	41.09 (33.42-48.77)	48.39 (43.75-53.03)	81.39 (69.70-93.08)
TT-R	22.73 (21.75-23.72)	32.29 (28.43-36.16)	45.27 (44.93-45.60)	56.19 (52.78-59.59)	79.62 (68.67-90.57)

(a)



p-->p'	2-->3	3-->5	5-->8	8-->12	12-->18
TT-A	74.14 (73.076-75.20)	71.63 (65.30-77.95)	79.78 (74.74-84.83)	73.95 (73.26-74.64)	81.39 (67.32-95.46)
TT-R	79.62 (68.67-90.57)	79.62 (68.67-90.57)	79.62 (68.67-90.57)	79.62 (68.67-90.57)	79.62 (68.67-90.57)

(b)

Figure 5.3: TT-A and TT-R lead to a similar maximum throughput after switchovers with various p' and with various ratios.

5.3 Switch-over Performance

In Chapter 4, we test ran the three ROP methods with TT-R (KmCol). The conclusion of previous experiments show that D-P is faster than D-I and D-S methods. In this

section, we aim to compare the switch-over performance of the ROP methods when the new groups have been computed using TT-A and TT-S instead of TT-R. The result of these experiments will show if changing the transition type has made any change of switch-over performance.

As we mentioned in Chapter 4, switch-over performance is measured using the time from the point when G' groups have started deploy onto $\mathcal{P}'$ until the point when the new processors becomes active and network load. Also we have mentioned that grouping data and switch-over will be done in the background of processors during the searching.

5.3.1 Experiments

We run similar experiments with same ratios and number of processors using TT-A and TT-S transition types as shown in Table 5.1. However, TT-A transition types are used mainly when the size of engine grows.

Table 5.1: Experiments of switch-over performance using TT-A and TT-S

Code	Distributed method	$p \rightarrow p'$	Switching type	Transition type
$E1$	D-S,D-I,D-P	$4 \rightarrow 12, 8 \rightarrow 12$	Increase	TT-A, TT-R, TT-S
$E2$	D-S,D-I,D-P	$6 \rightarrow 4, 12 \rightarrow 4$	Decrease	TT-S, TT-R

5.3.2 Results of Experiments

The experiments show that D-P is the fastest in all three transition types. Moreover, switch over using TT-S is faster than TT-A when enlarging the search engine size and using D-I and D-P methods. However, switch-over time of TT-S was almost similar to TT-R when the search engine shrank. This result confirms the result shown in the previous chapter that D-P method is a fastest redistributed method. Figures 5.5 and 5.6 show the switch-over performance and network load respectively. Figure 5.5 show the time of switch-over steps in Section (3.2) as follows:

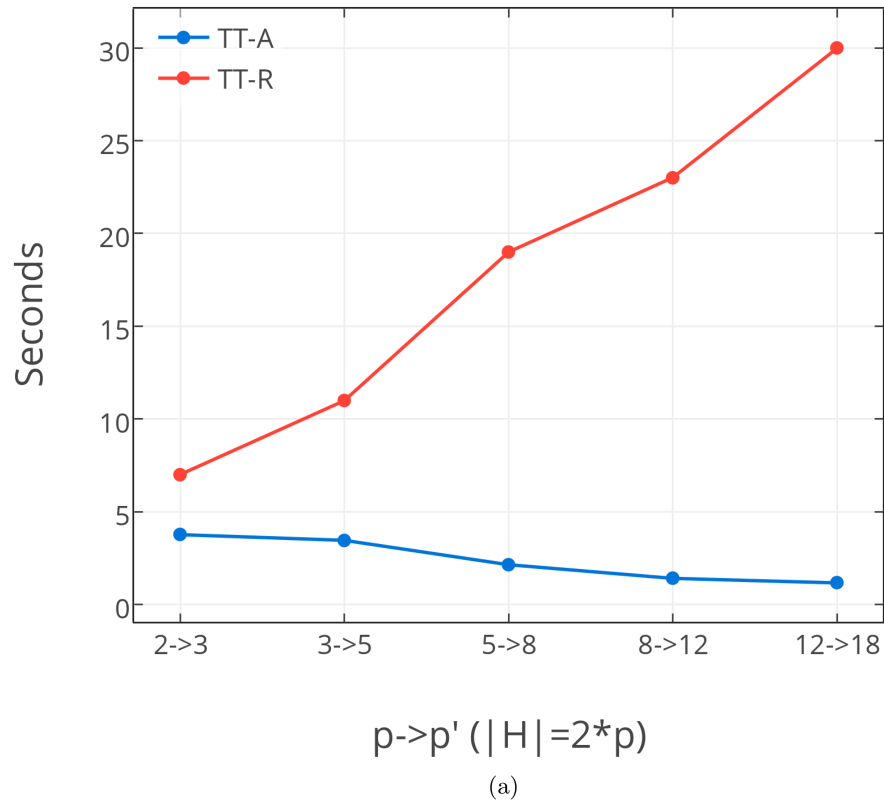
- Time of step 2 when run one of transition types (Under the line).
- Time of step 3 when use the ROP solution (D-P) (above the line).

Moreover, the network load of all transition types show that D-I and D-P less than D-S because less number of LC-clusters send to new processors, as explained in the previous chapter.

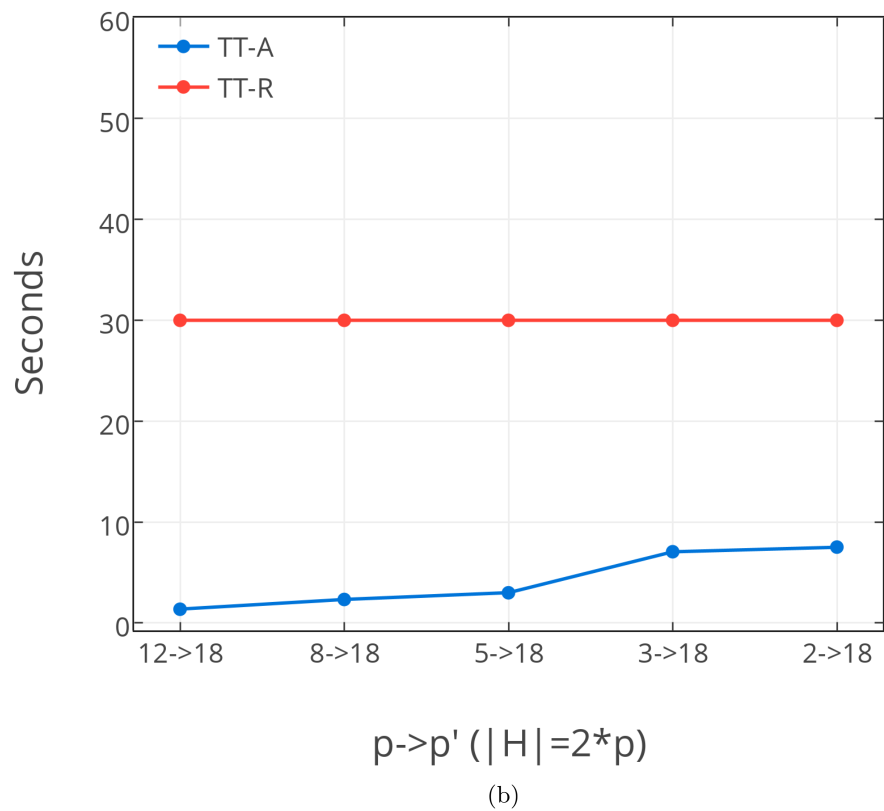
5.4 Conclusions

We have proposed a new algorithm for planning an incremental regrouping of a metric-space search index when a search engine is switched over to a different size. This algorithm is inspired by the results of a set of experiments we conducted. These experiments also indicate that our algorithm facilitates 2–10 times faster switchover planning and leads to a similar search performance when compared with computing the index from scratch.

In this chapter, we studied only the re-computation of the metric-space index when the search engine changes size. In the next chapter, we study the remaining aspects of an adaptive search engine, such as determining when and how to change the engine size while keeping the engine responsive.

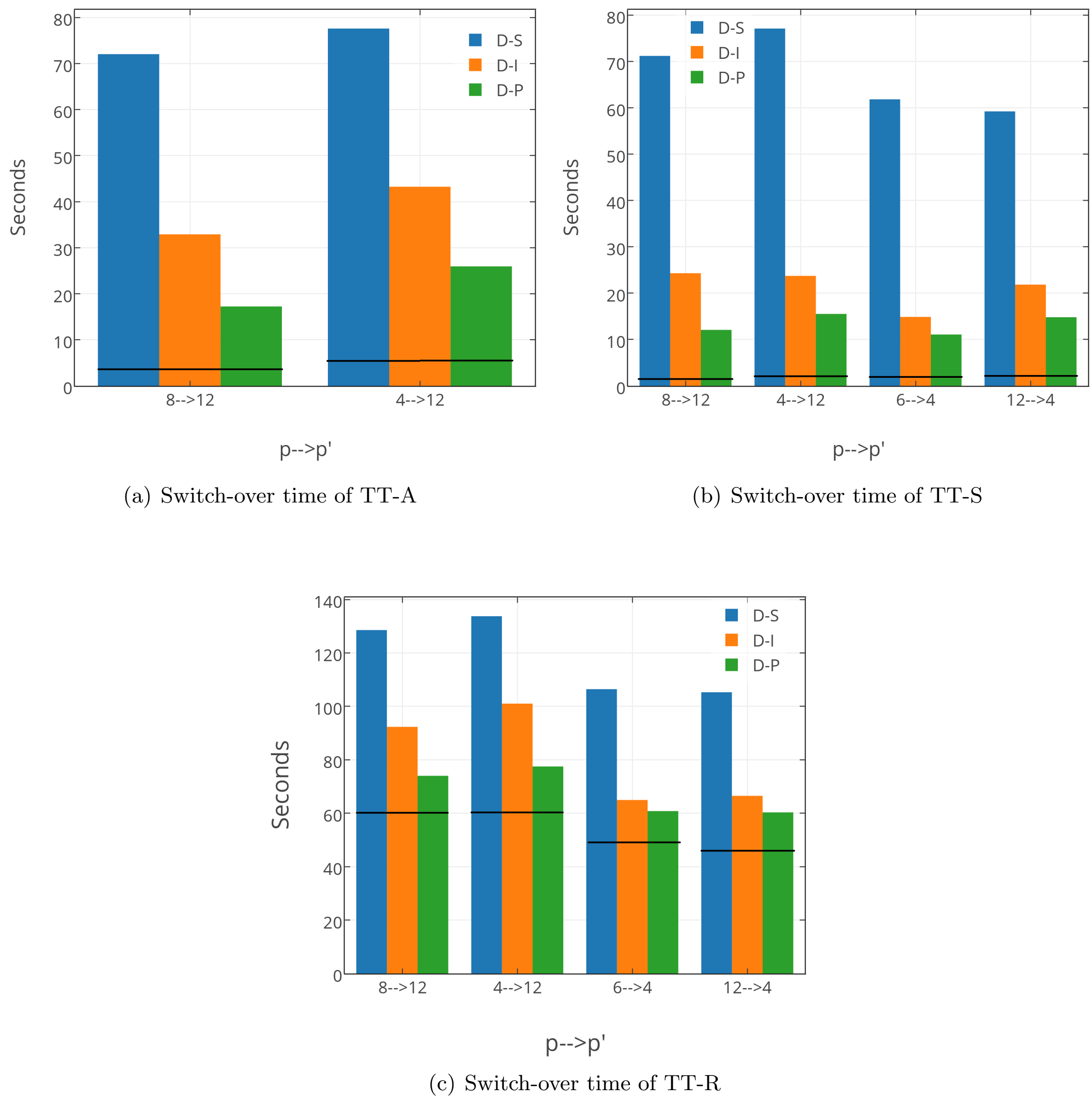


(a)



(b)

Figure 5.4: TT-A is faster than TT-R in switchovers with with various p' and various ratios.

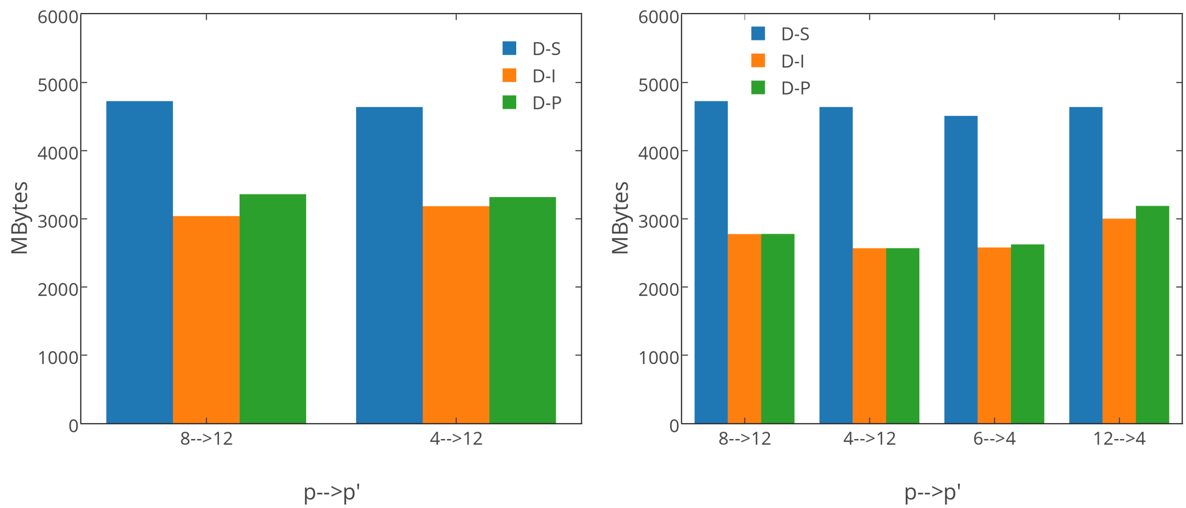


(a) Switch-over time of TT-A

(b) Switch-over time of TT-S

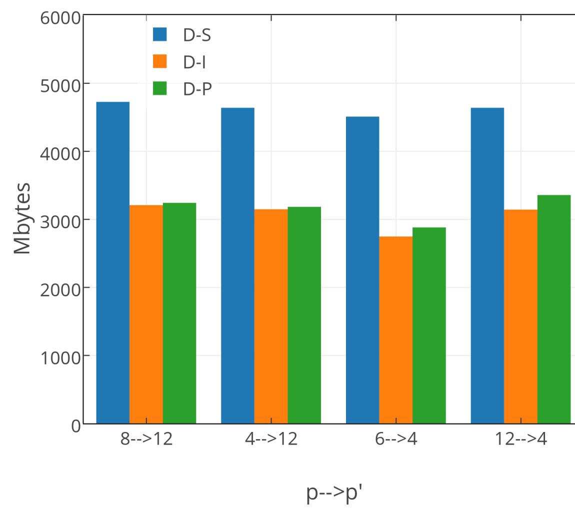
(c) Switch-over time of TT-R

Figure 5.5: TT-S and TT-A produce similar result of TT-R that show D-P is the faster redistributed method. The black line in the figures show time of step2 of switch-over when run one of transition types (Under the line) and time of step 3 when use the ROP solution (D-P) (above the line).



(a) Network load of TT-A

(b) Network load of TT-S



(c) Network load of TT-R

Figure 5.6: TT-S and TT-A produce network load like TT-R that show D-P and D-I have less load than D-S.

6 Determining Number of Processors

We now turn our attention to the problem of Continually determining the Number of Processors (CNP). In this chapter, we focus on determining the size of the search engine and, in particular, deciding when the search engine should change size. Therefore, we first recall CNP and describe in more detail our CNP solution (Section 6.1). Then, we describe our SDMP solution and recall that the SDMP solution is based on solutions to CNP, NGP and ROP (Section 6.2). Finally, we present our experiments design to evaluate our CNP and SDMP solutions and analyse the results (Section 6.3).

6.1 CNP

As explained in Chapter 3, adapting the search engine size with varying workloads means switching over to a different number of active processors p' in response to a higher or lower query workload. To predict p', the CNP solution uses a feedback controller. Feedback controller needs the recent overall CPU load $\mathcal{L}_T(P)$ for the set of currently active processors in the search engine. The feedback controller used to predict p' for the next period of time in the search engine depends on various parameters, as we will explain in Subsection 6.1.1.

6.1.1 Feedback Controller of Simulator

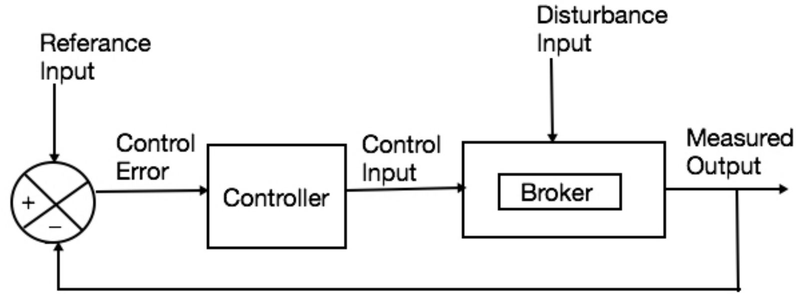


Figure 6.1: Feedback controller of search engine simulator

In this section, we propose a feedback controller of a search engine simulator that uses Determine-Processor-Number (Algorithm 6.1.1) to determine the number of processors when the query workload changes. We adapt elements of the feedback controller concept from [47, 48, 49, 51, 52, 53] which is illustrated in Figure 6.1 and explained in Section 2.7. These elements are used as different parameters in the CNP solution. In our feedback control system, we determine the number of processors of the next period of time by using the following elements:

Controller elements		
1	Control Input: $\mapsto$	number of processors (p')
2	Reference Input: $\mapsto$	maximum average load ($maxL_\phi$) minimum average load ($minL_\phi$) *optimal-load*
3	Disturbance input: $\mapsto$	rate of incoming queries
4	Measured output over recent time: $\mapsto$	overall CPU utilization average response time worst response time

Table 6.1: Elements of Feedback Controller of Search engine simulator

When the workload changes rate of incoming queries, ie Disturbance input, the control system measures the change of load , the overall load of all processors and the average response time. Then, the system control transforms the measured output so that it could be compared with the reference input. Control error will compare the difference between the reference input and the measured output. After that based on current and past values of control error the Controller computes values of the control input (new number of processors p') needed to achieve the reference inputs.

6.1.2 CNP Solution

Algorithm 6.1.1: Determine-Processor-Number($\mathcal{P}$)

Reference Input:
$maxL_\phi$ — maximum average loads of all processors,
$minL_\phi$ — minimum average loads of all processors,
optimal-load — target average load when the query workload decreasing, other.

Tuning Parameters:
lookahead-time — is a ratio used to compute the lookahead time from the monitoring duration d.
n — number of times checking the overall load of all processors.

Input: $\mathcal{P}$ — the set of currently active processors
Output: p' — new number of processors

1: (L, L_Δ) = Get-Variations($\mathcal{P}, n$)
2: **if** $L_\Delta > 0$ **then**
3: p' = increase-p($L_\Delta, L,$ *lookahead-time*$, maxL_\phi$)
4: **end if**
5: **else-if** $L_\Delta < 0$ **then**
6: p' = decrease-p($L_\Delta, L,$ *optimal-load*$, maxL_\phi, minL_\phi$)
7: **end if**
8: **end if**

Algorithm 6.1.2: Get-Variations($\mathcal{P}, n$)

Input:
$\mathcal{P}$ — set of processors to monitor
n — number of measurements to make, $n \geq 3$.

Output:
$L[]$ — recent overall load of all processors during period of time.
L_Δ — a variation of overall load of all processors during period of time.

1: **for** $i = 1 \ldots n$
2: $\mathcal{A}[i]$ = Measure-Processors-Load($\mathcal{P}$)
3: $L[i]$ = Sum$\{\mathcal{A}[i][P] \mid P \in \mathcal{P}\}$
4: **end for**
5: First_Average = Average$\{L[1], L[2], L[3]\}$
6: Second_Average = Average$\{L[n-2], L[n-1], L[n]\}$
7: L_Δ = First_Average − Second_Average

Algorithm 6.1.3: Measure-Processors-Load($\mathcal{P}$)

Tuning Parameters:
repeat_number — number of checks > 0,
checking_interval — delay between checks.

Input: $\mathcal{P}$ — set of processors to monitor.
Output: $\mathcal{A}[]$ — recent load of each processor $P \in \mathcal{P}$

1: **repeat every** checking_interval **for** repeat_number **times**
2: **for every** $P \in \mathcal{P}$ **loop**
3: current_load_sum[P] += Get-Current-Load(P)
4: **end for**
5: **end repeat**
6: **for every** $P \in \mathcal{P}$ **loop**
7: $\mathcal{A}[P]$ = current_load_sum[P]/repeat_number
8: **end for**

We propose the Determine-Processor-Number Algorithm, This algorithm takes as parameter the current set of processors $\mathcal{P}$ reference input and various tuning parameters. In this algorithm, we use there components of reference input : $maxL_\phi$, $minL_\phi$, *optimal-load*. The *optimal-load* is used consistently to maintain sufficient numbers of processors in the search engine when the query workload is decreasing.

At the beginning, we measure the recent overall load of all processors L and a variation of overall load of all processors L_Δ during a certain period of time by calling the Get-Variation (Algorithm 6.1.2) (Line 1).

if $L_\Delta = 0$ then there is no need to change the number of processors and if $L_\Delta > 0$ that shows the overall load is increasing. We measure the *expected-load* for the next period of time by multiplying the variation L_Δ by *lookahead-time* and then we add the result to the recent load L. We decide the next number of processors by comparing the *expected-load* with $maxL_\phi * p$, an estimate of the maximum attainable load of the current set of processors. The number of processors for the next period of time is computed by dividing the *expected-load* by $maxL_\phi$ as shown in Fig 6.2.

$$\text{increase-p}(L_\Delta, L, \textit{lookahead-time}) = \begin{cases} \lceil \textit{expected-load}/maxL_\phi \rceil & \text{if } \textit{expected-load} > maxL_\phi * p \\ p & \text{otherwise} \end{cases}$$

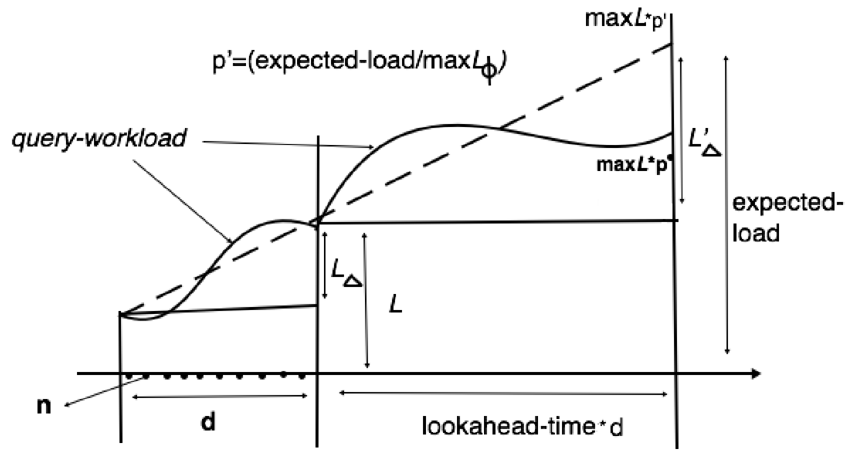


Figure 6.2: Computing p' when query workload is increasing

if $L_\Delta < 0$ then the number of processors are going to decrease. We divide L by the

optimal-load to ensure the search engine has enough processors as shown in Fig 6.3.

$$\text{decrease-p}(L_\Delta, L, \textit{optimal-load}) = \begin{cases} \lceil L/\textit{optimal-load} \rceil & \text{if } L/p < minL_\phi \\ p & \text{otherwise} \end{cases}$$

where $minL_\phi \leq \textit{optimal-load} < maxL_\phi$.

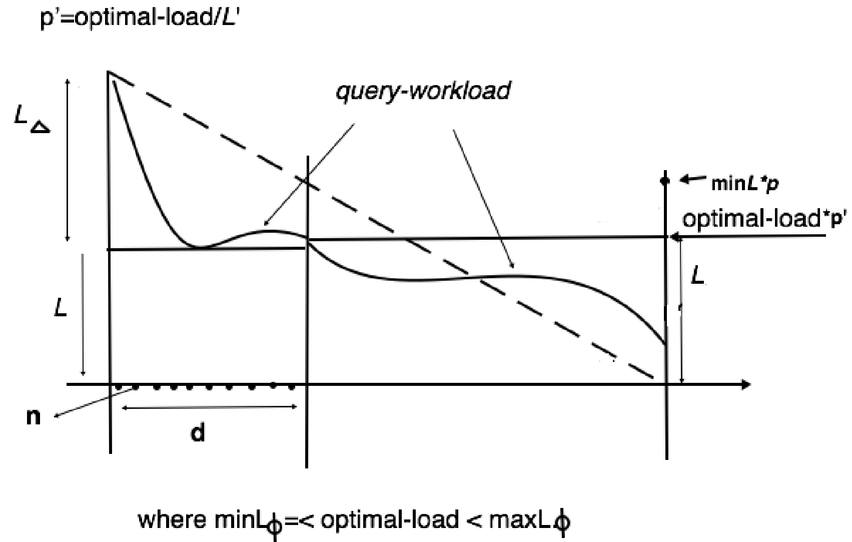


Figure 6.3: Computing p' when query workload is decreasing

The Get-Variation algorithm is used in Algorithm 6.1.1 to measure the overall load of processors and the variation of change in the overall load of processors during a given time period. In this algorithm, we measure the average load of each processor using the Measure-Processors-Load algorithm (Algorithm 6.1.3) and calculate the overall load of all processors $L[]$ for n numbers (lines 1-4). In lines 5-7, we measure the variation of the overall load of processors (L_Δ) by finding the average of the first three load measurements $L[0], L[1], L[2]$ and the average of last three $L[n-2], L[n-1], L[n]$ and then we subtract the two averages.

The Measure-Processors-Load algorithm (Algorithm6.1.3) reads the current CPU load for each processor at each time interval. Then, we calculate an estimate of the average load of each processor over the measurement period given by the parameters: *repeat_number* a number of times to check the current CPU load, *checking_interval* delay between checks. On lines 1-5, we read and sum the CPU load for each processor each time interval. Then, we calculate the average load for each processor and store it in the array $\mathcal{A}[P]$ (lines 7-10).

Algorithm 6.2.1: Adapting($\mathcal{P}, C, d_Q, H$)

Tuning Parameters:
tuning parameters as specified in sub-algorithms
Input:
$\mathcal{P}$ —a set of active processors,
C — LC-clusters,
d_Q: a metric on C,
H — a set of H-groups partitioning C.

1: $i = 1$
2: $p_i = 2, \mathcal{P}_i = 1$
3: **repeat forever**
4: p' =Determine-Processor-Number($\mathcal{P}_i$)
5: **if** ($p' \neq p_i$) **then**
6: (G_{i+1}, H_{i+1}) = Regroup(p', C, d_Q, H_i)
7: Re-allocation-steps = D-P(G_{i+1}, H_{i+1})
8: $\mathcal{P}_{i+1}$ = Switch-over (Re-allocation-steps)
9: $p_{i+1} = p'$
10: $i = i + 1$
11: **end if**
12: **end repeat**

6.2 SDMP

A solution to SDMP will be obtained from the solutions to CNP, GNP and ROP as follows:

CNP determines the new number of processors when p_t changes and NGP is applied to calculate the new groups from the current G-groups. Then, the result of NGP G-groups is passed onto ROP to compute the allocation of these groups onto the new set of processors giving a list of steps that lead from an old to a new allocation. When these steps are executed, the new groups are allocated. After new groups have been allocated onto the new processors, they will continue running the search until the value of p_t changes again as determined by the CNP solution.

For this purpose, we propose the Adapting algorithm (Algorithm 6.2.1) to adapt the search engine size with the varying workloads. The algorithm switches over to the new search engine size p' after computing and deploying G' groups onto many processors. On lines 1-10, the search engine runs a loop to adapt search engine size as follows:

On line 3, determine p' by using CNP solution i.e. Algorithm 6.1.1. If the search engine needs to change size (line 4) then apply NGP solution to compute G'-groups and then the ROP solution to deploy G'-groups onto p' (lines 5-6). In line 7, the search engine switches over to p' processors.

We considered two scenario parameters when designing a solution for SDMP: firstly, a frequency of change of sample query set Q and secondly, a frequency of the need to

change p due to query load fluctuations. These two parameters influence which one of the three transition types (TT-R,TT-S,TT-A) would be optimal when the search engine size changes. We focus only on the second parameter and assume in our scenario that Q does not change and the query load undergoes 2-4 major changes per day.

6.3 Experiments Design and Results

In the experiments, we aim to evaluate the Adapting algorithm as well as the Determine-Processor-Number algorithm (CNP solution). The Adapting algorithm is measured in terms of the effect on the search performance and the ability to change the search engine size with query workload variation. The performance is influenced by the same parameters as in Chapters 4 and 5 above.

A simulation of a realistic query workload is required to evaluate our solution. The Sogou log [1] (Figure 6.4) provides a sample search engine query load distribution by hours in the Sogou search engine [2]. This load has been used to determine the query rate evolution in our search client simulation. We converted the Sogou log from hours to minutes so that the simulation does not take too long. We fire the queries depending on the varying query rate in the Sogou log. However, the Sogou log gives a query rate appropriate to a single time zone. To evaluate our solution in simulated multiple time zones, we modified this query rate as explained in the next section.

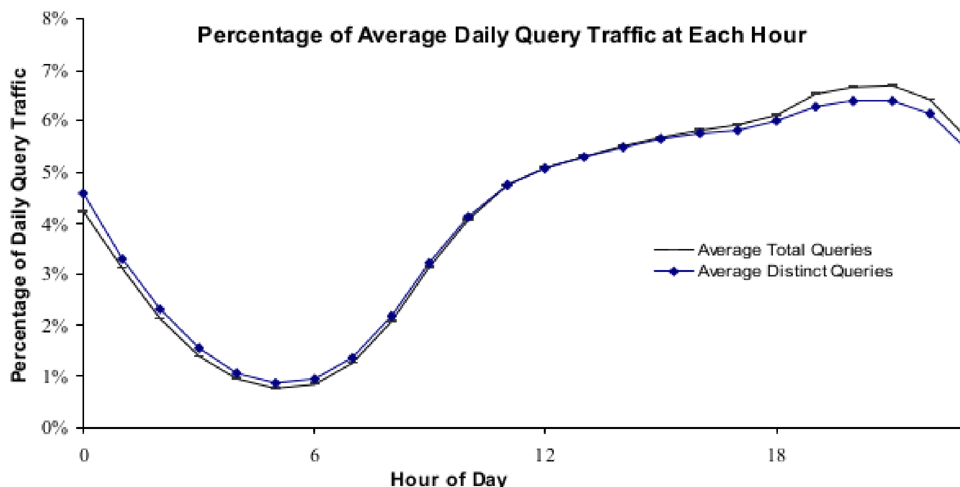


Figure 6.4: Query rate in the Sogou log (adapted from [43]).

[1] http://www.sogou.com/labs/dl/q-e.html
[2] http://www.sogou.com/

6.3.1 Search Scenarios

The variation of query workload change depend on different reasons such as different time zones among countries and special events in the world. For this purpose, we aim to evaluate the Adapting algorithm for three different search scenarios as follows:

Scenario 1 Single time zone over 50 hours.

Scenario 2 Multiple time zones with weights over 50 hours.

Scenario 3 Simulating a big event causing a temporary a sharp increase and then sharp decrease soon afterwards.

We run three type of experiments for these three scenarios to evaluate our solutions as follows:

E1 The parameters are set to ensure the search engine adapts its size as soon as the workload changes. In the experiments the values of the parameters are as follows:

- $maxL_{\phi}$ =0.80: the maximum average load per processor.
- $minL_{\phi}$ =0.50: the minimum average load per processor.
- *optimal-load* =0.7 : target average load when the query workload is decreasing to make sure that the search engine has enough processors for any changes.
- *lookahead-time*=5: is a ratio used to compute the lookahead time from the monitoring duration d.
- $n = 3$: number of measurements to make.
- *repeat_number*=3 : number of checks > 0.
- *checking_interval*= 1 second : delay between checks.

In the other experiments, we changed the value of *lookahead-time* parameter to check how increasing or decreasing the expected time for the next switch-over affects the search results. As the query workload fluctuated, we found that increasing this parameter will affect the result of searching if the search engine does not have enough processors. However, decreasing the value of *lookahead-time* more than necessary that might keep the search engine doing many switch-overs in a short period of time which would affect the search engine's performance.

E2 As explained above in this experiments we reduce the value of the *lookahead-time* parameter to check how the search engine responds. In this experiment, we decrease *lookahead-time* to 1 while keeping the other parameters as in E1.

E3 In this experiment, we increase *lookahead-time* to 10 and keep other parameters as in E1.

The results of these experiments for each scenario are as follows:

Scenario 1

In the experiment E1, the search engine starts with 2 processors and changes depending on the change in the workload as shown in the Figure 6.5(a). The result of this experiment shows that the search engine adapts its size dependent workload change without any significant delay. However, the experiment E2 shows some delay in response to workload change as we reduce *lookahead-time* to 1 as shown in the Figure 6.5(b). In E3, search engine increases p too quickly and too much in response to the increase of the workload as shown in Figure 6.6.

Scenario 2

In Scenario 2, we simulate two time zones as explained above. The time difference between the two zones was 6 hours. The result of experiment E1 in Scenario 2 show that the search engine adapts its size when the workload changes as shown in Figure 6.7(a). In experiment E2 there is some delay in the search engine adaptation as shown in Figure 6.7(b).

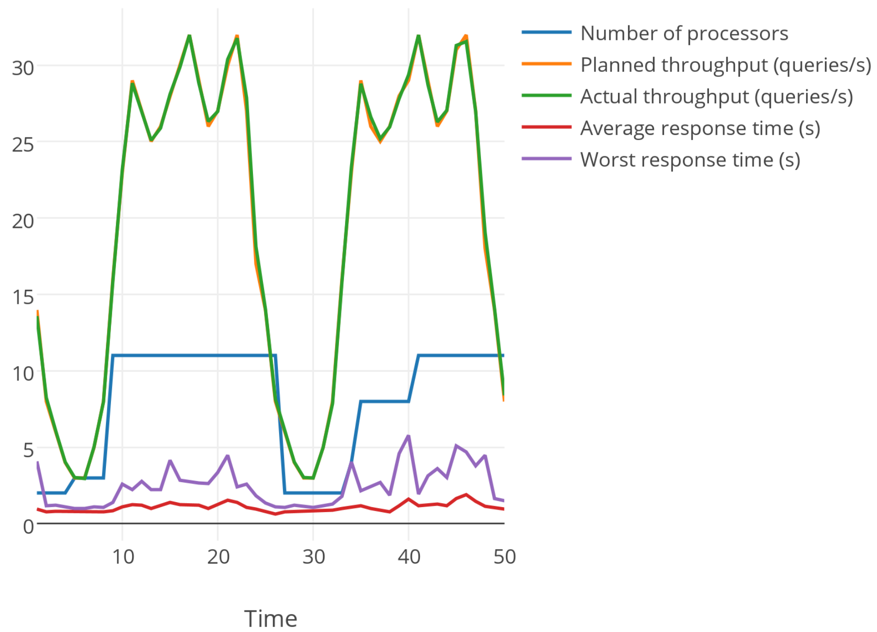
Scenario 3

As explained above, in Scenario 3, we aim to evaluate the Adapting algorithm when the workload sharply increases and then decreases. The results of E1 show that there is some delay in response to workload changes that make the average response time become worse during sharp increases (as shown in Figure 6.8). After increasing the time of checking the overall load, the search engine delayed at the beginning to adapt its size; thus it seems the search engine could not manage to adapt its size in response to workload changes. Scenario 3 tests show that the algorithm does not cope well with very sharp changes; this could be improved by taking the recent response times into account when deciding whether and by how much to increase P.

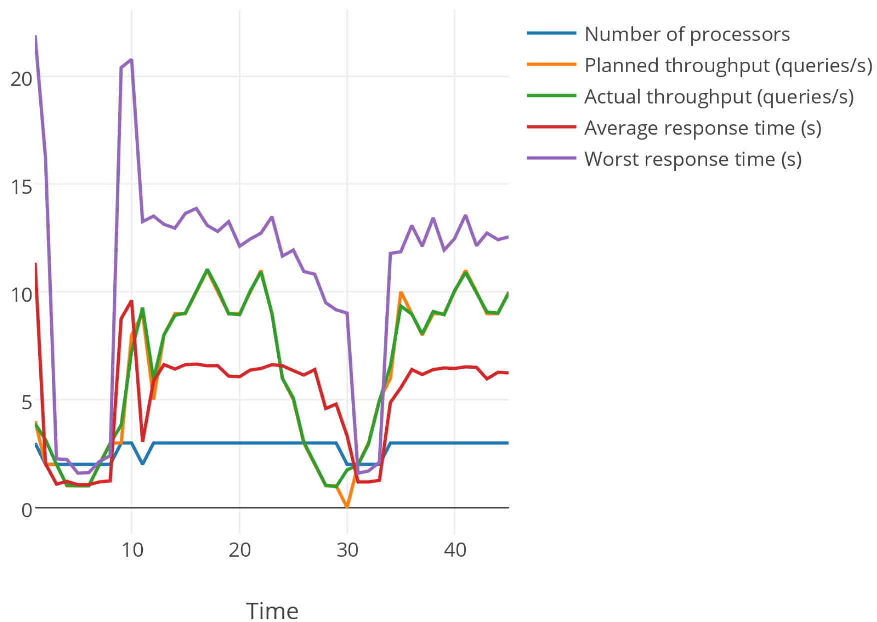
6.4 Conclusions

The experiments show that the Adapting algorithm has the ability to adapt the search engine size when the workload changes except when the change is very sharp. However, we plan in the future to improve our algorithm to be able to adapt the search engine when

the change is very sharp by monitoring daily/weekly patterns as well as special events and so predict the load based on such data.



(a) E1



(b) E2

Figure 6.5: In **Scenario 1**, search engine show good response time when the value of *lookahead-time* was suitable in E1, however, a small *lookahead-time* lead to high response times in E2.

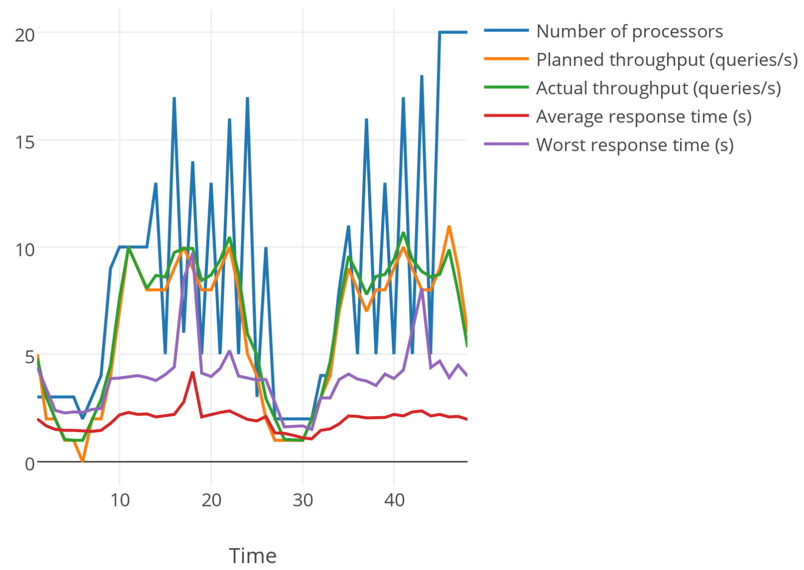
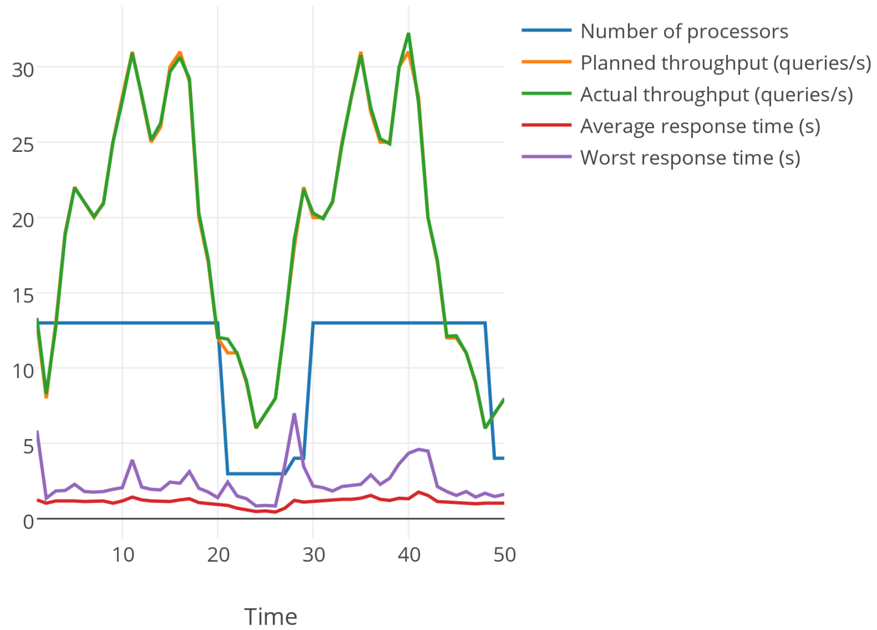
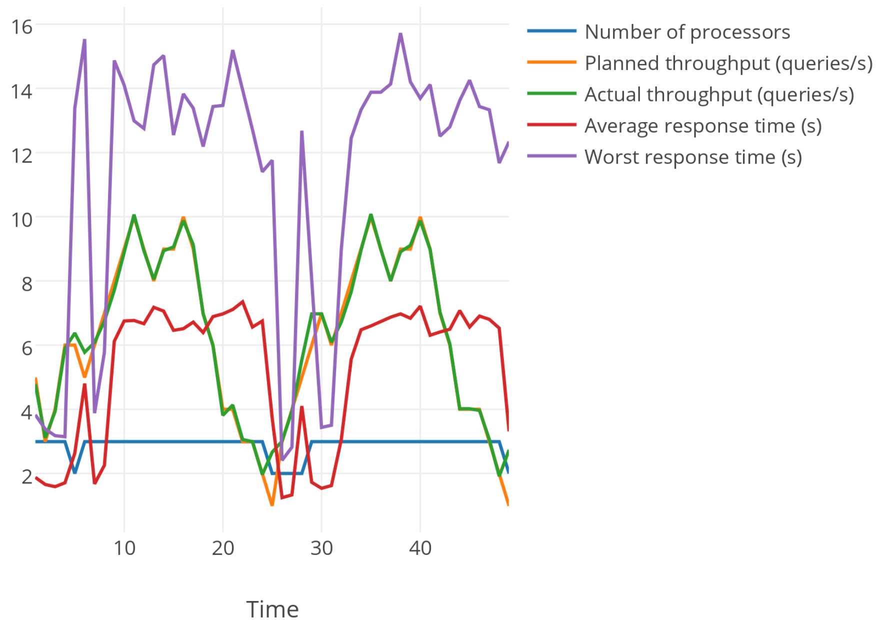


Figure 6.6: The search engine increases size too quickly when *lookahead-time* is too large in E3 **Scenario 1**.



(a) E1



(b) E2

Figure 6.7: In **Scenario 2**, search engine show good response time when the value of *lookahead-time* was suitable in E1, however, a small *lookahead-time* lead to high response times in E2.

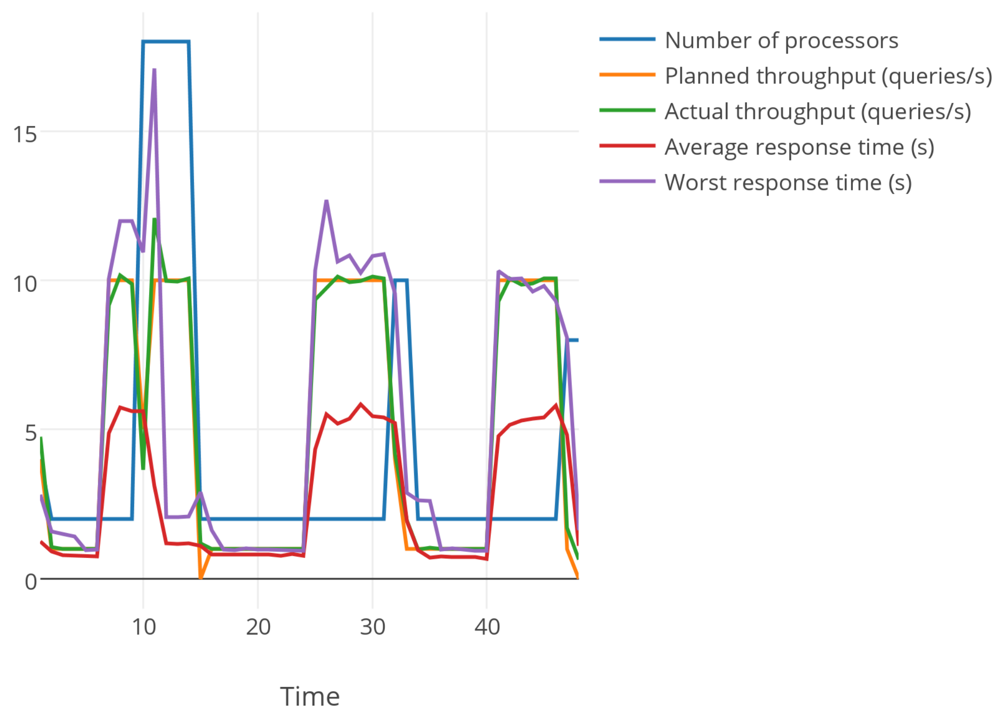


Figure 6.8: The algorithm does not cope well with very sharp changes of the workload in **Scenario 3**.

7 Validation of Search Engine Simulator

In this chapter, we will present the validation that has been done for our simulator. In Section 7.1, we explain the validation of simulator and type of validations that have been considered. Then in Section 7.2, we present all the experiments of the validations and the results of the experiments.

7.1 Validation of Simulator

Validation is to indicate that experiments have been properly conducted for our simulator properly with error free implementations. We validate our simulator in two ways:

- Compare our implementation with a third party implementation in terms of a performance metric for the same dataset and queries.
- Provide evidence that the parallel realization of the simulator is able to scale its performance with the number of processors.

We discuss both validation points below.

7.2 Experiments and Results

The experiments depend on the following parameters:

1. **Dataset** (D): The dataset as explained in Section 5.2, represents the set of objects that needs to be searched. We used a selected set of 1,000,000 objects from the CoPhIR Dataset. Each object comprises 282 floating-point number co-ordinates and each LC-cluster has 100 objects.

2. **Query profile** (QP)

   In our experiments, we use as queries 200,000 that following more realistic user behaviour as explained in Section 2.9. We fire the queries at a constant query rate.

7.2.1 Performance Metric Experiments

We compared our research implementation with a third party implementation from Yahoo! research group. The goal was to check our implementation of the LC index against a third party implementation in terms a performance metric such as total number of distance evaluations.

Veronica Gil-Costa from Yahoo! Research Latin America Labs provided a third party implementation to validate our simulator. The code she used to building the LC index is available in http://www.sisap.org/metricspaceslibrary.html/. She created the LC-index for our dataset and executed the same queries as we did and counted the number of distance evaluations.

In the first experiments, we compute the total number of distance evaluation of queries. We searched the queries using the third party LC index and compared the total number of distance evaluation of queries with the results of our implementation. The idea is that, as explained in 2.3.2 during the processing of a search query (q, r), if the first cluster center is c and its radius is r_c, start by measuring the distance $d(q, c)$ and adding the center c of the cluster (c, r_c, I) to the result set if $d(q, c) \leq r$. Then, we scan exhaustively I from the cluster (c, r_c, I) only if the query ball (q, r) intersects with the cluster (c, r_c, I). However, if the query ball (q, r) is totally and strictly contained in the cluster (c, r_c, I), we only consider this cluster and ignore others because all the points inside the query ball have been inserted into I. For example, if the query ball (q, r) is totally and strictly contained in the cluster (c, r_c, I), the number of distance evaluations will be 101 only. First we evaluated the distance between the query and the cluster radius R_c then we evaluated the distance between the query and the 100 objects inside the cluster. However, sometimes

we compared the query with more than one cluster if the query intersected many clusters.

In the experiments, we used two sets of queries where each set has 100,000 objects. We run each set with both LC index implementations. Both implementations have given exactly the same number of distance evaluations for the same dataset and queries. We show a small sample of the results for the two query sets in both implementations in Figure 7.1.

Query Set.1	
Third party Implementation result	Our simulator result
IDq 0 nDists 101	**0 ndist 101**
IDq 1 nDists 101	**1 ndist 101**
IDq 2 nDists 101	**2 ndist 101**
IDq 3 nDists 101	**3 ndist 101**
IDq 4 nDists 101	**4 ndist 101**
IDq 5 nDists 101	**5 ndist 101**
IDq 6 nDists 101	**6 ndist 101**
IDq 7 nDists 101	**7 ndist 101**
IDq 8 nDists 101	**8 ndist 101**
IDq 9 nDists 304	**9 ndist 304**
IDq 10 nDists 304	**10 ndist 304**
IDq 11 nDists 304	**11 ndist 304**
IDq 12 nDists 312	**12 ndist 312**
IDq 13 nDists 312	**13 ndist 312**
IDq 14 nDists 305	**14 ndist 305**
IDq 15 nDists 305	**15 ndist 305**
IDq 16 nDists 305	**16 ndist 305**
IDq 17 nDists 412	**17 ndist 412**

Query Set.2	
Third party Implementation result	Our simulator result
IDq 0 nDists 101	**0 ndist 101**
IDq 1 nDists 101	**1 ndist 101**
IDq 2 nDists 101	**2 ndist 101**
IDq 3 nDists 101	**3 ndist 101**
IDq 4 nDists 101	**4 ndist 101**
IDq 5 nDists 101	**5 ndist 101**
IDq 6 nDists 304	**6 ndist 304**
IDq 7 nDists 304	**7 ndist 304**
IDq 8 nDists 304	**8 ndist 304**
IDq 9 nDists 312	**9 ndist 312**
IDq 10 nDists 312	**10 ndist 312**
IDq 11 nDists 305	**11 ndist 305**
IDq 12 nDists 305	**12 ndist 305**
IDq 13 nDists 305	**13 ndist 305**
IDq 14 nDists 412	**14 ndist 412**
IDq 15 nDists 412	**15 ndist 412**
IDq 16 nDists 512	**16 ndist 512**
IDq 17 nDists 512	**17 ndist 512**
IDq 18 nDists 512	**18 ndist 512**
IDq 19 nDists 101	**19 ndist 101**

Figure 7.1: Comparing the number of distance evaluation performed for the queries Sets using our simulator and a third party implementation

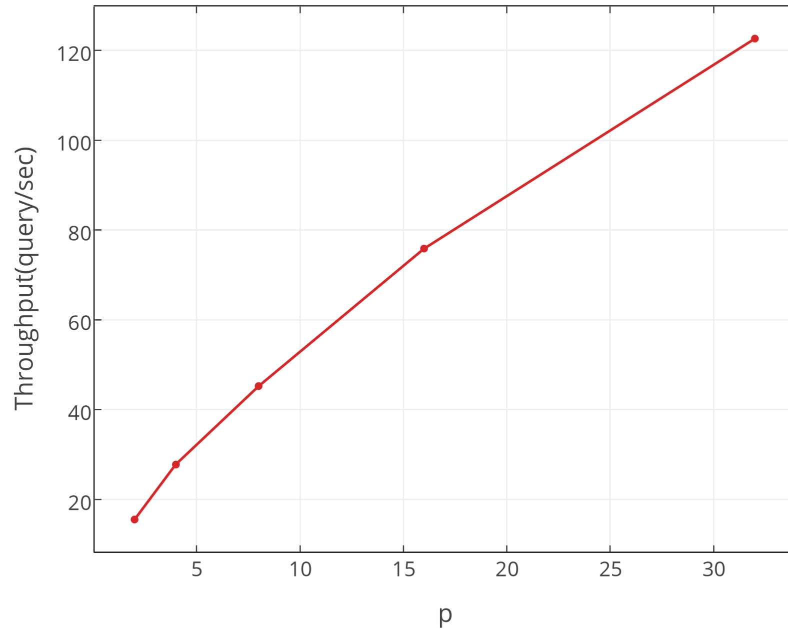
7.2.2 The Level of Scalability Experiments

Another validation is to provide some evidence that the parallel realization of the simulator is able to scale its performance with the number of processors. We provide the evidence by showing the level of scalability in Amazon EC2 and in a 32-core shared-memory com-

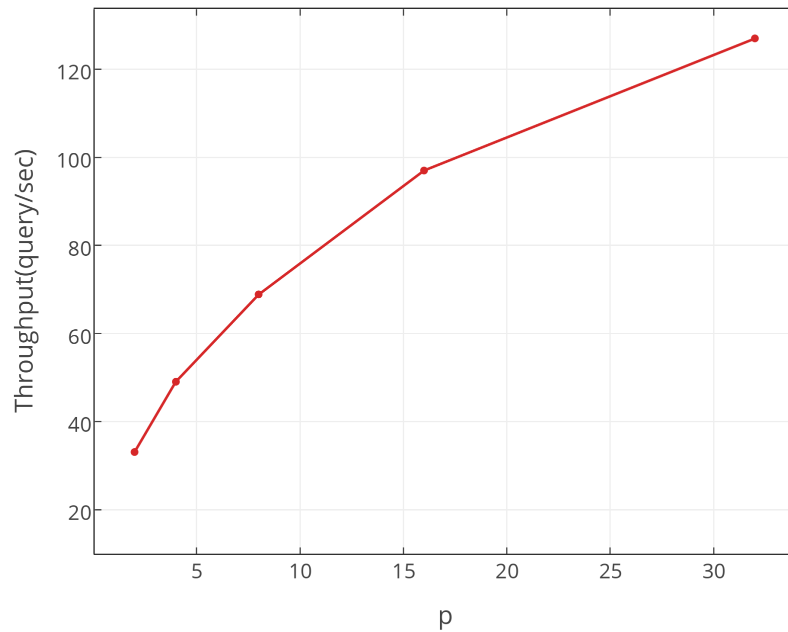
puter. Scalability refers to the simulator's ability to accommodate rising resource demand gracefully, without a noticeable loss in the quality of services. To measure scalability, we need to measure how the throughput increases when the search engine size increases.

In the second experiment, we compared the level of scalability of the simulator on Amazon EC2 and on a 32-core shared-memory computer. In both experiments, we used the sequence 2,4,8,16,32 for numbers of processors and measured the maximum throughput using the method described in Section 5.2. In the Amazon EC2 experiments, we used the same instances as stated in section 2.6 and we measured the confidence intervals for each experiment of the different number of processors using the same method explained in section 5.2. In the 32-core server, we deployed each processor on a separate core and all the processors shared the same memory.

The results in Figure 7.2 show that the parallel realization of the simulator is able to scale its performance nearly linearly with the number of processors. When we run the simulator in the 32-core shared-memory computer, the results show that the simulator is strongly non-linear due the bottleneck of shared memory. Therefore, in EC2 the simulator scales are much better than the 32-core shared-memory computer.



(a) Amazon EC2



(b) 32-core shared-memory computer

Figure 7.2: Scalability of the search engine simulator.

8 Conclusion

8.1 Summary of Achievements

In this thesis, we focus on the problem of how to self adapt parallel metric-space search engine size for dynamic query loads (SDMP). The SDMP solution was achieved by adapting the search engine by way of re-evaluating its load and, when appropriate, by switchover from p active processors to a different number of active processors.

We break SDMP out into three sub-problems. The three sub-problems are as follows:

- Continually determining Number of Processors (CNP): In the light of the changes in the query workload in the search engine, there is a problem of determining the ideal number of processors p active at any given time to use in the search engine.
- New Grouping Problem (NGP): When a change in number of processors is determined, we have to decide the groups G that will be distributed across the processors.
- Regrouping Order Problem (ROP): When we have new groups G we will need to plan how to redistribute the LC-clusters into the groups G onto processors while keeping the engine responsive, while minimising the switchover time and the incurred network load.

Solutions to these sub-problems can produce key answers and insights for solving the overall problem. The research achievements regarding the solutions for these sub-problem are as follows:

- ROP: We considered the following three methods to re-distribute the LC clusters according to the given G-groups G' to a new set of processors $\mathcal{P}'$:

  D-S Distributed from Scratch; The Index Planner sends all the LC-clusters to the assigned processors to replace their old LC-clusters,if any.

  D-I Distributed using Index Planner; The Index planner sends only the missing LC-clusters to each processor. The processors will remove any old LC-clusters that are no longer assigned to them.

  D-P Distributed using Processors; The Index Planner pre-computes the instructions for the processors and the processors swap LC-clusters among themselves according to the new plan.

  D-P was found to be the fastest among the three methods we considered. Moreover, D-P and D-I required less LC-clusters to be redistributed among the processors than D-S. The experiments show that D-P is 50% faster than D-S and 10% to 25% faster than D-I. In addition, D-P and D-I reduces the network load to around 30% less than D-S load.

- NGP: We compute G-groups from H-groups in the same way as in the Km-Col algorithm as explained in Chapter 2. We therefore focus on the computation of H-groups for p' processors from H-groups for p processors. We introduce the following three methods (called **transition types**):

  TT-R: Compute H-groups from scratch using K-means, like Km-Col.

  TT-S: Reuse the H-groups from previous configuration.

  TT-A: Increase the number of H-groups using Adjust-H.

  We have proposed a new algorithm for planning an incremental regrouping of a metric-space search index when a search engine is switched over to a different size. This algorithm is inspired by the results of a set of experiments we conducted. These experiments also indicate that our algorithm facilitates 2–10 times faster switchover planning and leads to a similar search performance when compared with computing the index from scratch.

- CNP: We have proposed the adapting algorithm for determining the new size of the search engine and the algorithm evaluated by using three search scenarios as follows:

  Scenario 1 Single time zone over 50 hours.

  Scenario 2 Multiple time zones with weights over 50 hours.

  Scenario 3 simulating a big event causing a temporary sharp increase and then a sharp decrease soon afterwards.

  The search engine updates the number of processors whenever the workload changes using the Adapting algorithm with all search scenarios.

  Moreover, we use CoPhIR to run our experiments which is the largest publicly available collection of high-quality images metadata. Moreover, we used 1 million objects from CoPHIR to evaluate the result from our research.

8.1.1 Strengths of Research

Amazon EC2 cloud: This research has been conducted using a real cloud i.e. Amazon EC2. One contribution is in using Amazon EC2 to manage search engine size and our methods to mitigate its unstable performance. Amazon EC2 provided suitable services at an acceptable cost.

CoPhIR dataset: One of the difficulties which we faced at the beginning of this research was to find a real dataset for our experiments. We downloaded different dataset from the internet to run the experiments and test our results. However these were less realistic than CoPHIR.

This research provides the Adapting algorithm to adapt the search engine size using cloud. We have not come across any research using cloud to manage search engine size.

8.2 Future Work

This research can be expanded in different directions as follows:

1. Limitation of Amazon EC2 cloud: various issues with EC2 instances affected the performance for the search engine. Moreover, Amazon EC2 is changing the types of instances from time to time and some of these instance take a fairly long time to be ready for use. We plan to use a more suitable environment to deploy and evaluate our solutions.

2. In this research we did not focus on potential changes of the sample query set Q when running SDMP experiments. We plan to consider this parameter in the future.

3. We created a method to determine a new search engine size using the most recent load. However, we would like to monitor daily/weekly patterns as well as special events and predict the load based on such data. When the search engine uses more parameters it will be able to manage the size depending on the result which repeatedly updates the search engine.

4. We plan to consider in the future the load presented to clusters. As we observe from our experiments some LC-clusters have higher load than other LC-clusters. We plan to measure the load of each LC-cluster during search and by regrouping the LC-clusters among processors so LC-clusters that have higher load will be redistributed into different processors. We expect that network load are going to drop down as only few of the LC-clusters move to new addresses during switch-over.

5. We measured CPU utilisation of the processors only to determine number of Processors. We have not yet evaluated the overall CPU utilisation of our method. Therefore, the loads of the Broker and Index Planner have not been measured. We plan to measure the loads of the Broker and Index Planner in the future as part of an overall CPU utilisation study.

6. We used a self-made experimental search engine to evaluate our solution for SDMP. We hope to implement our methods in a real search engine and evaluate them in this context. However, before scaling this work there is some limitations which need to be considered as follows:

   - Dataset size: In our experiments, we used a set of 1,000,000 objects (2.27 GB) from the CoPhIR to evaluate our solution. However, the dataset is much bigger in the real search engine and regrouping huge dataset for big numbers of processors will require a lot of time and which may affect the search engine performance.
   - Cost of EC2 instances: Cost of EC2 instances is dependent on the type of instances in Amazon. Indexing and grouping data for a real search engine will require an instance with big memory and each instance must be ready to hold a big group of LC-clusters, which thus would be more expensive. We need to measure the cost of running real search engines in EC2 instances.

In future work, we hope to consider these points to evaluate our solution in a real search engine.

A Process to Run Proposed Search Engine Simulator

A.1 Amazon EC2 configuration

Running our search engine simulator in Amazon EC2 requires the creation of an account in order to run all the configuration steps as explained below:

1. log into your Account
2. Launch Instances: create and configure your instances in the EC2 by selecting launch in the EC2 Dashboard. Follow the steps below for instance configuration:
   (a) Choose an Amazon Machine Image (AMI): select Amazon linux AMI from the options.
   (b) instance type: Select the type of the instances e.g m1.medium.
   (c) Create a Virtual Private Cloud (VPC).
   (d) A Key Pair: Create a new key pair and assign a name.
   (e) Security Group: configure your virtual firewall.
   (f) Launch Instance: start your instance by clicking launch.

(g) Primary IP Address(public): A primary IP address is used to access to the main instance in your VPC.

3. Connect to your Instance: connect your instance by using EC2 Dashboard or by running a script as explained in Section A.2.

4. Terminate Instances:

   Terminate your instances to prevent additional charges via Dashboard. All the data will be deleted from the instances.

For further details, see: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html

A.2 Shell scripts

In this section, we explain the different shell scripts that we developed to help run our search engine simulator in EC2. We consider as an example the NGP TT-A throughput experiment switching from 8 to 18 processors. We created three types of scripts to run the search engine simulator in Amazon EC2:

- Launch instances script: This script launches all the required Amazon EC2 instances. The script carries out the following steps:

  1. Allocate a private IP address: We allocate a private IP address to each instance e.g privateip0=10.0.0.20 . These private addresses are used to communicate between instances. The main instance has both a public IP address and a private IP address. The public IP address will be allocated by Amazon EC2 and used as the main IP to access the main instance from outside Amazon EC2. The private IP addresses are used to communicate to the other instances.
  2. Type of instance: In this part, we decide types of instance and select the Machine Image (AMI) which has been created during Amazon EC2 configuration e.g

     ```
     runinstance="ec2-run-instances ami-75037602
     --instance-type m3.medium --subnet subnet-2d5f0a46"
     ```

     . These details can used for all the instances and Th user can create different type for each instances

3. Launch the main instance: Launch the main instance and hold its details to associate the main instance to private address e.g

   ```
   {runinstance} --private-ip-address {privateip0}
   ```

   .

4. Launch other instances: Then we launch all the remaining instances and associate each instances to it private IP-Address e.g

   ```
   {runinstance} --private-ip-address {privateip1}.
   ```

   .

5. Associate public address: Associate the main instance to the public address e.g

   ```
   ec2-associate-address -i instanceid 54.229.175.232
   ```

   .

- Simulator script: This script is used to send the simulator codes to Amazon EC2 instances, execute them and get the result back.

  1. Send the simulator files from local machine to the main instance, e.g

     . The following is the list of simulator files that are sent to the instances:

     - IndexPlanner.jar
     - SearchProc.jar
     - SearchBroker8-18.jar (This depends on the number of processors)
     - Controller.jar (Used only for CNP experiments)
     - ips18.txt
     - myKeyPair.pem
     - TT-A-8-18.sh
     - Dataset1mCHLC100.ser
     - Dataset.txt
     - queries.txt
     - SampleQuery.txt

  2. Run the simulator execution:

     e.g

     ```
     ssh \${options} \${host1} sh NGP-p8-18.sh}
     ```

.

3. Send back the results from the main instance to the local machine

   e.g

```
scp \${options} \${host1}:outputfile.txt \${local1}}
```

   .

4. Stop the simulator e.g

```
ssh -n  \${options} \${host1} kill -9 `ps -ef
| grep java | grep -v grep | awk '{print \$2}'`}
```

   .

- Simulator execution: This script runs the search engine simulator in EC2.

1. Run the Index Planner: Copy all the necessary files in IndexPlanner node before starting Index Planner.
2. Run the Processors: Copy the processor files onto the processors. Then run all the processors and make them ready to accept requests from IndexPlanner or from Broker.
3. Run the Broker: Start run Broker in the main instance e.g SearchBroker8-18.jar.
4. Stop the simulator: When the Broker finishes search it will send a command to kill all jar files in the nodes. e.g

```
ssh -n  \${options} \${host1} kill -9 `ps -ef |
grep java | grep -v grep | awk '{print \$2}'`}
```

   .

- Terminate all instances After the work is complete, terminate all the instance in EC2 to avoid extra cost.

A.3 Search engine simulator

The search engine simulator has three packages holding different classes. In this section, we focus on the main classes in each package. These packages as follows:

- Clustering package:

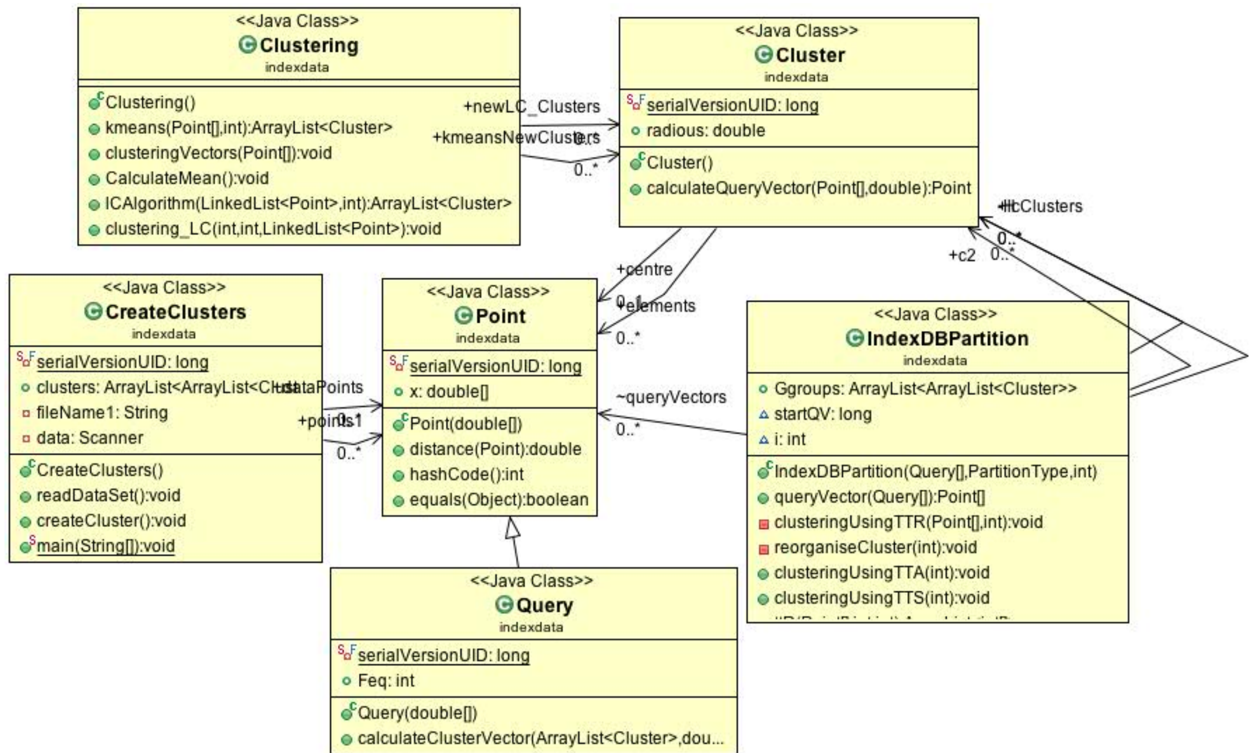


Figure A.1: Clustering package classes

The package to compute G-groups using different transition types. As shown in Figure A.1. This package includes the following classes:

1. Point: Represent a data point object.
2. Cluster: Represent LC-cluster objects each center, with elements set and radius.
3. Query: Represent query objects with a radius and also to calculate query vectors.
4. Clustering: user to returns H-groups computed by k-means and LC-clusters created by LC algorithm.
5. IndexDBPartition: holds all the transition types to compute G-groups.
6. Partition Type: lists different transition types.
7. DistanceComp: to compare distance between data points.
8. PointAndDistanceFromCenter: compute distance of points from center of the LC-clusters.

- Network load package: The network package provides the ability to monitor sockets to compute the network load of all the processors as shown in Figure A.2. This package includes the following classes:

1. MonitoringSocket: calculates the network load of the processors.

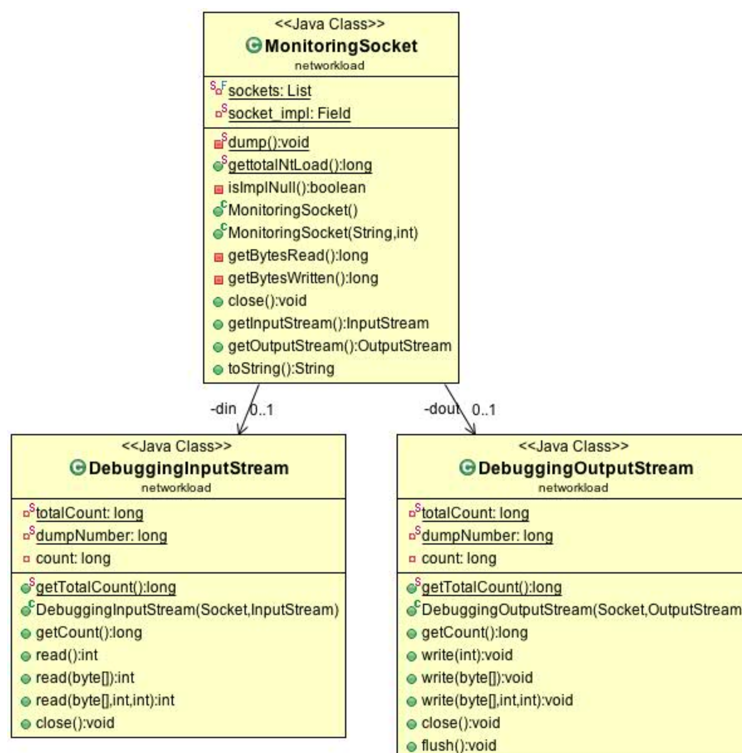


Figure A.2: Network load classes

2. DebuggingInputStream: a substitute for InputStream that counts the bytes being transferred OutputStream.
3. DebuggingOutputStream: a substitute for OutputStream that counts the bytes being received InputStream

Note all the classes in this package have been adopted from http://www.javaspecialists.eu/archive/Issue169.html. We modified them for our research by making MonitoringSocket as superclass for Processor class in the search engine package.

- Search engine package:

The search engine package holds all the classes used for simulating the search engine as shown in A.3.

These classes as follows:

1. IndexPlanner class: This class will be used in Index Planner node in the search engine architecture. It computes LC-clusters and G-groups using classes in Clustering Packet. Then it creates the index plan for these G-groups before distributing the index plan and LC-clusters onto the processors.
2. SearchProc class: This class is present in the Processor node. It works as a ranker and as a processor as explained in Chapter 2.

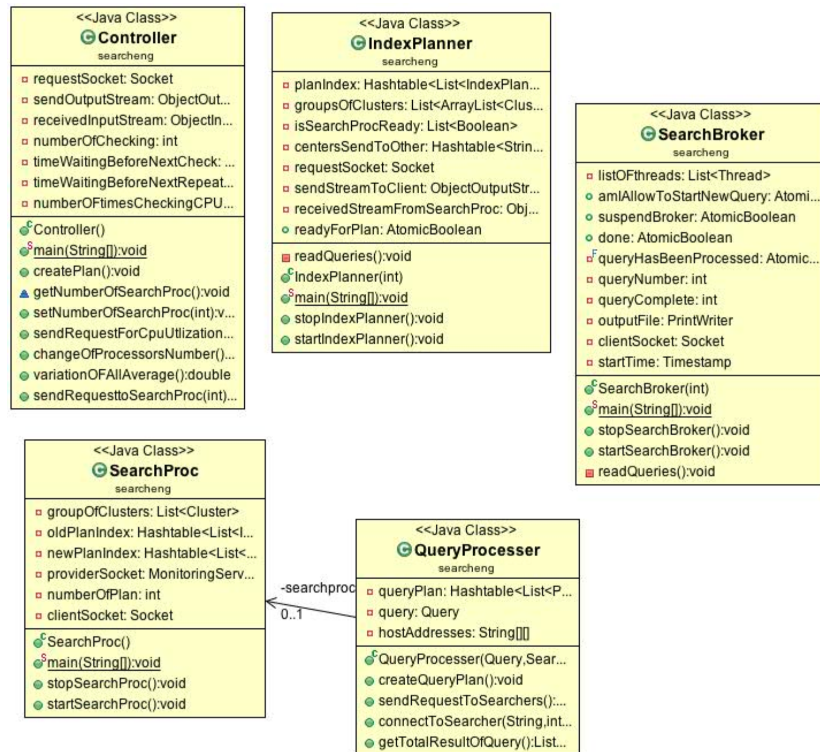


Figure A.3: Search engine package classes

3. Broker class: In this simulator, Broker class sends queries to the rankers and receives the result from rankers as mentioned in Chapter2. This class records the throughput of the search engine and response time.

4. Control class: We use control class for monitoring change in the query workload. Control has all the parameters needed to decide when the search engine changes size.

Bibliography

[1] Jonassen, Simon. Efficient query processing in distributed search engines. PhD thesis, Department of Computer and Information Science, *Norwegian University of Science and Technology*,Trondheim, Norway. 2013.

[2] Hunter Schwarz, How Many Photos Have Been Taken Ever, *BuzzFeed*, September 24, 2012.

[3] A. Jaimes, M. Christel, S. Gilles, R. Sarukkai, and W.-Y. Ma. Multimedia information retrieval: What is it, and why isnt anyone using it? In Proceedings of the ACM SIGMM International Workshop on Multimedia Information Retrieval, pages 38, *Hilton, Singapore*, 2005.

[4] M. Kankanhalli and Y. Rui. Application potential of multimedia information retrieval. *Proceedings of the IEEE*, 96(4):712720, 2008.

[5] Swain, M.J. Searching for multimedia on the World Wide Web. 1999. IEEE International *Conference on Multimedia Computing and Systems*, 1999.

[6] Smeulders, A. W. M., Worring, M., Santini, S., Gupta, A., and Jain, R., Content-based image retrieval at the end of the early years, *IEEE Transactions on Pattern Analysis and Machine Intelligence 22(12)*, 13491380 (2000).

[7] R. van Zwol, S. Ruger, M. Sanderson, and Y. Mass. Multimedia information retrieval: new challenges in audio visual search. SIGIR Forum, 41(2):7782, 2007.

[8] Ibrahim, Amani S., et al. "CloudSec: a security monitoring appliance for Virtual Machines in the IaaS cloud model." Network and System Security (NSS), *2011 5th International Conference on. IEEE*, 2011.

[9] Ostermann, Simon, et al. "A performance analysis of EC2 cloud computing services for scientific computing." Cloud computing. *Springer Berlin Heidelberg*, 2010. 115-131.

[10] Barham, P., Dragovic, B., Fraser, K ., Hand , S., Harris, T .L., Ho, A., Pratt, 1., Warfield, A.: *Xen and the art of virtualization.* In: SOSP. ACM, New York (2003).

[11] Mark Levene: An Introduction to Search Engines and Web Navigation (2. ed.). Wiley 2010: *I-XIX, 1-478.*

[12] E. Chavez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363-1376, 2005.

[13] E. Chavez, G. Navarro, R. Baeza-Yates, and J. Marroquin. Searching in metric spaces. *ACM Computing Surveys*, 3(33):273-321, 2001.

[14] Marin, M., Ferrarotti, F., Gil-Costa, V., (2010), Distributing a metric-space search index onto processors. In: Parallel Processing (ICPP), 39th International Conference on Parallel Processing, San Diego, California, USA. *The Institute of Electrical and Electronics Engineers*, Inc.

[15] V. Gil-Costa, M. Marin, and N. Reyes. Parallel query processing on distributed clustering indexes.*Journal of Discrete Algorithms*, 7:3-17, 2009.

[16] E. Chavez, G. Navarro, R. Baeza-Yates, and J.L. Marroquin. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273-321, September 2001.

[17] G. Navarro. Searching in metric spaces by spatial approximation. *VLDB*, pages 28-46, 2002.

[18] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. *In VLDB*, pages 426-435, 1997.

[19] Novak, D., Batko, M., and Zezula, P. (in press). Metric Index: An efficient and scalable solution for precise and approximate similarity search. *Information Systems.* doi:10.1016/j.is.2010.10.002.

[20] N. R. Brisaboa and O. Pedreira. Spatial selection of sparse pivots for similarity search in metric spaces. *In SOFSEM* 2007, LNCS 4362, pages 434-445, 2007.

[21] Akassh A Mishra and Chinmay Kamat. Article: Migration of Search Engine Process into the Cloud. International Journal of Computer Applications 19(1):19-23, April 2011. Published by *Foundation of Computer Science.*

[22] A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan, Searching the Web, ACM Trans. *Internet Tech.*, 1(1), 2001.

[23] V. Gil-Costa, and M. Marin, "Load Balancing Query Processing in Metric-Space Similarity Search", In 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012), May 13-16, 2012, *Ottawa*, Canada.

[24] V. Gil-Costa and M. Marin, "Approximate Distributed Metric-Space Search", ACM Workshop on Large-Scale and Distributed Information Retrieval (LSDS-IR 2011), *Glasgow* UK, Oct. 2011.

[25] D. Puppin. A search engine architecture based on collection selection. PhD thesis, Department of Informatics, *Pisa University*, Pisa, Italy, Sept. 2007.

[26] D. Puppin, F. Silvestri, and D. Laforenza. Query-driven document partitioning and collection selection. In INFOSCALE 2006, *Hong Kong*, May 30-June 1, 2006. ACM, 2006.

[27] D. Puppin, F. Silvestri, R. Perego, and R. Baeza-Yates. Load balancing and caching for collection selection architectures. In INFOSCALE 2007, *Suzhou*, China, June 6-8, 2007, page 2. ACM, 2007.

[28] Claudine Santos Badue, Ramurti Barbosa, Paulo Golgher, Berthier Ribeiro-Neto, and Nivio Ziviani. Distributed processing of conjunctive queries. In HDIR ?05: Proceedings of the First International Workshop on Heterogeneous and Distributed Information Retrieval (HDIR05),SIGIR 2005, *Salvador*, Bahia, Brazil, 2005.

[29] Fidel Cacheda and Vassilis Plachouras. Performance analysis of distributed architectures to index one terabyte of text. volume 2997, *pages 394-408*, 2004.

[30] Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. Design of a Parallel and Distributed WEB Search Engine. In Proceedings of Parallel Computing (ParCo) 2001 conference. *Imperial College Press*, September 2001.

[31] Kowalski G. Information Retrieval Architecture and Algorithms. 1st. *Springer-Verlag New York*, Inc , 2010.

[32] Papadopoulos, A., Manolopoulos, Y.: Distributed processing of similarity queries. Distributed and Parallel Databases *9(1)*,67-92 (2001).

[33] M. Marin, V. Gil-Costa, and C. Bonacic. A search engine index for multimedia content. In Proc. EuroPar 2008, pages 866?875. *LNCS 5168*, Aug. 2008.

[34] David Novak, Michal Batko, Pavel Zezula: Large-scale similarity data management with distributed Metric Index. Inf. Process. *Manage. 48(5): 855-872*. 2012.

[35] Voorsluys, W., Broberg, J., Buyya, E., (2011), "Cloud computing in a nutshell". In:*Cloud computing: principles and paradigms*, by Buyya, R., Broberg, J., Goscinski, A. John Wiley 'I&' Sons, Inc.

[36] Hosono, S., Kimita, K., Akasaka, T., Hara, T., Shimomura, Y., Arai, T., (2011), Toward Establishing Design Methods for Cloud-Based Business Platforms. In: Hesselbach, J., Herrmann, C., *Functional Thinking for Value Creation. 3rd CIRP International.*

[37] Wang, X., (2010), Clustering in the Cloud: Clustering algorithms to Hadoop Map/Reduce Framework. *Texas State University-San Marcos, Dept. of Computer Science. Technical Reports-Computer Science.* Paper 19. Independent Study Report.

[38] Gil-Costa Veronica, and Mauricio Marin. "Distributed sparse spatial selection indexes." 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing. *IEEE*, 2008.

[39] Marin, M., Gil-Costa, V., Hernandez, C.: Dynamic p2p indexing and search based on compact clustering. *In: SISAP. pp.* 124131 (2009).

[40] Novak, D., Batko, M., Zezula, P.: Metric index: An efficient and scalable solution for precise and approximate similarity search. *Inf. Syst.* 36(4), 721733 (2011).

[41] Doulkeridis, C., Vlachou, A., Kotidis, Y., Vazirgiannis, M.: Peer-to-peer similarity search in metric spaces. *In: VLDB* (2007).

[42] Yuan, Y., Wang, G., Sun, Y.: Efficient peer-to-peer similarity query processing for high-dimensional data. *In: Asia-Pacific Web Conference.* pp. 195201 (2010).

[43] Beitzel, Steven M., et al. "Hourly analysis of a very large topically categorized web query log." Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval. *ACM*, 2004.

[44] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schutze. Introduction to Information Retrieval. *Cambridge University* Press, 1 edition, July 2008.

[45] P. Zezula, G. Amato, V. Dohnal, and M. Batko. Similarity search. The metric space approach. Advances in Database Systems, 32, *Springer*, 2006.

[46] Bolettieri, Paolo, Andrea Esuli, Fabrizio Falchi, Claudio Lucchese, Raffaele Perego, Tommaso Piccioli, and Fausto Rabitti. "CoPhIR: a test collection for content-based image retrieval." *arXiv preprint* arXiv:0905.4627 (2009).

[47] Hellerstein, J. L., Diao, Y., Parekh, S., and Tilbury, D. M. (2004). Feedback control of computing systems. *John Wiley and Sons.*

[48] T.F. Abdelzaher, J.A. Stankovic, C. Lu, R. Zhang, and Y. Lu, Feedback Performance Control in Software Services, IEEE *Control Systems*, vol. 23, no. 3, June 2003.

[49] C. Lu, J.A. Stankovic, G. Tao, and S.H. Son, Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms, *Real-Time Systems* J., vol. 23, no. 1/2, pp. 85-126, 2002.

[50] Lim, H. C., Babu, S., Chase, J. S., and Parekh, S. S. (2009, June). Automated control in cloud computing: challenges and opportunities. *Proceedings of the 1st workshop on Automated control for datacenters and clouds* (pp. 13-18). ACM.

[51] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. *In Proc.* of IM, 2002.

[52] G. Soundararajan, C. Amza, and A. Goel. Database replication policies for dynamic content applications. *In Proc. of EuroSys*, 2006.

[53] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier internet applications. *In Proc. of ICAC*, 2005.

[54] Wang, Guohui, and TS Eugene Ng. "The impact of virtualization on network performance of amazon ec2 data center." INFOCOM, *2010 Proceedings IEEE. IEEE*, 2010.