UNIVERSITY OF CALIFORNIA
RIVERSIDE

Improving User Query Results Through Diversification

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Md. Mahbub Hasan

March 2014

Dissertation Committee:

Dr. Vassilis Tsotras, Chairperson
Dr. Vagelis Hristidis
Dr. Eamonn Keogh
Dr. Harsha V. Madhyastha

The Dissertation of Md. Mahbub Hasan is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to express my gratitude to my advisor, Dr. Vassilis Tsotras, for his excellent help and support throughout my PhD study. Special thanks to my committee members, Dr. Vagelis Hristidis, Dr. Eamonn keogh and Dr. Harsha V. Madhyastha for their valuable feedback. Finally, I wish to thank my parents and my wife, Nurjahan Begum. Without them this would not have been possible.

ABSTRACT OF THE DISSERTATION

Improving User Query Results Through Diversification

by

Md. Mahbub Hasan

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2014
Dr. Vassilis Tsotras, Chairperson

Due to the large size of many structured and semi-structured databases, queries often return a large number of relevant results. Result diversification has recently been proposed as an approach to increase user satisfaction in search engines and recommendation systems. The top-k returned results are not only *relevant* to the query but also as *diverse* as possible from each other.

In this dissertation, we address three diversification related problems and propose efficient solutions for them. We firstly consider diversification on semi-structured data. We show that diversity can occur not only in the document content but also (and more importantly) in the document structure. We present a novel algorithm for diversification that considers both the structure and the content of the matched results. We propose a distance measure that is an order of magnitude faster than the standard tree-edit distance. The second problem considers how to balance relevance and diversity in the final top-k returned results. Previous works balance relevance and diversity mostly in a predefined fixed way. We propose a principled method for adaptive diversification of query results that minimizes the user effort to find the desired results by dynamically balancing the relevance and diversity. Finally, we consider the distributed diversification problem on large result-sets dispersed over many nodes. Using the MapReduce framework, we consider two distinct approaches, one that focuses on optimizing disk I/O and one that optimizes for network I/O. Our methods are iterative in nature, allowing the user to continue refining the result if more time is available. Moreover, we prove that this iteration process converges while producing a 2-approximate diversified result when compared to the optimal solution.

Furthermore, in the last part of the thesis, we investigate the problem of answering top-k queries satisfying spatial constraints. We propose a novel index structure that uses R-tree to tackle the spatial constraints, and pre-computed inverted lists to answer the top-k queries efficiently using the well known threshold algorithm. We present a model than can estimate the expected size of the inverted lists for the threshold algorithm using the data properties and query parameters. With the proposed model, we could reduce the index size significantly without sacrificing the performance of the threshold algorithm.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Diversification of Query Results

Due to the large size of many structured and semi-structured databases, queries often return a large number of relevant results. Recently, several works [33][71][39][81][83][12][48] have advocated diversification in ranking results as a way of improving the user satisfaction. The idea is to return to the user a set of results (top-k) that are as *relevant* as possible to the query and at the same time, as *diverse* as possible from each other. This is in contrast to the traditional approach of retrieving and ranking results of a query solely based on relevance. One reason for this paradigm shift is due to the inherent ambiguity in the user queries; especially the keyword queries which are often ambiguous and consequently, have multiple interpretations. As an example, consider the query *memory* which might refer to computer memory (RAM, ROM, Flash etc.) or the song Memory from the acclaimed musical Cats. For such ambiguous queries, a ranking that considers only relevance (e.g., [29]) might return a large number of similar results from just one interpretation of the query, e.g., DDR3 RAM, and a user with different search intent (say music) might not find any relevant result in the final top-k results.

Diversification helps to address this concern. In particular, a diversified ranking includes not only relevant (as judged by the underlying ranking function) results, but also results that may be less relevant and are diverse with respect to other results in the ranked list. A diversified ranking covers results from multiple interpretations of a query, thereby increasing the probability of the user finding desired results based on her query intent [33].

Another reason of diversification is to help the user to explore the result space. For example consider a dataset with images crawled from the web [9]. A user is interested in butterflies so she provides a query image of a butterfly as shown in Figure 1.1. Assuming that relevance is based on shape similarity, millions of images match this query (for simplicity, the result space shown in the figure contains only ten images) and the top-3 based on relevance-only are depicted. However, the user could have better explored the result space by using other image features such as texture, color, etc. to capture diversity among the returned results. The second top-3 results depict a lower-relevance higher-diversity set of butterflies.

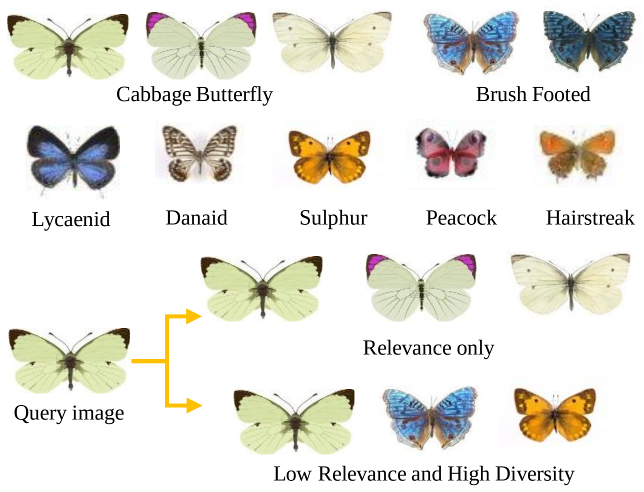


Figure 1.1: A set of different types of butterflies match with a sample query image and 3 images selected satisfying different criteria: relevance only and relevance with diversity

In this thesis, we address three diversification related problems and propose efficient solutions for them. We explain the problems in the next three sections (Sections 1.1.1 – 1.1.3).

The first two sections (Sections 1.1.1,1.1.2) consider diversification in a uniprocessor environment on a smaller result set (e.g. thousands of results). However, Section 1.1.3 considers diversification in a distributed environment on large datasets (e.g. millions of results).

1.1.1 Diversification on Semi-Structured Data

Vast repositories of semi-structured databases are accessed by user queries typically using an XML query language (such as XPath [32] and XQuery [28]). It is typical for such queries to return a large answer set, making it quite a challenge for the user to capture/view the whole result space. As stated earlier in this chapter, result diversification has been recently introduced as an approach to allow a user to explore the result space. Algorithms for XML result diversification have also been proposed recently [39][65] but, they only consider the data content (i.e. keywords) of the query.

What makes the problem challenging is that diversity can occur not only in the content of the documents but also (and more importantly) in the structure of the documents. Since the XML-based query can contain ancestor-descendent(//) or wildcard(*) relationships, there maybe significant structural differences (e.g., additional nodes in a matched path) among the returned results. Such diversity will not be explored by the content-only based diversification; instead we need an approach that takes into account differences *both* in the structure and content of the results.

To elaborate, let us consider an example document of bibliographic records shown in Figure 1.2. The document has three records: two PhD theses and a paper written by two different authors. An example XPath query (Figure 1.3) with one ancestor-descendent relationship describing “Find all bibliographic entries of Faloutsos”, has three exact matches shown by the thick lines in figure 1.2. Assume instead that we want to present the user with the two most diverse results. We should then provide the pair of matches (among the three possible pairs) that exhibits the highest diversity. Among the three matches, the one on the right (match 3) is structurally different from the other two because it is a record of a paper whereas the others are records for PhD theses. The matches on the left (match 1) and in the middle (match 2) are different because of the contents of the "PhDThesis" records (match 1 is a record for "Michalis" while match 2 is a record for "Christos"). Ideally, we would like to return to the user matches 2 and 3, since they are different both in content and in structure. Therefore, we need a diversification method that combines *both* the structural and content-based differences of the results.

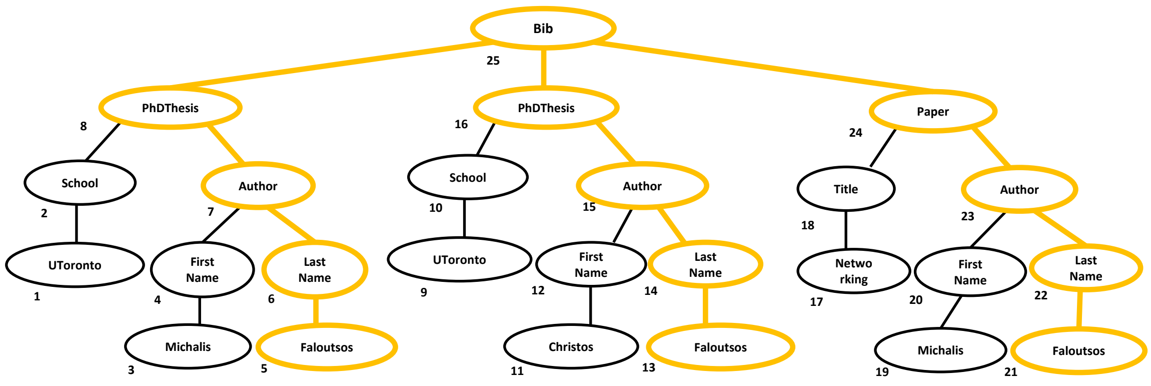
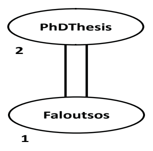


Figure 1.2: Bib document containing three records, (a) PhDThesis of Michalis Faloutsos, (b) PhDThesis of Christos Faloutsos and (c) a Paper of Michalis Faloutsos.



//PhDThesis//Faloutsos

(a) Tree Structure (b) XPath Expression

Figure 1.3: (a) An example query for the document in figure 1.2. (b) The XPath expression for the same query.

A naive way to find the most diverse k-subset from a set of N returned results, is to take the maximum of the total pair-wise distance as a measure of diversity for all of the $\binom{N}{k}$ subsets. Typically N and k are thousands and tens, respectively. Such an instance of the problem requires 100^2 distance computations. The distance measure, therefore, *must* be very efficient to keep the computation time tolerable. In addition to that, the number of times distances are computed *must* be reduced.

For structural query processing, a popular choice [18] of distance measure is the *tree edit distance* [87]. We focus on extending the tree edit distance in two ways. First, we consider the contents of the nodes and also the structural context of the query to perform well in presence of both types of differences and thus, provide meaningful diversification. Second, we leverage off the known skeleton (i.e. the query) of the results to compute the distance measure faster. We present a novel algorithm to achieve both of them. Our distance measure is comprehensive and our algorithm is at least an order of magnitude faster than the generalized tree edit distance with $O(n^2)$ worst case time complexity, where n is the number of nodes in the comparing trees.

Diversification is in general an NP-complete [83] problem. Therefore, enumerating all of the subsets to measure their goodness is necessary for exactness but prohibitive even if we have the best distance measure. For efficiency, we need an approximate algorithm that checks only a tiny fraction of the number of subsets the naive algorithm checks. In reality, the approximate algorithms still require orders of magnitude more time to do the diversification than it is required to produce the answer set. Therefore, the total latency for a user since the query is given becomes intolerable as the size of the result set increases. We present a novel pruning based speedup technique to mitigate such imbalance between the query processor and the diversification algorithm. Our technique speeds up the heuristic based approximate algorithms upto $2\times$ faster.

1.1.2 Adaptive Diversification of Query Results

At the beginning of this chapter, we mention that a diversified top-k results covers results from multiple interpretations of a query, thereby increasing the probability of the users' finding desired results from different interpretations. However, just focusing on diversity and displaying the set of most diverse results is ineffective since some of these results may have low relevance. In its most general form, the problem of query result diversification is modeled as a bi-criteria optimization problem [33][83][48], which uses a trade-off parameter(λ) to tune the relative effect of relevance and diversity factors during ranking. Using λ, the impact of the diversity factor can be increased for highly ambiguous queries so as to include more diverse elements in the result set; whereas for very specific (non-ambiguous) queries, this factor can be decreased to prevent inclusion of results of lesser relevance.

As an example, consider Figure 1.4(a) which depicts the result set returned for the query *Camera* (on a structured dataset like Amazon.com). As seen in the figure, the result set includes products from several categories including DSLR, Compact cameras and Accessories. Each result has a set of features (e.g. Brand, Megapixel, Zoom etc.). Note that the Lenses of DSLR cameras are considered in the Accessory category, therefore DSLR cameras do not have a Zoom feature. Figures 1.4(b)-1.4(d) show the Top-3 results selected by varying the trade-off parameter between diversity and relevance. Note that the relevance ranking [29] in this example assigns a higher score to DSLRs. For a user shopping for DSLR cameras, the ranking shown in Figure 1.4(b), which prefers relevance over diversity, might be sufficient. However, a user looking for a camera Lens would prefer the highly diversified ranking shown in Figure 1.4(d), where she could click on the Lens attribute value for attribute Type in the Accessory category to see more camera lenses.

Note that, for a given query, the user navigation cost (the user effort or actions required to find the desired results) varies for different choices of the trade-off parameter (see Figure 1.5, for the query Camera using the MMR algorithm [26] to implement diversified ranking). Moreover, in Chapter 3, we show experimentally, that the best value of the trade-off parameter λ varies for different queries. However, no previous work addresses the problem of computing a

ID	Product	Category	Features		Rel
1	Camera	DSLR	Brand: Canon	Megapixel: 18.0	0.9
2	Camera	DSLR	Brand: Nikon	Megapixel: 16.0	0.8
3	Camera	DSLR	Brand: Canon	Megapixel: 14.0	0.7
4	Camera	DSLR	Brand: Nikon	Megapixel: 12.0	0.6
5	Camera	DSLR	Brand: Sony	Megapixel: 12.0	0.6
6	Camera	Compact	Brand: Panasonic	Zoom: 7x	0.6
			Megapixel: 14.0		
7	Camera	Compact	Brand: Panasonic	Zoom: 5x	0.6
			Megapixel: 16.0		
8	Camera	Compact	Brand: Fujifilm	Zoom: 5x	0.4
			Megapixel: 12.2		
9	Camera	Compact	Brand: Kodak	Zoom: 3x	0.2
			Megapixel: 10.0		
10	Camera	Accessory	Type: Lens	Focal Length: 18 – 55 mm	0.3
11	Camera	Accessory	Type: Lens	Focal Length: 55 – 300 mm	0.2

(a) Result Set

ID	Product	Category	Features		Rel
1	Camera	DSLR	Brand: Canon	Megapixel: 18.0	0.9
2	Camera	DSLR	Brand: Nikon	Megapixel: 16.0	0.8
3	Camera	DSLR	Brand: Canon	Megapixel: 14.0	0.7

(b) High Relevance and Low Diversity

ID	Product	Category	Features		Rel
1	Camera	DSLR	Brand: Canon	Megapixel: 18.0	0.9
2	Camera	DSLR	Brand: Nikon	Megapixel: 16.0	0.8
6	Camera	Compact	Brand: Panasonic	Zoom: 7x	0.6
			Megapixel: 14.0		

(c) Moderate Relevance and Moderate Diversity

ID	Product	Category	Features		Rel
1	Camera	DSLR	Brand: Canon	Megapixel: 18.0	0.9
6	Camera	Compact	Brand: Panasonic	Zoom: 7x	0.6
			Megapixel: 14.0		
10	Camera	Accessory	Type: Lens	Focal Length: 18 – 55 mm	0.3

(d) Low Relevance and High Diversity

Figure 1.4: A subset of results of the query *Camera* and associated ranked list of results

trade-off parameter that will minimize the user effort. Instead, many hard-code it to a reasonable value (fixing the relative weight between relevance and diversity). Recently, several learning methods have been proposed [75][85] to learn the trade-off parameter λ. Unfortunately, these methods depend on training data provided by the experts which are expensive to collect and might not be available. Further, they compute a single trade-off parameter for a query, whereas we show how this trade-off changes as the query refinement or results viewing progress.

Because of the display interface, finding the desired result to a particular query might involve several steps. If the user does not find her desired result on the first page, then she might take additional actions to find the result, such as: (a) scan additional pages looking for the results of interest, or, (b) refine the query by clicking on a displayed attribute value to focus on a subset of the original results. Therefore, we want to compute at each step a set of k results (corresponding to a page in the users interface) that dynamically balances diversity and relevance (i.e., not fixed trade-off), such that the expected user navigation cost is minimized.

What makes the problem difficult is that when the query is posed, neither the target result nor the sequence of actions the user will execute to find it, are known. Therefore, to

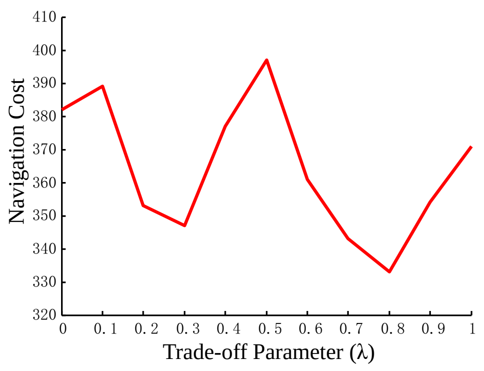


Figure 1.5: Navigation Cost vs. λ for Query Camera

compute the best set of results to display at each step, we must probabilistically consider all the unknown future user actions, which is a key challenge of our solution. For example, if the user poses the (highly ambiguous) query Camera while her target object is Lens, she will need further actions if we provide the results in Figure 1.4(b) (high relevance) in the first page. A higher diversified result set (like the one in Figure 1.4(d)) would have been more appropriate. If instead, a more specific query is posed, like DSLR Camera, then higher relevance and lower diversity is preferable, because the user may (with high probability) satisfy her search with just one page. Note that this dynamic balancing of relevance versus diversity can also occur, within the subsequent navigation steps of the same query, as it is progressively refined by the user (e.g. after posing the Camera query and getting the results in Figure 1.4(d), a user interested in lens might refine by selecting the condition Type: Lens).

To this end, we propose a user navigation model that considers factors such as the characteristics of the query result, the users familiarity with various refine conditions, the number of pages the user would have to navigate and the expected number of navigation steps required to reach a result of interest. The resulting model is adaptive to user actions and constructs a diversified result that minimizes the expected user effort. This is in contrast to the fixed diversity vs. relevance trade-off achieved by previous techniques, which leads to a much higher navigation cost, as shown in our experiments.

1.1.3 Distributed Diversification of Large Datasets

Our last contribution on diversification considers diversifying query results on large datasets. Since diversification is in general an NP-complete problem [83], many uniprocessor approximation algorithms have been proposed in the literature [81][83][36][84][26]. For large datasets, even the approximate algorithms are lethargic and therefore, parallelization becomes the method of choice for faster response time. MapReduce[38] has become extremely popular over the last few years for processing large volume of data in a distributed setting. Several database problems (e.g. clustering[35], join[21]) have been solved successfully using MapReduce. Unfortunately, none of the previous uniprocessor diversification algorithms can be easily extended to a MapReduce distributed environment. For example, parallelizing the computation of the method in [26] requires one read of the entire file (result dataset) to produce one diversified result in the output. Therefore, computing the top-k diverse result set, requires at least k file reads which becomes infeasible (in disk I/O [72]) for large result datasets. Another challenge is to minimize the communication cost between the distributed nodes (network I/O [35]). This motivates the need to develop distributed algorithms that can diversify large result datasets (e.g. millions of results) fast while preserving the quality of the diversification, as well as achieving linear speedup for increasing number of nodes.

In this thesis, we propose two distinct approaches for result diversification using the mapreduce framework (Chapter 4). We show that different approaches perform better in different scenarios (e.g. disk rate, network speed, number of cluster nodes, data size). We present a cost model that can dynamically choose the most suitable approach for a given computation environment.

1.2 Spatial Top-k Queries

We also investigate the problem of answering top-k queries satisfying spatial constraints. This kind of problem is important in several applications. For example, in online social networks (say Twitter[10]) the users express their thoughts in short messages (*tweets*). Each tweet is associated with a geolocation which denotes the originating location of the tweet. An interesting problem would be to summarize the tweets' content from a spatial region (say, Los Angeles) in a few keywords (top-k terms) to investigate the current trending topics in the region.

In Chapter 5, we propose a novel index structure for fast answering of such spatial queries. Our index structure uses R-tree [50] to tackle the spatial constraints, and pre-computed inverted (sorted term) lists to answer the top-k queries efficiently using the well known threshold algorithm [46]. Furthermore, we propose a model than can estimate the expected size of the sorted term lists for the threshold algorithm using the data properties and query parameters. With the proposed model, we could reduce the index size significantly without sacrificing the performance of the threshold algorithm.

1.3 Thesis Overview

The rest of the thesis is organized as follow, In Chapter 2, we explain our novel distance measure and diversification algorithms for semi-structured data. We present our adaptive diversification algorithm in Chapter 3. Chapter 4 describes our distributed diversification approaches and the proposed cost model. Chapter 5 explains our index structure for spatial top-k queries. Finally, we conclude and present future work in Chapter 6.

Chapter 2

Semi-Structured Diversification

This chapter explains our solutions for diversification on semi-structured data [55]. Section 2.1 provides necessary background and formulates our problem. We discuss our distance measure in Section 2.2 and the diversification algorithm in Section 2.3. Section 2.4 evaluates the performance of our algorithms. Finally, Section 2.5 presents related work.

2.1 Problem Formulation

An XML document is an ordered labeled tree T. T is a graph with vertices $V(T)$ and edges $E(T)$. An edge (u,v)$\in E(T)$ represents a parent child relationship where u is the parent of v. Only the root has no parent. A node u can have zero or more children in a strict left to right order. Nodes with zero child are leaves. $anc(u)$ defines the set of ancestors of node u. Every node has a label denoted by $label(u)$. A postorder traversal of a tree visits the children of a node from left to right before visiting the node. For example, the postorder traversal of the tree in Figure 1.2 is shown by the numbers beside each node. We denote the nodes of a tree by the postorder sequence $t_1, t_2, \ldots, t_n$, where $n = |T|$ is the number of nodes in T. The subtree rooted at node t_i is denoted by T_i. The postorder sequence of T_i is a subsequence of T ending at t_i and starting at $l(t_i)$. $l(t_i)$ is the leftmost node of the tree T_i. For example, $l(7) = l(4) = 3$

and $l(25) = 1$ in Figure 1.2. We are given a query Q which is also an ordered labeled tree where edges represent XPath axes. We are also given a set of XML documents $\mathcal{D}$, in which we find matches for the query.

A map between a node s_i in a tree S and a node t_j in a tree T is an ordered pair (s_i, t_j). We define a relation $M : V(S) \rightarrow V(T)$ or simply a $mapping$ $M : S \rightarrow T$ such that M maps some nodes of S to some nodes of T with the following conditions.

1. $1 \leq i \leq |S|$ and $1 \leq j \leq |T|$
2. For any two pairs (s_i, t_j) and (s_u, t_v) in M

   (a) $i = u$ if and only if $j = v$ (One-to-one).

   (b) s_i is to the left of s_u if and only if t_j is to the left of t_v (Sibling-order).

   (c) s_i is an ancestor of s_u if and only if t_j is an ancestor of t_v (Ancestor-order).

Note that M is not a function and, therefore, is not defined for all nodes in S and T. Nodes mapped by M capture similar structure in both S and T. $M' : T \rightarrow S$ is the *inverse* mapping of M such that for all $(s, t) \in M$, $(t, s) \in M'$. If M is defined for every node $s \in S$, then M is a *complete* mapping. If M is complete, M' is not guaranteed to be complete. If both M and M' are complete, they are called *maximal* mappings.

M is called an *outer* mapping if (i) for every leaf s in S, $M(s)$ is a leaf in T and (ii) $root(T) = M(root(S))$. If both M and M' are outer then they are called *minimal* mapping. Figure 2.2(b) shows an example of minimal mapping between S and T.

An *exact match* of a query Q is another ordered labeled tree T, such that there is a complete and minimal mapping $M : Q \rightarrow T$ and for all $(q, t) \in M$, $label(q) = label(t)$. There has been many algorithms proposed for finding all of the exact matches of query Q in $\mathcal{D}$ [14][23][59][78][30]. There is also algorithm [18] that finds *approximate* matches where query Q may not have complete mappings. Each approximate match of the query Q is an exact match of query Q', where Q' is a relaxed version of the original query Q [16]. We consider the matching algorithm $\mathcal{A}$ as given and $\mathcal{A}(\mathcal{D},Q)$ is the set of matches denoted by $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$. We denote a distance measure by $d(.,.)$, which computes the dissimilarity between two matches T_i and T_j.

A set $R \subset \mathcal{T}$ of size k is the most diverse if the total pair-wise distance $\sum_{T_i,T_j \in R} d(T_i, T_j)$ is the maximum. The matches in R are said to be the top-k diverse matches for the query Q in the document set $\mathcal{D}$.

[TOP-k DIVERSE MATCHES]. *For a given Q and $\mathcal{D}$, find the k-subset R of the set of matches $\mathcal{T}$ such that the total pair-wise distance of R (i.e. $\sum_{T_i,T_j \in R} d(T_i, T_j)$) is the maximum over all such subsets.*

The optimal algorithm to find the top-k diverse matches requires enumerating all the k-subsets of the set $\mathcal{T}$ and selecting the one with maximum pair-wise distance. This algorithm has $O(|\mathcal{T}|^k)$ time complexity and therefore, too slow for interactive queries. To solve the problem efficiently there are two lines of attack; speeding up the distance measure and considering only a fraction of the subsets heuristically. In Section 2.2, we describe our approach of computing distance very fast by taking both the structure and content of the query into account. In Section 2.3, we describe our heuristic approach to find the diverse subset efficiently.

2.2 Distance Measure for Diversification

To diversify a set of matches for an XML query, we need a distance measure that can compare two trees. The tree edit distance [87] is the most widely used distance measure for tree structures. The idea is to transform one tree to the other such that the total cost of the sequence of edit operations performed for the transformation is minimum and hence the distance between the two trees.

There are three types of edit operations. The *delete* operation removes a node n from the tree and connects the children of n as the children of the n's parent preserving the sibling order of the children. The *insertion* operation on a node n adds an edge from some node p to n and makes a subsequence of children of p the children of n. The *rename* operation changes the label of a node.

For every operation, an associated cost is defined. The cost can depend on the operation, the label of the node(s) being operated on as well as the context at which the operation is being performed. The simplest cost model assumes equal cost for all of the three operations: insertion, deletion and rename. Such cost model makes tree editing distance symmetric i.e. transforming any of the trees to the other yields the same distance.

Any valid mapping $M : S \to T$ can be translated to a sequence of edit operations to convert one tree to another. The sequence of operations is (i) delete all non-mapped node in S, (ii) rename all mapped nodes that do not have the same label and, (iii) insert all non-mapped nodes in T. Since, M preserves the structural similarity by the three conditions described in the definition of mapping, at any intermediate stage of the sequence of operations M remains valid. The converse is also true. If we are given a sequence of edit operations, there exists a mapping $M : S \to T$ that has cost no higher than that of the sequence of edit operations[87]. Therefore finding the least costly sequence of edit operations is the same as to finding the least costly mapping as defined below.

Definition 1 *Given a mapping $M : S \to T$ and a equal cost for the operations, we define the $cost(M)$ as*

$$cost(M) = (|S| - |M|) + (|T| - |M|) + |M_m|$$

where $M_m = \{(s,t) \in M | label(s) \neq label(t)\}$.

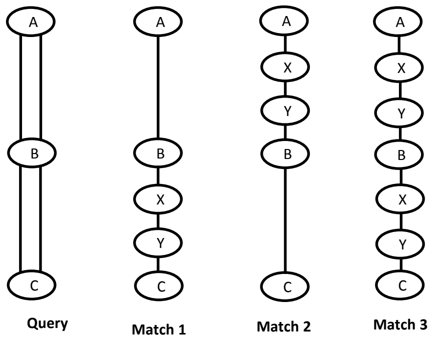
The term $|S| - |M|$ denotes the number of non-mapped nodes in the tree S and this is the number of deletions we need to perform. Similarly, $|T| - |M|$ is the number of insertions and $|M_m|$ is the number of rename operations.

Definition 2 *Tree edit distance between S and T, $TED(S, T)$, is the smallest cost over all mappings $M : S \rightarrow T$.*

Tree edit distance finds the best possible mapping preserving the structural similarity. But while computing the distance between two matches, TED does not utilize the information that both the matches have complete-minimal mapping from the query. In the next two subsections, we present a new algorithm that uses these two mappings for computing distances. We start with adding the structural context sensitivity in the distance measure and, add the content sensitivity in the later subsection.

2.2.1 Context Aware Diversity

We first provide a simple example showing that TED fails to capture the desired dissimilarity because of ignoring the structural context of the query.



Tree Edit Distance

	Match1	Match2	Match3
Match1	0	2	2
Match2	-	0	2
Match3	-	-	0

Tree Edit Distance preserving Query Mapping

	Match1	Match2	Match3
Match1	0	4	2
Match2	-	0	2
Match3	-	-	0

Figure 2.1: For the query shown on the left there are three matches found. The distance values for the tree edit distance and our proposed variant are shown on the right.

Consider the example in Figure 2.1 where we have three matches for the query shown on the left. Based on the structure of the query, the two most diverse matches should be match

1 and 2. The reason is the XY segment is located in different parts of matches 1 and 2 while match 3 has some parts common with both match 1 and 2, separately. But according to the tree edit distance, all of the pairwise distances are 2. Therefore, TED cannot distinguish the two most diverse matches (i.e. 1 and 2) in this example. More precisely, while converting match 1 to match 2, TED needs only two operations: delete B from match 1 and insert B as in match 2. Recall both of the B nodes in match 1 and 2 are mapped from the node B in the query. This implicitly maps the two B nodes of match 1 and 2 together. Therefore, B must not be deleted or inserted while the editing distance between match 1 and 2 is computed in the context of the query. However, as in the above example, the generalized algorithm for tree edit distance does not always preserve this query mapping. If we consider an implied mapping between the matches using the mappings from the query, the distance between match 1 and 2 becomes 4, and thus, makes them the most diverse pair.

In this section we describe our algorithm to compute the modified tree edit distance that considers the query mappings as contextual information. We denote the modified distance measure as *Seeded Tree Edit Distance (STED)*.

Consider the set of matches $\mathcal{T}$ of a given query Q. Let $S, T \in \mathcal{T}$ be any two matches for the query Q as shown in Figure 2.2 and M^S and M^T are the complete minimal mappings from Q to S and T.

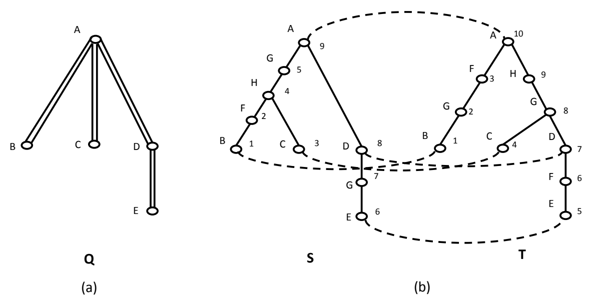


Figure 2.2: An example query (Q) on the left with two matches S and T. The induced minimal mapping between S and T is shown by the dashed lines.

We define a new mapping $M : S \to T$ where $(s_i, t_j) \in M$ for all $(q, s_i) \in M^S$ and $(q, t_j) \in M^T$. Note that, M may not be complete but always minimal. We call M a *seed* map. From now, M always refers to a minimal mapping and, therefore, the direction of the map is not important at any point.

Note that, if $\hat{M} : S \to T$ is a *super mapping* such that $\hat{M} \supseteq M$ then $cost(\hat{M}) \leq cost(M)$ under equal costs of edit operations. Because we want to preserve the seed mapping M as the context, we modify the tree edit distance to find a super mapping of M that minimizes the total cost instead of *any* mapping.

Definition 3 *The seeded tree edit distance, $STED(S,T,M)$, between S and T given a minimal mapping $M : S \to T$, is the smallest cost over mappings $\hat{M} \supseteq M$.*

To compute STED using existing algorithms for computing tree edit distance, we can just change the cost model trivially. More precisely, if $(s,t) \in M$ then cost of deleting s, inserting t and mapping s (or t) to a different node $x \neq t$ (or s) is raised to infinity. This change in cost model guarantees that (s,t) would be in the optimal mapping.

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
S_1	0	1	2	α	α	α	α	α	α	α
S_2	1	1	1	α	α	α	α	α	α	α
S_3	α	α	α	0	α	α	α	α	α	α
S_4	α	α	α	α	α	α	α	α	α	α
S_5	α	α	α	α	α	α	α	α	α	α
S_6	α	α	α	α	0	1	α	α	α	α
S_7	α	α	α	α	1	1	α	α	α	α
S_8	α	α	α	α	α	α	1	α	α	α
S_9	α	α	α	α	α	α	α	α	α	6

Figure 2.3: Tree edit distance matrix of S and T

The classic algorithm for tree edit distance is a dynamic programming algorithm which computes a matrix of size $|S| \times |T|$ where a cell (i, j) denotes the tree edit distance between S_i and T_j. For example, Figure 2.3 shows the matrix for trees in Figure 2.2 when the change in the cost model is adopted. Clearly most of the entries are invalid and contribute nothing to the final distance value. This motivates us to develop an efficient algorithm for finding $STED$ for two trees when the seed map is given. The algorithm is described sequentially and is justified with necessary definitions and lemmas as we go along.

Let $U^T = \{x | x \in V(T) \text{ and } \exists_y [(x, y) \in M \text{ or } (y, x) \in M]\}$ be the set of mapped nodes in a tree T. Note that all of the leaves and the root of T are in U^T. If the tree is divided at every node in U^T by keeping two copies in the two halves, we will get $|U^T|$ chunks from T. Let $C(T, M)$ or C^T in short denote the set of chunks found in tree T and C^T_u denote the chunk rooted at a node $u \in U^T$. For example, Figure 2.4 shows the chunks of S and T from Figure 2.2[1].

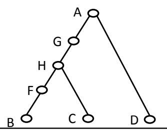
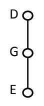
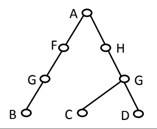
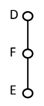
Chunks of S		○ B	○ C		○ E
Chunks of T		○ B	○ C		○ E
STED	5	0	0	1	0

Figure 2.4: Mapped chunks of S, T and corresponding $STED$

Since $M : S \to T$ is a one-to-one mapping, every chunk C^S_u from tree S has a mapped chunk $C^T_{M(u)}$ in the tree T. The submapping $M_u : C^S_u \to C^T_{M(u)}$ induced from M is minimal by definition. Note that no internal node in C^S_u is mapped by M. Moreover, no internal node in C^S_u will be mapped by the optimal mapping $\hat{M}$ to a node in $C^T_{M(v)}$ where $u \neq v$. The following lemma describes the fact more formally.

Lemma 1 *Optimal mapping $\hat{M}$ for $STED(S, T, M)$ will not map any node from one chunk C^S_u to another chunk $C^T_{M(v)}$ such that there are $u, v \in U^S$ and $u \neq v$.*

[1]The reader may wonder why defining the leaves as *tiny* chunks. In reality, they have inconsequential effect on the performance but, helps to simplify the description by far.

Proof 1 *Let $n \in C_u^S$ and $m \in C_{M(v)}^T$ are two nodes in S and T (see Figure 2.5). For contradiction, lets assume $(n, m) \in \hat{M}$. Therefore, u is $anc(n)$ in S and $M(v)$ is $anc(m)$ in T. Since v and $M(v)$ are matched so v is $anc(n)$ in S. Now, by construction, v can not be in the path from u to n. Therefore, v is also $anc(u)$. Since u and $M(u)$ are matched, $M(u)$ has to be $anc(m)$ and $desc(M(v))$. Because no internal node in the path from $M(v)$ to m can be a mapped node by M, this is a contradiction.*

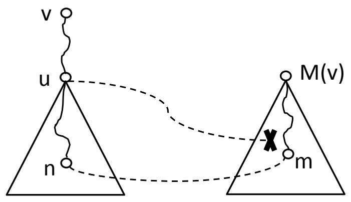


Figure 2.5: Contradiction of Lemma 1

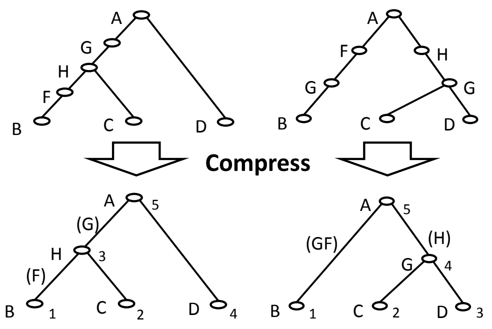
Using the above lemma, we can now say that finding optimal mappings for every pair of mapped chunks is sufficient. If we only find the mappings for the mapped pairs of chunks, compute the editing distance for these mappings and then, sum these distances for all of the mapped pairs of chunks; it is the same as the optimal editing distance between S and T. Mathematically,

$$cost(\hat{M}) = \sum_{u \in U^S} cost(M_u : C_u^S \to C_{M(u)}^T) \tag{2.1}$$

To find the mapping between C_u^S and $C_{M(u)}^T$, we now present an $O(n^2)$ algorithm where n is the number of nodes in C_u^S and $C_{M(u)}^T$. Since there is a one to one mapping between chunks, from now on we denote C_u^S and $C_{M(u)}^T$ by S and T for simplicity of description. Similarly, we denote M_u as M.

Definition 4 *The leaf-sequence $L(u)$ of a node u in a tree T consists of the leaves of T rooted at u in the left to right order.*

For example, in the left tree of Figure 2.6 $L(5) =< 1, 2, 4 >$. We extend the definition of mapping for a leaf-sequence by taking the sequence of the matched nodes in the other tree i.e. $M(L(5)) = M(< 1, 2, 4 >) =< M(1), M(2), M(4) >$.



Post Order Number	Leaf Sequence	Stem Sequence		Leaf Sequence	Stem Sequence
1	B	BF		B	BGF
2	C	C		C	C
3	BC	HG		D	D
4	D	D		CD	GH
5	BCD	A		BCD	A

Figure 2.6: Compressed trees of first mapped pair chunks in Figure 2.4 and corresponding leaf and stem sequence

Recall that in a chunk, only root and leaves are mapped by M. No other internal nodes in a chunk will be in M. Our goal is to find the mappings for these internal nodes in $\hat{M}$. The following lemma states the key of our algorithm classifying the internal nodes that will not be mapped by $\hat{M}$ at all. In other words, two internal nodes can be mapped only if their leaf sequences are also mapped by M.

Lemma 2 *For a node u in S, if there is no node in T with the leaf-sequence $M(L(u))$, then u is not mapped by $\hat{M}$.*

Proof 2 *For contradiction let u is matched with v in T (see Figure 2.7). Since v has a different leaf-set from $M(L(u))$, there is at least one map $(w, M(w))$ such that either $w \in L(u)$ and $M(w) \notin L(v)$ or the vice versa. Without losing generality, assume $w \in L(u)$. Since u is an $anc(w)$, v has to be an $anc(M(w))$ according to the definition of mapping. Since $M(w) \notin$*

$L(v)$, by construction, there is a $x \in L(v)$ which is also $anc(M(w))$. Since x is a leaf node of a chunk, there has to be $(y, x) \in M$ such that y is a $anc(w)$ and a $desc(u)$. This leads to contradiction since no chunk has an internal node mapped by M.

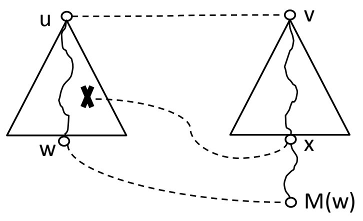


Figure 2.7: Contradiction of Lemma 2

There can be multiple nodes in the same tree having the same leaf-sequence and nodes with the same leaf-sequence form a path in a tree. Based on this observation, we can compress S and T by collapsing paths to single nodes. Figure 2.6 shows the compressed trees of the first pair of chunks in Figure 2.4, where nodes in a single path with the same leaf sequence are shown as the label of the corresponding edge. We call a collapsed path a *stem*.

Definition 5 *A stem is a subsequence P of the nodes in the postorder sequence of a tree such that $\forall_i L(P_i) = L(P_1)$ where P_i is the ith node in P. $L(P)$ is defined to equal $L(P_i)$.*

The table in Figure 2.6 shows the stem and leaf sequence of all the nodes of the two compressed trees. We are now required to find the pairs of stems from the two trees having leaf sequences mapped from one to the other. We need to do it efficiently without checking all possible pairs of stems. We argue that, one parallel scan through S and T in post-order is sufficient.

Our algorithm parallely scans the nodes in trees S and T in post-order. Assume the algorithm is currently looking at two nodes u and v from S and T, respectively. If their leaf-sequences are mapped by M, we compute the mapping between their stems in a way described later. If their leaf-sequences are not mapped by M then, we can skip *either u or v* and advance the scan with the confidence that the skipped node will never be mapped by an $\hat{M}$. The lemma 3 justifies this decision. Note that the parallel scan requires at most $|S| + |T| - 1$ checks for pairs of stems.

Lemma 3 *Let u be a node in S. Let p_l and p_r be the leftmost and the rightmost nodes in $M(L(u))$ in the tree T, respectively. Also let v be a node in T and, q_l and q_r are the leftmost and rightmost nodes in $L(v)$ in the tree T. If $M(L(u)) \neq L(v)$ then*

1. *if $p_r > q_r$ or ($p_r = q_r$ and $p_l < q_l$), then for no node $x > u$, $M(L(x)) = L(v)$.*
2. *If $q_r > p_r$ or ($q_r = p_r$ and $q_l < p_l$), then for no node $x > v$, $M(L(u)) = L(x)$.*
3. *no other case occur.*

Proof 3 *If x and y are any two nodes and $x > y$ (i.e. x is to the right of y) then only one of the following is true*[2].

$\star$ *x is an ancestor of y and, therefore, $L(y)$ is a subsequence of $L(x)$. $L(y) \subseteq L(x)$*

$\star$ *At their least common ancestor, x is in a right subtree to y and, therefore, they have no common subsequence. $L(x) \cap L(y) = \emptyset$.*

1. *Note that $M(L(u)) \neq L(v)$. The given condition essentially describes three possible scenarios as shown in Figure 2.8(a-c). For any node $x > u$, there can be two cases.*

   $\star$ *If $L(x) = L(u)$ then trivially $M(L(x)) \neq L(v)$.*

   $\star$ *If $L(u) \subset L(x)$ or $L(u) \cap L(x) = \emptyset$ then there is a leaf $t \in L(x)$ where $M(t) \notin M(L(u))$. Now $M(t)$ can be in two possible places.*

   • *$M(t) < p_l$: Definitely $p_r \in M(L(x))$. Since $M(t) \notin L(v)$ (in Figure 2.8(b-c)) and $p_r \notin L(v)$ (in Figure 2.8(a)), therefore $M(L(x)) \neq L(v)$.*

   • *$M(t) > p_r$: Trivially $M(t) \notin L(v)$, therefore $M(L(x)) \neq L(v)$.*

[2]We are abusing the set operations for sequences. We believe the context clarifies the intended meaning.

2. *Since M is minimal, using the inverse mapping M' and similar arguments as above we can prove that for no node $x > v$, $M(L(u)) = L(x)$.*

3. *For any two nodes in a tree it is not possible to have both $L(x) \cap L(y) \neq \emptyset$ and $L(y) \not\subseteq L(x)$. Therefore, the remaining cases as shown in the Figure 2.8(d) cannot occur.*

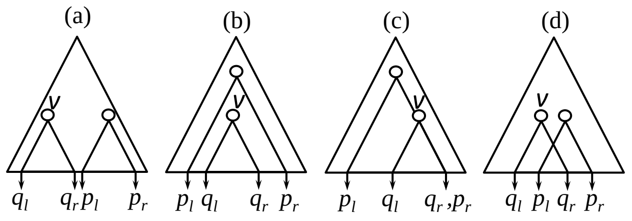


Figure 2.8: The tree T. (a-c) Three possible scenarios for case 1. (d) An impossible scenario which cannot occur as described in case 3 of lemma 3.

The remaining piece of the puzzle is to find the optimal mapping between the stems having leaf-sequences mapped from one to the other. The following lemma describes how we compute the optimal mapping and the distance as well. Here, SED stands for string edit distance [62]; The proof of this lemma is skipped for brevity as it is a straight forward specialization of the tree edit distance for paths (see [87] for details).

Lemma 4 *If P^S and P^T are two stems of S and T, respectively, such that $M(L(P_1^S)) = L(P_1^T)$, then $TED(P^S, P^T) = SED(P^S, P^T)$.*

Using the above lemmas 1 to 4 we have designed our algorithm 1 for computing STED. $STED(S, T, M)$ takes in two trees S and T and divides them into chunks. For each mapped pair of chunks, the algorithm parallelly scans to see if there is any pair of stems with mapped leaf sequence. For mapped stems, the algorithm computes the string edit distance of the stems and add the value to the total cost. For non-mapped pair of stems, the algorithm adds the length of one of the stems that is guaranteed to remain unmapped in $\hat{M}$. Note that, adding length of the stem is equivalent to insertion/deletion of all of the nodes in the stem.

The running time of the proposed algorithm is $O(n^2)$ with the requirement of a minimal seed mapping M. In the worst case, when both of the trees are simple paths the algorithm costs exactly $O(n^2)$ time to compute the string edit distance where n is the number of nodes in the trees. Note that, the standard tree edit distance is at least $O(n^3)$ [18] and, therefore, our algorithm is faster than the generalized tree edit distance by at least an order of magnitude (i.e. a factor of n) while being more meaningful as well.

2.2.2 Content Based Diversity

In the previous section, we have described how the mapping induced by the query can be used to compute accurate and efficient distances for diversification. If we use STED for the matches in Figure 1.2, both (1, 3) and (2, 3) produce distance values of 1 and, consequently, result in a tie. Because (1,3) involves the same person (i.e. "Michalis") while (2,3) does not, the obvious choice for the diverse pair is (2,3). Now the question is, how we can modify STED to capture true diversity by breaking such tied situation.

The answer is, by taking the contents (i.e. nodes in the document that are structurally unrelated to the query) into consideration. Contents can create different levels of differences between matches even if the matches are structurally similar to each other. For example, in Figure 1.2 the first two matches are PhDThesis records linked to Faloutsos, but their authors are different.

When two nodes of a map (s_i, t_j) have the same label (i.e. $label(s_i) = label(t_j)$), under the equal cost model no cost is added to the total. There can be differentiating features in the branches of the subtrees S_i and T_j that are not matched to the query and, hence are ignored by STED. For example, First Name/Michalis is a branch of Author in match 1 which is not matched to any part of the query. Let S_i^R and T_j^R are the two trees rooted at s_i and t_j that contain the remaining branches unmatched to the query. We add a $correction$ cost $c \in [0, 1]$ as a cost of the map (s_i, t_j) to capture the amount of mismatch present in S_i^R and T_j^R.

The correction cost c can trivially be computed by simply taking the $TED(S_i^R, T_j^R)$ and normalizing by the maximum possible distance between a pair of matches. However, TED is too costly to use for computing the fractional contributions from the contents just to break the ties. We develop a novel approach to obtain the correction cost c efficiently.

At first, we classify nodes of an XML document in one of the four categories: *value, attribute, entity* and *connector*.

$\star$ All leaf nodes are *Value* nodes.

$\star$ A parent of a value is an *Attribute*

$\star$ A parent of an attribute is an *Entity* if it is not an attribute itself.

$\star$ A node other than the above three is a *Connector*.

Similar classification has been proposed in [64] when the *Document Tree Descriptor (DTD)* is not available. We scan the documents once to identify the type of every node. For example in Figure 1.2, there are four attributes; School, First Name and Last Name, Title and, three entities; PhDThesis, Author and Paper.

The four classes of nodes are defined keeping the usual structure of an XML document in mind. In general, an attribute (similar to a "variable" in programming languages) has exactly one value and no other child. Therefore, attributes do not require the above mentioned correction cost as their values are always compared. In contrast, entities generally have multiple attributes and may need some correction cost. Since connectors have no attribute/value, having correction cost for them is not meaningful.

To compute the correction cost for entities, we only consider the number of mismatched attributes. Two attributes are mismatched if they have the same label but different values. For example, in Figure 1.2 the entity Author in all the documents has two attributes First Name and Last Name. While comparing the two Author nodes in 1 and 2, the number of mismatched attributes is 1 because of the different first names. If an attribute is present in only one entity and absent in the other, it does not confirm any difference between the entities and, therefore, these attributes are not counted as mismatch [65]. For example, had there be a Middle Name attribute for the Author entity in match 1, the number of mismatched attributes would still be 1.

We define the correction cost for entities as below.

$$c_e = \frac{\text{Number of mismatched attributes}}{\text{Total number of distinct attributes}} \tag{2.2}$$

Here, the total number of attributes is a normalization constant. Examples of correction costs for entities: $c_{Author} = 0.5$ for the match pairs (1, 2) and (2, 3). Let us revisit the problem of breaking ties for the matches in Figure 1.2. (1, 3) has a distance of 1 and (2, 3) has a distance of 1.5 when the correction costs are used with the equal cost model. Thus, adding content awareness breaks the tie meaningfully in favor of the true diverse set of matches.

2.2.3 Algorithm for STED

Algorithm 1 shows the pseudocode for finding STED between two matches S and T when the minimal mapping M is given. Consider two matches S and T of Figure 2.2 as the inputs of algorithm 1. The algorithm creates all the chunks as shown in Figure 2.4 at lines 1 and 2 using the algorithm 3 . For each pair of mapped chunks, we initiate two pointers n and m (lines 5-6) that iterate through the chunks in their respective postorder sequence. The algorithm also computes (using the algorithm 5 at line 7-8) two sequences (i.e. arrays), B and E, that store the beginning and ending leaves of the leaf-sequences. For example, B_i and E_i are the beginning

Algorithm 1 $SeededTreeEditDistance(S, T, M)$

```
Require: $S$ and $T$ are two trees, $M : S \to T$ is a minimal mapping
Ensure: Return the seeded tree edit distance
 1: $C^S \leftarrow Chunks(S, M)$//algorithm 3
 2: $C^T \leftarrow Chunks(T, M)$
 3: $sum \leftarrow 0$
 4: for each pair $(C^S_u, C^T_{M(u)})$ do
 5:    $n \leftarrow$ first node of $C^S_u$ in post-order
 6:    $m \leftarrow$ first node of $C^T_{M(u)}$ in post-order
 7:    $B^S, E^S \leftarrow LeafSequences(C^S_u)$
 8:    $B^T, E^T \leftarrow LeafSequences(C^T_{M(u)})$
 9:    while $n$ and $m$ are not nil do
10:       $i \leftarrow n, j \leftarrow m$
11:       $P^S, n \leftarrow FindStem(n, C^S_u)$
12:       $P^T, m \leftarrow FindStem(m, C^T_{M(u)})$
13:       if $B^T_j = M(B^S_i), E^T_j = M(E^S_i)$ and $i$,$j$ are not leaves then
14:          $sum \leftarrow sum + SED(P^S, P^T)$
15:       else if $B^T_j = M(B^S_i), E^T_j = M(E^S_i)$ and $i$,$j$ are leaves then
16:          $sum \leftarrow sum + SED(P^S - i, P^T - j) + cost(i, j)$
17:       else if $M(E^S_i) > E^T_j$ or $(M(E^S_i) = E^T_j$ and $M(B^S_i) < B^T_j)$ then
18:          $sum \leftarrow sum + |P^T|$, $n \leftarrow$ i
19:       else
20:          $sum \leftarrow sum + |P^S|$, $m \leftarrow$ j
```

Algorithm 2 $FindStem(n, C)$

```
Require: A chunk $C$ and a node $n$ in $C$
Ensure: Return the stem $P$ and the next $n$ after the stem
 1: $i \leftarrow n, P \leftarrow \epsilon$
 2: while $L(i) = L(n)$ and $n$ is not nil do
 3:    $P \leftarrow Concatenate(P, n)$
 4:    $n \leftarrow$ next node of $C$ in post-order
```

and ending leaves of $L(i)$.

At every iteration, the stems of the of nodes n and m are found (lines 11-12) by the algorithm 2. Algorithm 2 creates and returns the stem of node n by concatenating nodes with the same leaf-sequence as $L(n)$ in the post-order of C. The algorithm also returns the first node after n with different leaf-sequence.

When the stems are ready, the algorithm 1 checks to see if the stems have mapped leaf-sequences (i.e. the beginning and ending leaves are same). The algorithm handles the pair of stems with mapped leaf-sequences in two different ways (lines 13 and 15) based on the first node of the stem. If the first nodes (i and j) are leaves, by the definition of chunks they are

Algorithm 3 $Chunks(S, M)$

```
Require: A tree $S$ and a minimal $M$ mapping to or from $S$
Ensure: Return $C$, a set of chunks of $S$
1: $C \leftarrow \emptyset, Q = \{x|x \in V(S)$ and $\exists_y[(y,x)or(x,y) \in M]\}$
2: for each $u$ in $Q$ do
3:     $C_u \leftarrow FindChunk(u, \epsilon, Q)$
4:     add $C_u$ to $C$
```

Algorithm 4 $FindChunk(n, C_u, Q)$

```
Require: A node $n$, the list of mapped nodes $Q$ and the current chunk $C_u$ to add in
Ensure: Return the modified current chunk $C_u$
1: add $n$ to $C_u$
2: if $C_u = \epsilon$ or $n \notin Q$ then
3:     for each child $v$ of $n$ in left to right order do
4:         $C_u \leftarrow FindChunk(v, C_u, Q)$
```

matched to the query nodes by the query processor and we want to preserve their mapping. Note that, if i is a leaf, so is j and vice versa. To preserve the mapping between the leaves, the algorithm computes string edit distance for the rests of the stems and add the cost for the mapping of the leaves (line 16). When i and j are not leaves, the algorithm simply takes the string edit distance between the stems.

When the leaf-sequences are not mapped, there can be two cases as described in the lemma 3. In the first case, the node i remains active for the next iteration but stem the P^T of the node j is inserted/deleted (line 18). In the remaining case, the node j remains active and P^S is inserted/deleted (line 20).

Figure 2.9 shows the iterations of the loop at line 4 for the first pair of chunks in Figure 2.4. The final $STED(S, T, M)$ is 6 which is equal to the last entry of the tree matrix in Figure 2.3.

Algorithm 5 $LeafSequences(S)$

```
Require: A tree $S$
Ensure: Return two arrays $B$ and $E$ containing the start and end nodes of the leaf sequences of every
    node in $S$
1: $B \leftarrow \epsilon, E \leftarrow \epsilon$
2: $FindLeafSequence(Root(S), B, E)$
```

Algorithm 6 $FindLeafSequence(u, B, E)$

```
Require: A node $u$ and two arrays $B$ and $E$ to store the start and end of $L(u)$
1: if $u$ is a leaf then
2:     $B_u \leftarrow u, E_u \leftarrow u$
3: else
4:     for each child $v$ of $u$ do
5:         $FindLeafSequence(v, B, E)$
6:     $i \leftarrow$ leftmost child of $u$
7:     $j \leftarrow$ rightmost child of $u$
8:     $B_u \leftarrow B_i, E_u \leftarrow E_j$
```

2.2.4 Properties of the Distance Measure

The seeded tree edit distance (STED) has some elegant properties: triangular inequality, fast lower bound and upper bound.

2.2.4.1 Triangular Inequality

The original tree edit distance holds the triangular inequality. STED also holds the triangular inequality as long as the seed mapping is same.

Theorem 1 $STED(T_1, T_2, M) + STED(T_2, T_3, M) \geq STED(T_1, T_3, M)$

Proof 4 *The STED can be obtained by creating a cost model from M. Since, M is fixed, and TED holds triangular inequality, STED also holds triangular inequality.*

When we add content awareness in STED, it becomes a bit complicated. If we choose to use the described definition for "mismatch" in the previous section, the triangular inequality *does not* hold. But there is a way around for performance critical applications where triangular inequality is the key for performance. If we consider absent attributes as *mismatched* attributes, triangular inequality *holds* with the given definition of correction cost.

2.2.4.2 Lower Bounds

Another desirable property of a distance measure is the availability of low cost lower bounds for fast similarity search. There is a simple lower bound for STED that requires $O(n)$ time for computation while STED itself requires $O(n^2)$.

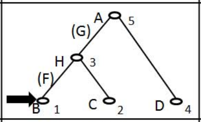
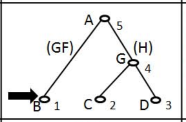
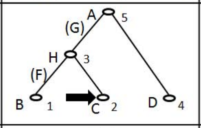
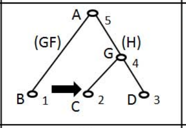
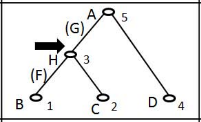
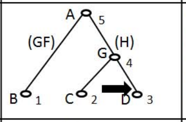
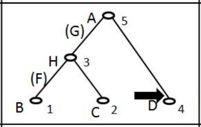
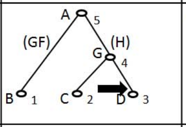
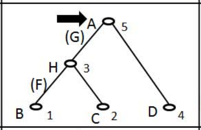
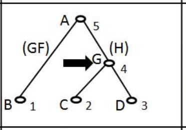
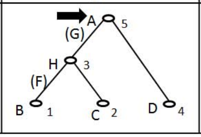
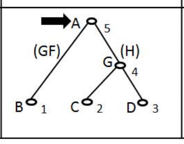
Iteration	C_1^S	C_1^T	Comment	STED
1			• leaf Sequences same and nodes are leaves •Find SED of stems •Advance the pointers lines (15 -16)	1
2			Same as iteration 1	0
3			• leaf Sequences are not same • delete HG from C_1^S • advance the pointer of C_1^S Step 20	2
4			Same as iteration 1	0
5			• leaf Sequences are not same • delete GH from C_1^T • advance the pointer of C_1^T Step 18	2
6			• leaf Sequences same but nodes are not leaves •Find SED of stems •Advance the pointers lines (13 -14)	0

Figure 2.9: Iterations performed by the algorithm 1 for the first pair of chunks in Figure 2.4.

STED requires computing the string edit distance for stems with mapped leaf-sequences. A trivial lower bound for String edit distance is the absolute difference between the lengths of the strings being compared. If we use such trivial bounds whenever STED needs a string edit distance, the resulting distance value is a lower bound to the original STED.

2.2.4.3 Upper Bounds

Similar to lower bound, an upper bound of STED can be computed by taking the sum of the lengths of the two stems used in string edit distance computations. Such an upper bound also requires $O(n)$ time for computation.

2.3 Diversification

Result diversification is in general an NP-complete [83] problem. Many heuristics [40] have been proposed to find approximate diverse result set (greedy heuristic, interchange heuristic, clustering heuristic, etc.). In this chapter we utilize the greedy heuristic algorithm [41] (see algorithm 7) which selects a seed of one or two matches (line 1). Once the seed is selected, the algorithm finds the next object to add in the final result set (line 4-5). To do that, the algorithm compares each of the remaining matches to the already added matches in the result set and add the one that has the maximum total distance to the current result set. The algorithm stops once k matches are added to the result set (line 3). The algorithm computes linear number of editing distances on the number of matches ($|(\mathcal{T})|$) as $k << |\mathcal{T}|$. We consider three methods *Diameter Seed* [41], *Lower Bound Seed* and *Random Seed* for selecting the seeds.

- ***Diameter Seed:*** Select the farthest pair of points (diameter) in the set of matches (i.e. $\mathcal{T}$) as the seed. Finding the diameter is inherently quadratic in time complexity for high dimensional data.
- ***Lower Bound Seed:*** Select the farthest pair of points by using the lower bound (as described in Section 2.2.4) instead of the true tree edit distance. This approach is also quadratic but promises to be faster.
- ***Random Seed:*** Select one match as the seed at random. This approach is efficient but suffers degradation in quality (see Section 2.4).

Algorithm 7 $Greedy - Diversification(\mathcal{T}, K, Algo)$

```
Require: A set of matches $\mathcal{T}$, the final result set size $k$
Ensure: Return the set $R$ of top-$k$ diverse matches from $\mathcal{T}$
1: $R \leftarrow$ initial seed(s)
2: $\mathcal{T} \leftarrow \mathcal{T}$-$R$
3: while $|R| < k$ do
4:     find $T_i$ in $\mathcal{T}$ such that the total pair-wise distance of
       $R \cup T_i$ is maximum for all $T_i \in \mathcal{T}$
5:     $R \leftarrow R \cup T_i, \mathcal{T} \leftarrow \mathcal{T}$ - $T_i$
```

In Figure 2.10, we demonstrate the trends of the seed selection algorithms as the number of matches increases. We compare the running time of the algorithms with that of a standard query processor (LCS-Trim [78]). As the figure suggests, the curves are diverging and therefore, the motivation of having a diverse result set no longer worth the waiting time after the matches are available from the query processor. Clearly we need an efficient diversification algorithm taking sublinear time with the increasing number of matches.

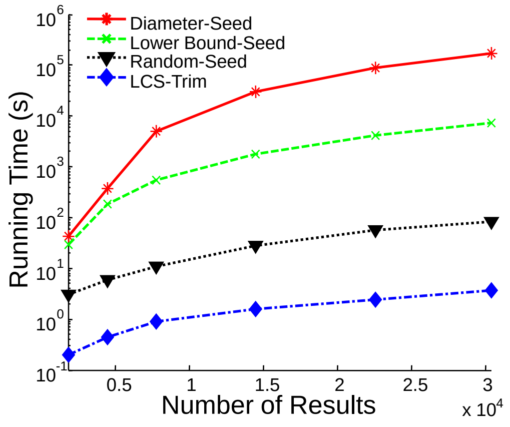


Figure 2.10: Comparison of running times of different diversification algorithms with a sample query processor, LCS-Trim.

2.3.1 Novel Heuristic for Seed Selection

As we have discussed random-seed linear time diversification algorithm improves the running time but degrades the quality, which motivates to propose a new and fast heuristic for seed selection, so to have similar time complexity as random-seed while improve the overall quality of diverse result set.

We propose a new scoring technique for selecting the initial seed. Instead of a random seed, we want to start from one of the matches which have an extreme value for a relevant but low cost feature. One such feature is the count of nodes in a match. Counting nodes for every match and selecting the one with the maximum count takes one linear scan over the matches. Note that, this process does not require any distance computation. We name this selection method as *QMax*.

However, there is still a significant gap between the query matching algorithm and the fastest diversification algorithms (Figure 2.10) indicating a large latency for the users posing the query. As an effort to reduce the gap further, we develop pruning strategies based on triangular inequality property of STED.

2.3.2 Pruning while Maintaining Diversity

In the greedy diversification process, all of the remaining matches are compared against the current result set (say R_c) to find the one having the maximum total distance. More precisely, one needs $|R_c|(N - |R_c|)$ comparisons to add the next match to the result set and a total of kN distance computations to complete the whole process where k is the size of the final result set and N is the size of the initial result set. To reduce such a large number of STED computations, we use our novel pruning technique for diversification.

For efficient pruning, we need a very tight upper bound of the original total distance for a very low cost. The reason is once we know that an upper bound is smaller than the current maximum we can safely ignore the candidate match. Triangular inequality can be used to get a very low cost upper bound of any original distance by using the following formulae.

$$\begin{aligned} UB(A,B) = UB(A,C) + UB(B,C) &\geq UB(A,C) + d(B,C) \\ &\geq d(A,C) + d(B,C) \geq d(A,B) \end{aligned} \tag{2.3}$$

Note that, computation of an upper bound requires either two true distances or two upper bounds or one upper bound and one true distance.

How should we compute and use these upper bounds to achieve maximum pruning? We propose algorithm 8 for this purpose, which computes the true distances between every successive pair in the output of query matching algorithm. Let us assume at any instance, C is a candidate match for which we want to compute the total distance to the current result set R_c. Let us also assume that the upper bound or the true value of the total distance from C_{-1} (i.e.

previous match of C in the output of query matcher) is $UB(R_c, C_{-1})$. The upper bound of the total distance from C is then:

$$UB(R_c, C) = d(C, C_{-1}) * |R_c| + UB(R_c, C_{-1}) \tag{2.4}$$

Note the recursive nature of the above equation which enables sequential computation of the upper bound of the total distance from the candidates. If the upper bound is larger than the current maximum total distance, the algorithm just makes the upper bound equal to the true total distance.

Although the bounds can be computed elegantly in the above way, the bounds do not posses enough tightness because of the repeated use of triangular inequality. In the case when the successive pairs of candidates are very similar to each other, the first term of the right hand side of equation 2.4 remains very small and, thus making the bounds more tight. Fortunately, the query matching algorithms typically outputs the matches with strong local similarity. Consequently, concatenated bounds achieve significant speedup over the linear diversification algorithms.

2.3.3 The Algorithm for Pruning Based Diversification

The above pruning technique is implemented within the diversification algorithm described in the algorithm 8. The algorithm takes as input the set of matches $\mathcal{T}$ and the desired size of the output k. The algorithm selects the result with the maximum number of nodes as the seed for the diversification process (line 1). Then it iterates for the rest of the positions in the output set. $SumofDistances(R, T_i)$ denotes the sum of distances from the result T_i to all results in R. In each iteration the algorithm 8 scans the result set $\mathcal{T}$ to find the furthest result (i.e. $bestT$) from the current output set R. While checking a candidate result T_j, the algorithm computes the upper bound U in line 7 and test to see if the candidate is larger than the $currentBest$. If the test succeeds, the algorithm computes the sum of the distances from T_j to R (line 9) and updates appropriate histories (lines 11-12) if the true sum is larger than the $currentBest$.

Once the scan is complete in an iteration, the $bestT$ is removed from the $\mathcal{T}$ and added to the output set R. Note that the number of distance computations at line 7 and line 9 of the algorithm 8 depends on the amount of previously computed distance values we store in memory for reuse. Pessimistically, we assume there is no extra memory here.

Algorithm 8 $DiversificationByPruning(\mathcal{T},K)$

```
Require: A set of matches $\mathcal{T}$ and the size of the final result set $k$
Ensure: Return the set $R$ of top-$k$ diverse matches from $\mathcal{T}$
 1: $R \leftarrow \{T_x$ such that $|T_x|$ is maximum$\}$
 2: $\mathcal{T} \leftarrow \mathcal{T}$-$T_x$
 3: for $i \leftarrow 2$ to $k$ do
 4:     $U_{-1} \leftarrow SumofDistances(R, T_1)$
 5:     $currentBest \leftarrow U_{-1}$
 6:     for $j \leftarrow 2$ to $|\mathcal{T}|$-$|R|$ do
 7:         $U \leftarrow (i-1) * d(T_j, T_{j-1}) + U_{-1}$
 8:         if $U > CurrentBest$ then
 9:             $U \leftarrow SumofDistances(R, T_j)$
10:             if $U > CurrentBest$ then
11:                 $currentBest \leftarrow U$
12:                 $bestT \leftarrow T_j$
13:         $U_{-1} \leftarrow U$
14:     $R \leftarrow R \cup \{bestT\}$
15:     $\mathcal{T} \leftarrow \mathcal{T}$-$\{bestT\}$
```

2.4 Evaluation

To experimentally demonstrate the utility of our algorithms, we have used the *Treebank* dataset[3] because of its rich structural variations. We have selected seven queries (Table 2.1). The queries are structurally different from each other to cover several extreme cases. The experiments are performed in a standard unix system on a 2.10 GHz processor and 4GB of RAM.

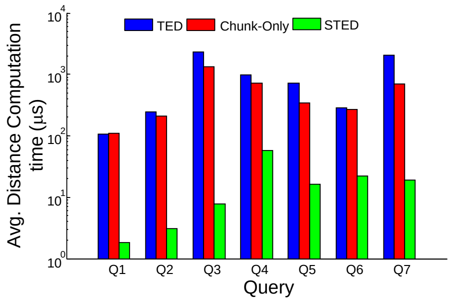
[3]http://www.cs.washington.edu/research/xmldatasets/

Query	**XPath Expression**	**Matches**
Q_1	$//S[/VP/NP][/VP]$	11752
Q_2	$//EMPTY//X/VP/PP//NP$	1412
Q_3	$//S[//NNS][//JJ][//VP//NNP]$	30349
Q_4	$//EMPTY//S/VP/*/SBAR//PP//NN$	28463
Q_5	$//S[//DT][/VP//PRP\_DOLLAR\_]$	4559
Q_6	$//S[/NP][/VP/*/PP//NNP]$	35326
Q_7	$//S[//NP/NNP][//CC][/*/VP[/VBZ]$ $[//NP/\_NONE\_]]$	906

Table 2.1: Query Set

2.4.1 Speedup of STED

Our first experiment is to evaluate the performance of STED in comparison with the generalized tree edit distance that uses a modified cost model to preserve the seed map. We also use an intermediate algorithm which divides the trees into chunks as STED but, computes regular tree edit distances (with the modified cost model) for every pair of chunks. Figure 2.11(a) shows the average time taken to compute the distance between two results for the queries in Table 2.1 by all of the three methods. STED performs at least two orders of magnitude faster than the tree edit distance while the *chunk-only* version achieved notable amount of speedup demonstrating the importance of the our chunking approach. In Figure 2.11(b), we show the average time required to compute just the correction costs for content aware distance. By comparing the bars to those of Figure 2.11(a), we conclude that adding correction costs for content awareness does not increase the computation time significantly.



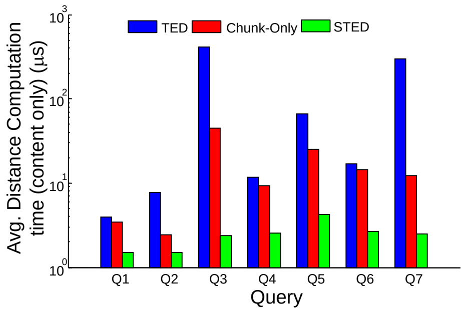


Figure 2.11: Average time taken to compute (a) the distance (structure and content) (b) only the content distances between two results

2.4.2 Evaluation of Diversification Algorithms

We compare the seed selection algorithms, Diameter Seed (***Dia***), Lower Bound Seed (***LB***) and Random Seed (***Rand***), against our proposed heuristic, ***QMax***. Note that, Dia and LB require quadratic number of distance computations for seed selection, while the $Rand$ and $QMax$ need no distance computation for seed selection.

2.4.2.1 Qualitative Analysis

For qualitative analysis, we compare the final result sets returned by different algorithms with the result set generated by the optimal (brute force) algorithm for the same query and input parameters. We use two criteria to measure the quality, *precision* and *percentage gap*. Precision of an algorithm is the fraction of the optimal top-k matches that the algorithm returns. Percentage gap is the percentage of the deviation of the total pair-wise distances of an algorithm from that of the optimal algorithm. For the efficiency of the brute force algorithm, the number of candidate results (N) is fixed to be 100 (Figure 2.12). In both the measures $QMax$ performs better than LB and $Rand$, and very close to Dia.

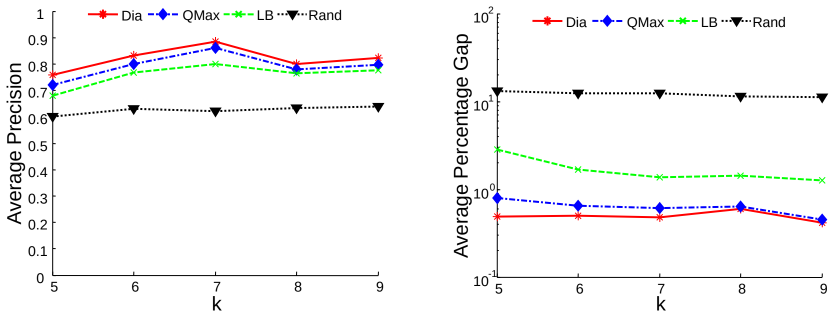


Figure 2.12: (a) Average Precision vs k (b) Average Distance Gap vs k.

2.4.2.2 Scalability Analysis

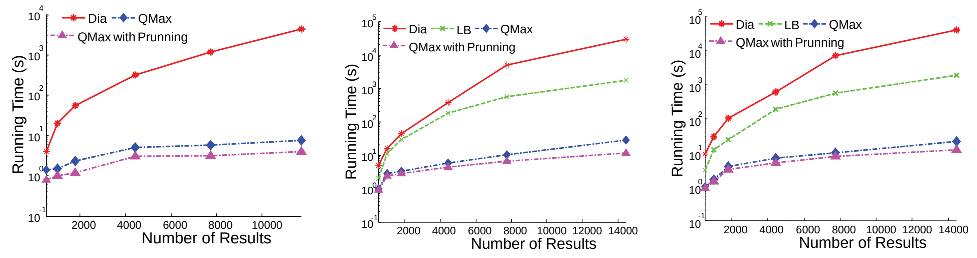
Our next experiment is to evaluate the scalability of the diversification algorithms as the number of candidate results and k increase. We have shown the experiments for three different queries

(Q_1,Q_3,Q_4). Q_1 allows no structural variation and, therefore, result into diversification for contents only. Q_4 is a path query, while Q_3 more complicated and can generate a wide range of structurally diverse results. (Note that, for Q_1, as there is no structural variation, Dia and LB show same performance. We skip the curve for LB for visual clarity)

In Figure 2.13, we show the running times of the algorithms to produce top-25 diverse results from different result sets. Clearly $QMax$ outperforms the quadratic algorithms Dia and LB (Figure 2.13). When the pruning is added to $QMax$, the running time is further improved up to a factor of 2. Note that our pruning technique speeds up greedy methods without changing the accuracy. The curve for Rand is skipped for visual clarity as it overlaps the curve for $QMax$.

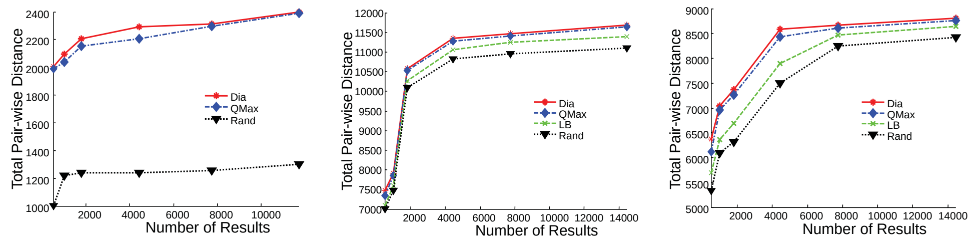
In Figure 2.14, the total pair-wise distances of the top-25 diverse matches are shown for different algorithms. In both Figures 2.12(b) and 2.14, $QMax$ achieves insignificantly less accurate results compared to Dia. Reader may interpret this little loss on accuracy as the price paid for the huge speedup shown in Figure 2.13. In practice, the small difference in the total distance does not add subjectively noticeable changes in the reported output.

We have also studied the running time and the total pair-wise distance of the algorithms for different values of k for a fixed result set size (Figures 2.15 and 2.16). Quadratic seed selection methods (Dia and LB), need so large an amount of time for selecting the seed that the rests of the algorithms (with complexity $O(k|(\mathcal{T})|)$) negligibly increase the total time (Figure 2.15). In contrast, $QMax$ selects the seed very fast and therefore, the running time for $QMax$ (with or without pruning) linearly increases with k. This ensures a possible adaptation of our algorithm as an *anytime* algorithm, where the user can preemptively stop the computation at any time with the best answers she could get in the elapsed amount of time.



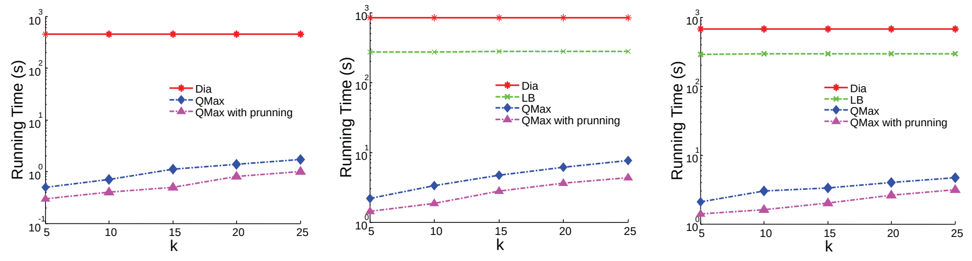
(a) Q_1 (b) Q_3 (c) Q_4

Figure 2.13: Running time vs $|(\mathcal{T})|$ ($k = 25$)



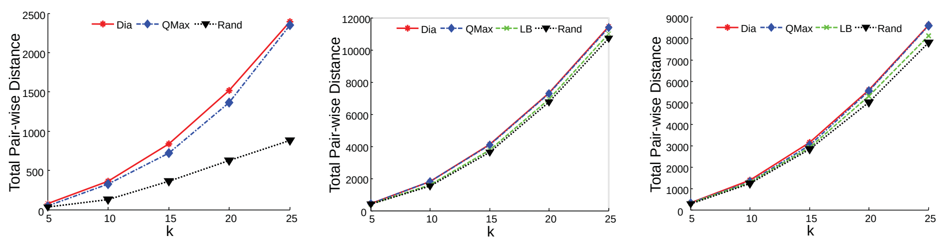
(a) Q_1 (b) Q_3 (c) Q_4

Figure 2.14: Total Pair-wise Distance vs $|(\mathcal{T})|$ ($k = 25$)



(a) Q_1 (b) Q_3 (c) Q_4

Figure 2.15: Running time vs k ($|(\mathcal{T})| = 5000$)



(a) Q_1 (b) Q_3 (c) Q_4

Figure 2.16: Total Pair-wise Distance vs k ($|(\mathcal{T})| = 11500$)

2.5 Related Work

Our work in this chapter is related to three different branches of computing research.

Query Processing on Semi-structured Data (i.e. XML documents) has been addressed in several occasions and there are three different types of algorithms have been proposed; path based [14][30], twig based [23][59] and sequence based [78]. In path based methods, the original query is divided into paths from root to leaves and, the matches corresponding to these paths are joined together to construct a complete match. Twig based methods perform better than path based ones by considering the twig as a whole and, therefore, eliminates expensive stitching operations. Sequence based methods first convert the query and document into sequences and perform a search for the query subsequence in the document sequence. It has been recently shown that LCS-Trim [78] outperforms the other approaches. Hence we have chosen it as the query processor for our algorithms in this chapter. It should be noted that the choice of query processing algorithm is orthogonal to our problem.

Diversifying search results is also a well addressed area of research. [48] provides a general framework for the result diversification problem. Specialized solutions for relational and web databases are also proposed [12][41][81]. There are rich surveys on diversification [42][51][83] that classify the available algorithms into two principal types, the greedy best first approach and the iterative gain maximization approach. In this chapter, we focused on the greedy best first method because it needs few linear scans of the data and does not depend on a large number iterations to produce better quality results.

Despite the wide range of work on diversification, there is little on diversifying XML query results. [39][65] proposed result diversification based on keyword queries instead of classic XPath or XQuery queries and concentrate only on the content information of the documents. None of these methods formally consider the structural diversity among the results. We present the first of such diversification algorithm that treats the results as trees rather than collections of labels.

Computing dissimilarity between trees using the *tree edit distance* (TED) [87] is one of the first methods for comparing tree-like structures. [49] proposed $O(n^2)$ lower and upper bounds of the tree edit distance. [17] provides an $O(n^2)$ algorithm for approximating tree edit distance through string edit distance. None of these methods defined the special case of computing TED in presence of a given seed mapping. We present the first *exact* algorithm to compute the seeded tree edit distance.

Chapter 3

Adaptive Diversification

This chapter is based on [53]. At first, we provide a user navigation cost model that captures the actions of a user navigating a result set (Section 3.1). The cost model is necessarily probabilistic since it computes the cost based on possible future actions taken by the user while navigating a result set. We propose ways of estimating the expected cost in Section 3.2. We then show that the problem of computing the best set of results to minimize the expected user effort is NP-hard (Section 3.3) and propose efficient approximate algorithms to compute an appropriately balanced diversified result-set that minimizes the expected user navigation cost (Section 3.4). We present the results of an extensive experimental evaluation of the proposed techniques compared to state-of-the-art ranking methods using two real datasets (Section 3.5). We validate our cost model and measure user navigation time with a user study at Amazon Mechanical Turk, which shows that our methods outperform the state-of-the-art (Section 3.6). Finally we discuss the related work in Section 3.7.

3.1 Problem Definition

We proceed with the problem setting (Section 3.1.1), and describe our navigation cost model (Section 3.1.2), which mimics the actions of a user navigating query results and quantifies the user effort, that is, assigns cost to the user actions. We then conclude with a formal problem definition (Section 3.1.3).

3.1.1 Preliminaries

Let $D = \{r_1, \ldots, r_n\}$ be a database consisting of n tuples and $\mathcal{A}= \{A_1, \ldots, A_m\}$ be a set of attributes. Each attribute has an associated domain $ADom(A_i)$ consisting of uninterpreted constants. The database D is heterogeneous and each tuple $r_i \in D$ has a value for a subset $A_i \subseteq \mathcal{A}$ of attributes and a null (ϵ) value for the rest.

Query: The user exploring D, navigates the results $R_Q \subseteq D$ of a query Q which could be a keyword query. Each attribute-value combination in the results of R_Q, denoted by $c : A_i = v_i$, is a condition and can be used to refine the query. We denote the set of all conditions of R_Q by $C(R_Q)$.

Example: Figure 1.4(a) shows a result-set R_Q for $Q : Camera$, where the user could refine the query by selecting condition $c_i : Brand = Canon$, which would return the two Canon cameras ($\#1, \#3$).

Paginated Result Subset S_k: Typically, the result set R_Q is too large to fit into a single page on the user interface. These results are therefore paginated and only a small subset $S_k \subseteq R_Q$ of size k is presented to the user at a time. The size k of the result subset depends on the display size of the device. For example, e-commerce and web-search interfaces typically show $10 - 15$ results at a time on desktop browsers and 5 on mobile devices. The latter are the key focus of this chapter, which are even more challenging because the screen size does not allow displaying other result information like facets.

Example: Figures 1.4(b-d) show examples of various paginated result subsets (with $k = 3$) for the query results in Figure 1.4(a).

The choice of the subset S_k is critical in determining the effectiveness of a search interface. In particular, S_k should contain relevant and a diverse set of results.

Let $rel(r, Q)$ be the relevance score of a result $r \in R_Q$, where a higher score means the result r is more relevant to query Q. Existing object relevance ranking functions like [29] can be used to compute $rel(r, Q)$. Also, let $dist(r_i, r_j)$ be the distance between two results r_i,$r_j \in R_Q$. For instance, $dist(r_i, r_j)$ can be the Euclidean distance between the vectors representing r_i and r_j. Currently, most systems model search result diversification as a bi-criteria optimization problem that balances the effect of relevance and diversity using a trade-off parameter ($\lambda \in [0, 1]$) as follows:

$$S_k = \underset{S \subseteq R_Q, |S|=k}{\text{argmax}} \left[(1-k)(1-\lambda)\sum_{r_i \in S} rel(r_i, Q) + 2\lambda \sum_{r_i, r_j \in S} dist(r_i, r_j)\right] \quad (3.1)$$

We described several scenarios where this bi-criteria optimization is problematic. The primary reason is the difficulty in selecting a value of λ for a given navigation step of a query.

Instead of fixing the trade-off between relevance and diversity, we model the diversification problem in terms of the user navigation effort. We do this by designing a holistic model of the user navigating a list of paginated results that considers all the actions taken by a user, discussed next.

3.1.2 Navigation Cost Model

Users execute queries on a search interface to satisfy a certain information need, which may be satisfied by a certain objects in the query result. Given a user query Q and corresponding result set R_Q, the query interface presents the first page of results $S_k \subseteq R_Q$ of size k to the user. Each result consists of a set of attribute-value conditions. These conditions are selectable (e.g. by clicking on the associated link), thereby refining the query. Note that the search interface does not provide any facet conditions to refine the results, but the results serve this dual purpose. Such an arrangement is desirable especially in mobile devices where there is not enough space to show both results and facet conditions. In particular, the user chooses among the following possible actions at each step:

1. *TERMINATE*: If the users search need is satisfied on the current page, the search is terminated.

2. *NEXT-PAGE(R_Q)*: The user can navigate to the next page of the result set in the hope of satisfying her search need there. In this case, the search interface computes the next page of results and presents these results to the user.

3. *REFINE (Q,c)*: Typically, a user has a notion of the properties or conditions that a target (desired) object must have. If one of these conditions is found in one of the displayed results, then the user can refine her query by selecting (clicking) this condition c. The query is then refined to $Q \wedge c$. For example, if a user is looking for *Compact* cameras while reading through the result subset in Figure 1.4(d), she could click on the *Compact* attribute value. This user repeatedly executes REFINE and NEXT-PAGE actions until the target object is found, at which point the user TERMINATEs the navigation. This iterative result navigation process is captured by the recursive navigation model presented by algorithm 9. At the beginning of each step (line 1), the system computes a page of k results to be presented to the user. The user reads all the results, represented by READ-RESULT(S_k) and the rest of the navigation repeats recursively.

In our model, the user executes an action based on the displayed result set. Such navigation models, where the user only selects actions proposed by the system, have been used extensively in the navigation of keyword-based query results [60][73][27]. Each user action

Algorithm 9 $NAVIGATE(Q, R_Q)$

```
1: $S_k = COMPUTE - PAGE(Q, R_Q)$
2: $READ - RESULT(S_k)$
3: if search need satisfied by $S_k$ then
4:     $TERMINATE$
5: else
6:     Choose one of the following:
7:         (a): Select a condition $c \in C(S_k)$ to refine; $REFINE(Q, c); Q \leftarrow Q \wedge c$
8:         (b): $NEXT - PAGE(R_Q \setminus S_k)$
9:     $NAVIGATE(Q, R_Q)$
```

(REFINE, READ-RESULT, NEXT-PAGE) is an effort on the part of the user. The total effort of the user to satisfy her search need is the *navigation cost*.

As an example, consider the user navigating the result set in Figure 1.4(a) using the initial pagination in Figure 1.4(d). Further assume that the user is interested in Lenses with $55 - 300$mm Focal Length ($\#11$). As a first step, the user would read (READ-RESULT) the first page of 3 results. Next, the user REFINEs by Type: Lens to see only the Lens results, since she is interested in Lenses. Up to this point the navigation cost consists of 3 READ-RESULT actions and 1 REFINE. Upon REFINE, the interface presents the 2 lens results ($\#10$,$\#11$), which the user reads and finds the desired object, thereby TERMINATING the navigation. The overall navigation cost is 5 READ-RESULT $+ 1$ REFINE.

If we assume that reading a result incurs unit cost (as was assumed in [60]), and the cost of REFINE (click) action is a constant greater than one (say $\alpha = 3$), the total cost is $5.1 + 1.3 = 8$. This assumption about the constant α reflects our belief that REFINE incurs more user effort than reading a result, since the user has to consider all the conditions and then decide on a condition to click on. Similarly, the user could do NEXT-PAGE instead of REFINE if she does not find any useful condition to refine on. The cost of NEXT-PAGE is β.

3.1.3 Problem Statement

The overall navigation cost depends on the result subset S_k that is presented to the user at each step. For example, if the user in the example above is presented with a paginated result subset containing the Camera Lens with $55 - 300$ Focal Length($\#11$), then she would find the desired target object on the first page and TERMINATE the search. In this case, the total navigation cost is 3 (3 READ-RESULTs).

Therefore, we need to compute a paginated result subset $S_k \subseteq R_Q$, that will minimize the expected navigation cost, by appropriately balancing relevance and diversity. Let $cost(Q, R_Q, S_k)$ denote the cost of navigating the result set R_Q of a query Q, using the paginated result subset S_k. Then the minimal cost of navigation, $cost(Q, R_Q, k)$, is the cost of the paginated result set S_k^{opt} , amongst the $\binom{|R_Q|}{k}$ k-subsets, that has the minimum cost. Formally,

Minimum Cost Diversification (R_Q, Q, k)**:** Given a query Q and its result set R_Q ($|R_Q| \geq k$), compute the result subset S_k^{opt} of size k such that the expected navigation cost incurred to satisfy the users search need is minimized.

$$cost(Q, R_Q, k) = min_{S_k \subseteq R_Q} cost(Q, R_Q, S_k) \tag{3.2}$$

Next, we show how to estimate $cost(Q, R_Q, S_k)$, the cost of a result subset S_k for a result set R_Q.

3.2 Navigation Cost Estimation

The navigation cost of a result set R_Q, computed as discussed in Section 3.1.2, depends on the actions taken by the user in reaching a target object and can be exactly determined after the navigation is complete. However, solution to the minimum cost diversification problem requires the selection of result subset S_k before knowing what sequence of actions the user will perform after viewing S_k. In this section, we propose a way to estimate the cost of navigating a result subset S_k by means of a probabilistic cost model that assigns uncertainty measures to each possible action a user can take and computes the expected navigation cost for a given S_k.

We begin by introducing the probability measures that capture the uncertainty in user action. In the user navigation model explained by algorithm 9, the user, at each navigation step, has three choices (1) TERMINATE the navigation (line 4) (2) REFINE by a condition (line 7) and (3) to go to the NEXT-PAGE of results (line 8), and we introduce probability measures for each of these actions (ways to estimate these probabilities are proposed in Section 3.2.1).

• P_T **(TERMINATE Probability):** This is the probability that the user finds the result she is looking for in S_k, and therefore terminates the navigation process.

• P_R **(REFINE Probability):** This is the probability that the user chooses to refine the result set R_Q by adding a condition c to the query Q. On the other hand, the user could instead choose to see the next page of the results. Since these are the only two choices supported by the navigation model, the probability that the user chooses the NEXT-PAGE action is $(1 - P_R)$.

• P_R **(REFINE by condition c Probability):** If the user chooses to REFINE, then she also has to select a condition $c \in C(S_k)$ to refine by. The probability P_c captures the probability that the user selects a condition c.

Given the probabilities defined above, the entire navigation process can be expressed by the following recursive cost equation (cost(.,.,.) is overloaded):

$$\begin{aligned} cost(Q, R_Q, S_k) =& S_k + (1 - P_T). \\ & \Big[P_R.\{\alpha + \sum_{c \in C(S_k)} P_c.cost(Q \wedge c, R_{Q \wedge c}, k)\} \\ & + (1 - P_R)(\beta + cost(Q, R_Q \setminus S_k, k))\Big] \end{aligned} \tag{3.3}$$

This cost equation can be described as follows:

1. The user reads the results in S_k, and decides about her next action. The cost for reading the results is $|S_k|$ (assuming unit cost for the READ-RESULT action). If the user finds the target object then she terminates the navigation.

2. Otherwise, with probability $(1 - P_T)$, she can either refine the query or go to the next page,

   a. The user decides to refine the query with probability P_R. Let α be the cost for a REFINE action. As the user can select any condition $c \in C(S_k)$, we consider the cost associated with each selection candidate c (shown as $cost(Q \wedge c, R_{Q\wedge c}, k)$) weighted by the P_c value.

   b. With probability P_N the user decides to go to the next page. β is the cost of a NEXT-PAGE action, and the cost entailed by the NEXT-PAGE action and the cost of the rest navigation is $cost(Q, R_Q \setminus S_k, k)$.

The cost equation (Equation 3.3) depends on the key probability terms P_T, P_R and P_c which are computed as follows.

3.2.1 Computing Probabilities

In this section, we present specific and reasonable methods to compute the probabilities used in our cost model. Depending on the specific application, other computation methods may more closely model the user. The computation of these probabilities is orthogonal to the methods and algorithms presented in Section 3.4, which are the key contributions of this chapter.

Computing P_T: This is the probability that the user finds the target object in S_k and therefore terminates the navigation. Since the target object is not known before navigation, a reasonable assumption is the probability of a potential target object is proportional to its relevance score $rel(r, Q)$. If the user finds the target object amongst the result subset S_k, then she can terminate the navigation. Therefore, we estimate the probability of termination as being proportional to the sum of relevance scores in the paginated result subset S_k and normalize it with sum of relevance scores of all the results in R_Q as $P_T = \sum_{r \in S_k} rel(r, Q) / \sum_{r \in R_Q} rel(r, Q)$. The choice of the relevance function is orthogonal to this chapter and can be computed in various ways such as TF-IDF [74] for keyword queries.

Computing P_R: The key assumption we make is that the user has a high likelihood of refining the query when the results in S_k are diverse. This is the key purpose of diversity which is to provide to the user a variety of attribute values that better represent the result set R_Q. For example, the diverse result subset in Figure 1.4(d) contains results from different categories and also contains a variety of attribute values, and hence offers many refinement opportunities, which translated to a high P_R. In contrast, the result subset of Figure 1.4(b) would translate to a low P_R.

To compute the diversity of a result subset S_k, we need to compute the distance between all pairs of results $r_i, r_j \in S_k$ [83]. Distance measures like Euclidean distance, Cosine Similarity can be used for this purpose and again the choice is orthogonal to this chapter. The diversity of S_k can be defined as $div(S_k) = \sum_{r_i, r_j \in S_k} dist(r_i, r_j)$. Hence, P_R can be computed as,

$$P_R = \frac{\sum_{r_i, r_j \in S_k} dist(r_i, r_j)}{max_{S \subseteq R_Q, |S| = k} \sum_{r_i, r_j \in S} dist(r_i, r_j)}$$

Here the denominator is used for normalization and is equal to the maximum possible diversity of a k-result set from R_Q. Finding the maximum diversity in the denominator is similar to the p-dispersion problem, and therefore known to be NP-Hard [48]. We use MMR [26] to compute the maximum diversity for our experiments (in Section 3.5).

3.3 Complexity Avalysis

We proceed with the complexity analysis of the Minimum Cost Diversification (MCD) problem. We prove that a simpler version of MCD is NP-hard, which means that MCD is also NP-hard. In particular, we consider the Minimum Cost Single Step Diversification (MCSSD) Problem, which is based on the following simplified navigation model:

Single Step Diversification Model (SSDM): In this simplified navigation model, the system shows a subset $S_k \subseteq R_Q$ of k (k is unbounded) results. Then, the user either performs a single NEXT-PAGE action, in which case the system shows all the remaining results ($R_Q \setminus S_k$) or selects one of the attribute conditions ($c : A = v$) in S_k and executes a REFINE action in which case the system shows all the results with condition ($c : A = v$). In SSDM the cost of a REFINE is 0, the cost of READ-RESULT action is 1, and that of NEXT-PAGE action is $|R_Q| + 1$ (or any other value larger than the number of results).

Minimum Cost Single Step Diversification (MCSSD): Compute a set $S_k \subseteq R$ of size k such that the expected cost based on SSDM is minimized.

Theorem 2 *MCSSD is NP-Hard.*

Proof 5 *In MCSSD, we have an initial cost k for reading the initial result subset S_k. The cost of the next step is either (a)$(|R_Q|+1)+|R_Q|-k$ in case of NEXT-PAGE or (b) $0+\#$results shown after executing the REFINE, which can be at-most $|R_Q|$. Therefore, it is always cost-efficient to perform the REFINE action. However, a result $r \in R_Q$ can be reached only if there exists a condition $c \in C(r)$ in $C(S_k)$, REFINing by which would lead to r. As shown below, to minimize the navigation cost it is sufficient to select a minimal sized result subset $S_{k'}$ of size k', which contains at-least one condition from every result in R_Q.*

Note that displaying an additional result r' in $S_{k'}$ that is already covered (i.e., $S_{k'}$ contains a condition c' of r') increases the expected cost for the following reason. Assuming all results have equal probability of being the target object, if r' is the target object, the cost saving is $\frac{1}{|R_Q|}(\#results\_with\_condition\_c' - 1) + \frac{(|R_Q|-1)}{|R_Q|}(1 - \#results\_with\_condition\_c')$, which is negative since there at least 2 results with condition c'. $\#results\_with\_condition\_c'$ represents the cost of viewing the results of REFINE on c', whereas 1 represents the cost of viewing one more result (r') in the initial S_k.

Similarly it is shown that not covering a result in the initial S_k leads to increased cost. Next, we reduce MINIMUM-SET-COVER to MCSSD.

MINIMUM-SET-COVER(U, S): Given a universe of elements $U = \{e_1, \ldots, e_m\}$ and a family $S = \{s_1, \ldots, s_n\}$ of subsets of U where each $s_i \in U$, compute a sub-family $C \subseteq S$ such that $\cup_{s_i \in C} s_i = U$ and size $|C|$ of C is minimum.

MINIMUM-SET-COVER reduces to MCSSD as follows: For each element $e \in U$, create a Boolean attribute A_e. For each set s_i in family S create a result r_i and add to r_i attribute conditions $A_c = 1$, for each $c \in s_i$, and null to the rest attributes. It is easy to verify that a solution to MINIMUM-SET-COVER translates to a solution to MCSSD and vice versa.

3.4 Adaptive Diversification

Exact Algorithm: To compute the paginated result set such that cost of navigating R_Q is minimized, it is necessary to compute the cost, using Equation 3.3, of each subset $S_k \subseteq R_Q$ of size k and selecting the subset S_k^{opt} that has the minimum cost. We show that this problem is NP-hard in Section 3.3, by reducing Set Cover to a simplified version of this problem. There are two sources of complexity that make the exact algorithm computationally expensive:

1. Computing the navigation cost of each subset $S_k \subseteq R_Q$ of size k requires evaluating Equation 3.3 for $O(|R_Q|^k)$ subsets.

2. Since Equation 3.3 is recursive, to solve it we must compute for each condition c in S_k (more formally $c \in C(S_k)$), the minimum cost (according to Equation 3.2), which in turn requires computing the minimum cost over all subsets of R_Q (Figure 3.1(a)). This process continues recursively for deeper levels of the recursion tree. The depth of the recursion is $|R_Q|$ in the worst case, since each refinement may eliminate just one result in R_Q. The width of each recursive step, i.e., the cardinality of the summation, can be up to $m.k$, which is the number of attribute values displayed at each step.

Approach overview: We attack the problem by proposing efficient techniques to approximate both sources of complexity. We first show how to effectively eliminate the recursion from Equation 3.3 using a sequence of two relaxations. Then we show how to avoid evaluating the simplified equation for every combination of result subsets using a greedy algorithm.

Our approach starts by eliminating recursion from Equation 3.3 using two relaxations. Figure 3.1 shows the recursive tree to compute Equation 3.3 and the simplifications achieved through the two relaxation steps.

Relaxation 1 **(Eliminate Conditions from Recursion Tree):** The navigation cost function (Equation 3.3) has two recursive calls - one each for REFINE and NEXT-PAGE actions, respectively to compute the navigation cost for subsequent navigation steps.

Intuitively, the navigation cost of a result-set R_Q is proportional to its size $|R_Q|$, since for a larger result-set the user must explore more results to reach the results of interest. This assumption is backed by our experiments in Section 3.5 (specifically Figure 3.3) and we use this observation to simplify the cost equation. Formally,

$$cost(Q, R_Q, k) \propto |R_Q| \qquad (o1)$$

The cost associated with REFINE actions, denoted by cost($Q \wedge c, R_{Q\wedge c}, k$), is the navigation cost incurred to reach the target results from the refined result set $R_{Q\wedge c}$. Based on the observation above, this cost is proportional to size of $R_{Q\wedge c}$, i.e.

$$cost(Q \wedge c, R_{Q\wedge c}, k) \propto |R_{Q\wedge c}| \qquad (o2)$$

Based on observations $o1$ and $o2$ and ignoring the constants of proportionality, the cost of the REFINE by a condition c can be estimated as:

$$cost(Q \wedge c, R_{Q\wedge c}, k) = \frac{(|R_{Q\wedge c}|)}{|R_Q|} cost(Q, R_Q, k)$$

Analogously, the cost of NEXT-PAGE action $(cost(Q, R_Q \setminus S_k, k))$ can approximated as $\frac{|R_Q \setminus S_k|}{|R_Q|} cost(Q, R_Q, k)$, since the user is left with $R_Q \setminus S_k$ of the original result-set R_Q after a NEXT-PAGE action.

By plugging in these approximations and rearranging terms, our cost equation can be rewritten as:

$$\begin{aligned} cost(Q, R_Q, S_k) = & S_k + (1 - P_T). \\ & \big[\alpha P_R + \beta(1 - P_R) + \\ & cost(Q, R_Q, k)\{P_R \sum_{c \in C(S_k)} P_c \frac{(|R_{Q\wedge c}|)}{|R_Q|} + (1 - P_R)\frac{|R_Q \setminus S_k|}{|R_Q|}\}\big] \end{aligned} \tag{3.4}$$

Equation 3.4 replaces the recursive calls of REFINE and NEXT-PAGE actions (in Equation 3.3) with $cost(Q, R_Q, k)$(Figure 3.1(b)). However, it still requires evaluation of all possible k-subsets of R_Q to compute $cost(Q, R_Q, k)$ according to Equation 3.2. We address this by the next relaxation.

Relaxation 2 **(Eliminate Result Subsets from Recursion Tree):** Our goal is to find the result subset S_k that minimizes Equation 3.4. But note that this same optimal S_k is used to compute $cost(Q, R_Q, k)$ according to Equation 3.2. Hence, we can replace $cost(Q, R_Q, k)$ by $cost(Q, R_Q, S_k)$ in Equation 3.4.

Then, by solving for $cost(Q, R_Q, S_k)$, Equation 3.4 can be further simplified as:

$$cost(Q, R_Q, S_k) = \frac{|S_k| + (1 - P_T).\{\alpha P_R + \beta(1 - P_R)\}}{1 - (1 - P_T).\{P_R \sum_{c \in C(S_k)} P_c \frac{(|R_{Q \wedge c}|)}{|R_Q|} + (1 - P_R)\frac{|R_Q \setminus S_k|}{|R_Q|}\}} \tag{3.5}$$

Equation 3.5 has no recursion, and can be easily computed for a given S_k. Note that Relaxation 2 does not incur any approximation error, in contrast to Relaxation 1.

Given the relaxed cost Equation 3.5, we still need to compute the cost of all possible k-result subsets S_k of R_Q to find the optimal S_k^{opt} with minimum cost.

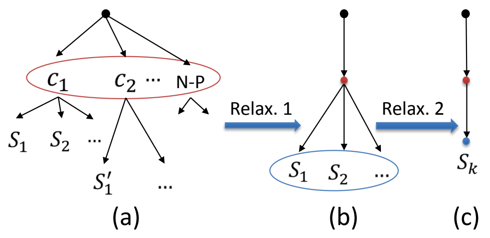


Figure 3.1: Elimination of Recursion Using Relaxations 1&2

For that, we present an efficient greedy algorithm, called Adaptive Diversification Algorithm (ADA), which incrementally builds the result set S_k by adding at each step the result with minimum incremental navigation cost. At each iteration p ($0 \leq p \leq k$), ADA makes use of two sets: the set of remaining results E and the set of selected results S_p, with $|S_p| = p$. Note that $E \cup S_p = R_Q$, the set of all the results. Initially $E = R_Q$ and $S_0 = \emptyset$. At each iteration, ADA computes $cost(Q, R_Q, S_{p-1} \cup r)$ (using Equation 3.5) for each result $r \in E$ and moves the result with minimum navigation cost to S_p. This process continues until we select k results (i.e. $p = k$).

Algorithm 10 shows the pseudo-code of our diversification algorithm ADA. The first result is chosen as the object with highest relevance score (line 3) since we want to provide to the user the most relevant object. After that, in each iteration, ADA ranks the results in E according to Equation 3.5 with S_k replaced by $S_{p-1} \cup r$, removes the result with minimum navigation cost from E, and adds it in the selected result set (line $6 - 9$). The algorithm terminates when we select k results.

Algorithm 10 $ADA(Q, R_Q)$

```
Require: Query $Q$ and Result Set $R_Q$
Ensure: Return set of $k$ results, $S_k \subseteq R_Q$
 1: $S_0 \leftarrow \emptyset$
 2: $E \leftarrow R_Q$
 3: $r \leftarrow \text{argmin}_{r_i \in R_Q} rel(r_i, Q)$
 4: $S_1 \leftarrow S_0 \cup r$
 5: $E \leftarrow E \setminus r$
 6: for $p \leftarrow 2$ to $k$ do
 7:     $r \leftarrow \text{argmin}_{r_i \in E} cost(Q, R_Q, S_{p-1} \cup r)$
 8:     $S_p \leftarrow S_{p-1} \cup r$
 9:     $E \leftarrow E \setminus r$
10: return $S_k$
```

Complexity: The running time of ADA depends on the cost computation time in line 7, which is invoked up to $O(k.|R_Q|)$ times. To compute $cost(Q, R_Q, S_{p-1} \cup r)$ using Equation 3.5, we need to calculate P_T, P_R, P_N, the cost of all possible REFINEments $(\sum_{c \in C(S_k)} P_c \frac{(|R_{Q \wedge c}|)}{|R_Q|})$ and of NEXT-PAGE action $(\frac{|R_Q \setminus S_k|}{|R_Q|})$. Computation of P_T and P_R (Section 3.2) is dominated by their denominators, which depend on the result set R_Q and k. However, in ADA we only need to compute these probabilities for the original result-set R_Q, which takes time $O(|R_Q|)$ and $O(k.|R_Q|)$(using MMR [26]), respectively. The computation of all REFINEments $cost(\sum_{c \in C(S_k)} P_c \frac{(|R_{Q \wedge c}|)}{|R_Q|})$ requires $O(k.m + |R_Q|)$ time. The cost of NEXT-PAGE and P_N can be computed in $O(1)$ time. Therefore, the total running time of ADA is $O(k.|R_Q|^2)$ (assuming $|R_Q| > k.m$). In practice, the execution time is much faster than this worst case bound (Section 3.5).

Example: Let us apply ADA to the result set in Figure 1.4(a). We are interested to find the 3 results returned by ADA. But before that we analyze Equation 3.5 more closely to infer the implication of the cost equation. The cost of a result set S_k is minimized when the denominator of the right hand side in Equation 3.5 is maximized, which implies having higher P_T value. But when the result set size $|R_Q|$ is high, we have smaller P_T value (since the denominator part of P_T in Section 3.2 is high). Therefore, the navigation cost depends on the cost of REFINE and NEXT-PAGE actions. Since the NEXT-PAGE cost (denoted as $\frac{|R_Q \setminus S_k|}{|R_Q|}$) is the same for all result sets, the navigation cost is minimized for the result set S_k containing highly diverse results with popular selective conditions (i.e., with high P_c). As $|R_Q|$ becomes smaller, P_T dominates cost equation, therefore cost is minimized for highly relevant results.

The conclusion of the above discussion is that, initially when $|R_Q|$ is large (small P_T), ADA prefers diversity over relevance. As $|R_Q|$ becomes smaller (higher P_T) in the next iterations, ADA increases preference to relevance, and provides highly relevant results.

Returning to the running example, ADA initially prefers diversity over relevance in Figure 1.4(a) since $|R_Q|$ is relatively large. The first result is the result $\#1$ as it has the highest relevance score (diversity is not a factor when selecting the first result, which is always selected by relevance). The second result would be from *Compact* or *Accessory* categories. Assuming all the conditions in *Compact* and *Accessory* categories have similar selectivity (similar P_c and similar diversity with respect to $\#1$), the second result is $\#6$ because of its high relevance score. The third result would be from the *Accessory* category to increase diversity, and specifically $\#10$ since it has higher relevance score than $\#11$. Thus ADA would return result set in Figure 1.4(d). In the next iteration, as R_Q becomes smaller, ADA will return more relevant result-set snippets like the one in Figure 1.4(c), and in the last iterations like the ones in Figure 1.4(b).

3.5 Evaluation

In this section, we describe the results of an extensive experimental evaluation of our approach. The setup, including methodology, datasets, baselines and metrics used, is described in Section 3.5.1. Sections 3.5.2 and 3.5.3 demonstrate the quality and performance of diversification algorithms in terms of estimated user navigation cost and actual user navigation time. In Section 3.5.4 we present the results of applying our algorithm to large result-sets and show that our techniques scale almost linearly with result-set size. All experiments were performed on a 2.5 GHz Intel Core $i5$ CPU, 8GB RAM machine running Windows 7. We used MySQL as our database and all algorithms were implemented in Java.

3.5.1 Setup

We evaluated our approach on two datasets:

1. **UsedCars:** This dataset consists of a listing of $15,191$ used cars, extracted from a popular car-trade website. Each tuple in this dataset has 10 attributes, 4 categorical and the rest numeric.

2. **Electronics:** This dataset consists of $65K$ products from the Electronics product catalog of a popular e-commerce website. The products were sampled from various Electronics categories, such as Laptops, Desktops, Cameras, Printers etc. and therefore the dataset is highly heterogeneous in nature. The dataset has a total of 86 (51 categorical and 35 numeric) attributes, but each product has values for a small subset (avg. 12) of these attributes and *null* for the rest.

Queries: We selected 8 queries each from the two datasets. These queries are shown in Figure 3.2 along with result-set sizes. Note that we are interested optimizing the navigation of diverse result sets, and therefore these queries were selected to be deliberately ambiguous so as to include results from a variety of categories. For that, we use single-keyword queries, although our methods support any number of keywords or query conditions; more keywords could be used if larger e-commerce datasets were available. For each query, we select a target object, which we assume the user is looking for, i.e. the navigation terminates when the user locates this target object.

Electronics			UsedCars		
Query ID	**Query**	**# Results**	**Query ID**	**Query**	**# Results**
Q_1	Kodak	193	Q_9	Honda	789
Q_2	Dell	125	Q_{10}	BMW	730
Q_3	Canon	1097	Q_{11}	2001	2034
Q_4	Nikon	511	Q_{12}	2005	920
Q_5	Camcorder	789	Q_{13}	Dallas	2932
Q_6	Speaker	737	Q_{14}	Irving	1064
Q_7	Desktop	394	Q_{15}	Black	2163
Q_8	Laptop	518	Q_{16}	Blue	1183

Figure 3.2: Query Set

State-of-the-art: Current approaches to diversification use a fixed relevance-vs.-diversity trade-off parameter (λ in Equation 3.1) to diversify rankings. However, as we argued earlier, setting this parameter is not always obvious and depends on the characteristics of the result set. In Section 3.5.2, we provide evidence to further support this claim. We compare with two commonly used approaches for ranking results:

1. **Baseline** 1 **(REL):** In this approach, the results were ranked solely based on relevance, i.e. by setting $\lambda = 0$ in Equation 3.1.

2. **Baseline 2 (MMR-**$\lambda = 0.5$**):** As a second baseline, we choose the Maximal Marginal Relevance (MMR) diversification algorithm [26]. MMR computes a diversified result-set by balancing relevance and diversity based on Equation 3.1. MMR is an approximation

algorithm since computing a diversified set based on Equation 3.1 is NP-Hard [26]. In our experiments, we set $\lambda = 0.5$ giving equal weight to diversity and relevance factors.

Note that, for a fair comparison, we used MMR both as a baseline and to compute P_R in ADA algorithm. We chose MMR since it outperforms other algorithms in terms of time and generates quality results [83]. However, our use of MMR does not preclude the use of other diversification algorithms, e.g. GMC, GNE [83], which may produce better quality results but take more time compared to MMR [83].

Next, we describe the relevance (*rel*)and diversity (*dist*) measures used in our experimental evaluation. We reemphasize that computing these measures is orthogonal to our problem and any suitable *rel* and *dist* versions can be plugged in to our approach. Due to space limitations we omit experiments with additional measures.

Computing *dist*: We use the sum of distances between the attribute values as the distance (*dist*) between two results $\left(dist(r_i, r_j) = \sqrt{\sum_{A_k \in \mathcal{A}} (r_i(A_k) - r_j(A_k))^2}\right)$. For numeric attributes, we used the Manhattan distance and for categorical attributes, the Kronecker delta function was used between the values of attribute.

Computing *rel*: In a structured result-set, the relevance of a result depends on relevance of its attribute values. We estimated the relevance of each attribute value by computing the Google Trends scores (see [29] for more details). The rationale for using Google Trends is based on the idea that the relevance of a term can be based on its frequency in a query workload.

Since results in R_Q satisfy all conditions in Q, the relevance score was computed using the unspecified attributes in $\mathcal{A}$ by Q, as was proposed by [29], where unspecified refers to an attribute that does not match any query condition. For example in Figure 1.4(a), all the records satisfy the query condition "*Camera*" with their *Product* attribute. Therefore, we compute the relevance score using the unspecified attributes (e.g. *Category*, *Brand*, *Type*).

Methodology: For each query in Figure 3.2, we picked a result $t \in R_Q$ as the target object. The chance of selecting a result as the target object is proportional to its relevance score, which means the results with high relevance scores have a higher chance to be selected as the target object. We then simulated the user navigation until target t is reached. Since multiple navigation paths can lead to the target object t, we used a randomized simulation [60] to select navigation paths. Note that, given a set of displayed results S_k, the set of conditions that can lead to t is $C(S_k) \cap C(t)$. We assumed that the user will select one of these conditions, or go to next page, according to the navigation probabilities (Section 3.2). For example in Figure 1.4(a), if the target object is $\#4$, and we select Figure 1.4(c) as the displayed result subset, the conditions that lead to $\#4$ are ”Product=Camera”, ”Category = DSLR” and ”Brand = Nikon”. The user would go to the next page if she does not like or know these three conditions. Therefore, in our simulation, we computed P_N as $\prod_{c \in (C(S_k) \cap C(t))} (1 - P_c)$ (the probability that the user would not like any condition in $C(S_k) \cap C(t)$ and P_R as $(1 - P_N)$. In case of refinement, the user could refine the query by selecting any condition in $C(S_k) \cap C(t)$. The choice of selecting a condition $c \in (C(S_k) \cap C(t))$ is proportional to P_c.

We used $k = 10$ in the experiments in Sections 3.5.2, 3.5.3, and showed the findings averaged over 1000 runs (50 random target objects, and 20 runs per target object) for each query.

3.5.2 Qualitative Analysis

In this section, we present the experimental results of the qualitative evaluation of the three different algorithms (REL, MMR and our algorithm ADA). Figures 3.3(a), 3.3(b) show the average navigation cost and average number of REFINE and NEXT PAGE actions incurred respectively by each algorithm to reach the target object for the queries of Electronics dataset. Note that, all algorithms require similar number of REFINE actions (i.e. selection of target object conditions) to filter out enough undesired objects (Figure 3.3(b)). Since REL displays results from popular categories, it requires a larger number of NEXT-PAGE actions to display the conditions of the less relevant target objects. MMR has a fixed ratio of relevance and diversity, which happens to work well for some queries with small number of results like Q_1, Q_2 and Q_4, but is ineffective for other queries like Q_3, Q_6, where more NEXT-PAGE actions are required to find the target object conditions.

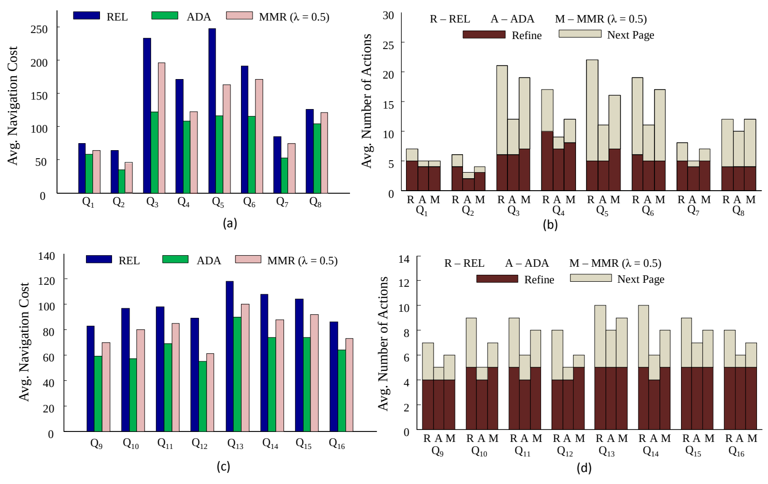


Figure 3.3: (a) Avg. Navigation Cost (b) Avg. Number of Refine and Next Page actions incurred for Electronics Dataset using $\alpha = 1$, $\beta = 1$. (c), (d) show the same figures respectively for UsedCars Dataset

ADA outperforms the other two algorithms, because of its adaptive nature. As discussed in Section 3.4, when $|R_Q|$ is high, ADA prefers diversity over relevance to pick the top-k results. As R_Q becomes more selective over iterations, ADA switches to preferring relevance. Therefore, by balancing diversity and relevance based on the result set at hand, ADA displays the target object conditions much earlier compared to the other two algorithms. This results in fewer NEXT-PAGE actions, and thus reduces the navigation cost of ADA algorithm (Figure 3.3(a)). The improvement of ADA over the other two algorithms is more pronounced for the queries that have large number of results (e.g. Q_3, Q_5, Q_6).

Figures 3.3(c), 3.3(d) show the average cost and actions respectively for the queries of UsedCars dataset. Similar to the Electronics dataset, ADA outperforms the other two algorithms for all the queries of UsedCars. Since the UsedCars dataset is homogeneous, REL and MMR perform slightly better as compared to Electronics dataset, due to less variability in attribute conditions.

We also compare the average navigation cost incurred by the three algorithms, REL, MMR and ADA, with the expected optimal navigation cost computed by solving Equations 3.2 and 3.3. Due to the exponential complexity, we compute the expected optimal cost for a smaller range sizes of initial result sets (R_Q) and query parameter (k). Figure 3.4 shows the average navigation costs for $|R_Q| = 100$ and $k = 5$ across all queries for the two datasets Electronics and Usedcars. As seen from the figure, on average, each algorithm incurs more navigation cost compare to the optimal. We see that our algorithm ADA is only 1.07 and 1.03 times worse than the optimal for the Electronics and Usedcars datasets respectively, which implies that the result sets displayed by ADA at different navigation steps are close to optimal. For MMR and REL, these factors are 1.36 (1.32) and 1.69 (1.65) respectively for the Electronics(Usedcars)dataset.

Datasets	**Average Navigation Cost (α=1, β=1)**			
	REL	**ADA**	**MMR (λ = 0.5)**	**Expected Optimal**
Electronics	48.25	30.625	38.75	28.525
UsedCars	30.5	19.15	24.375	18.512

Figure 3.4: Average Navigation Cost for Electronics and UsedCars Datasets

Figure 3.5 shows average navigation cost of MMR with increasing trade-off (λ) values (high λ value implies preference to diversity over relevance). Since ADA is independent of λ value, therefore the cost of ADA is shown as a straight line. We skipped REL since it incurs higher cost compared to ADA and MMR. As seen from Figure 3.5, there is no value for λ that is optimal for a given datasets or even for a particular query. Intuitively λ should change adaptively, at each navigation step, depending on the characteristics of the result set. By balancing the relative importance of relevance and diversity adaptively at each step, ADA shows better performance (on average) compared to MMR with a fixed λ.

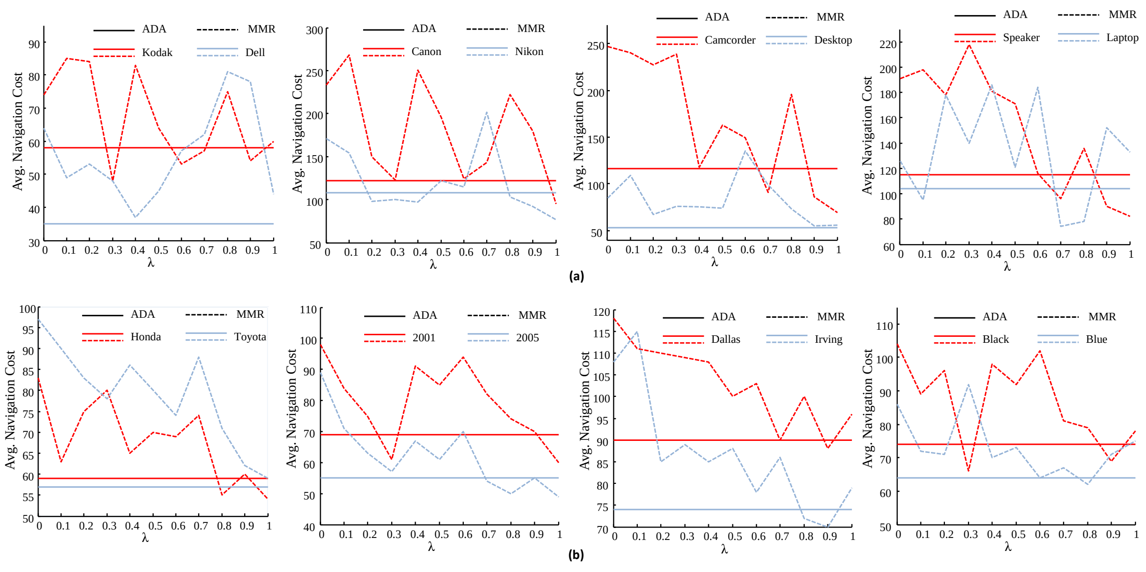


Figure 3.5: Average Navigation Cost vs. Tradeoff (λ) values for (a) Electronics Dataset (b) UsedCars Dataset ($\alpha = 1$, $\beta = 1$)

Experiments varying model parameters (α, β)**:** We also compared the average cost incurred by each algorithm while varying the cost NEXT-PAGE actions (hitherto assumed to be unit). Intuitively, a user reads several results and its conditions before deciding whether to go to the NEXT-PAGE, instead of selecting a condition to REFINE by, and therefore these actions, captured in our model by β, should have a higher cost than those of REFINE or reading a result. We evaluated the algorithms with higher values of NEXT PAGE action and results of these experiments are displayed in Figure 3.6. As expected, the overall navigation cost increases with the increased β value. Since REL and MMR do not consider β in selecting results, they suffer a higher increase in navigation cost. But ADA adapts its actions, displaying a diverse set of results and conditions based on avoiding the costly NEXT-PAGE actions, and therefore the overall cost increases at a slower rate.

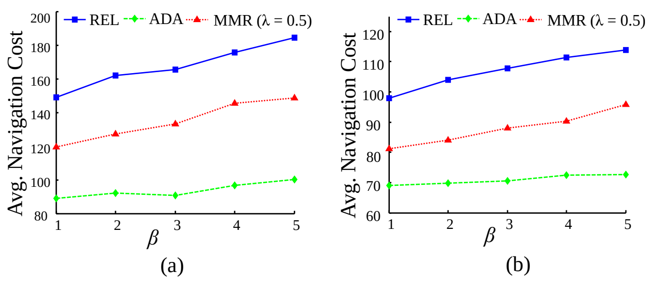
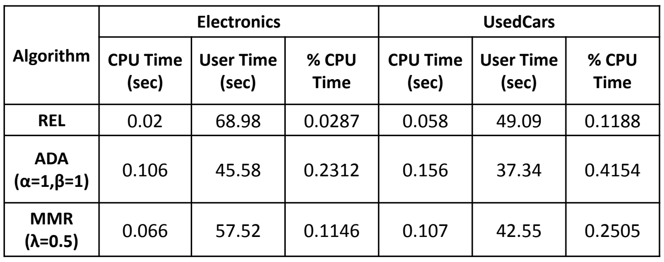


Figure 3.6: Average Navigation Cost vs. β for (a) Electronics Dataset (b) UsedCars Dataset ($\alpha = 1$)

3.5.3 Performance Analysis

We now present the performance results of REL, MMR and our ADA algorithms. Figure 3.7 shows the average (across all queries) computation (CPU) times taken by each of these algorithms to reach the target object. As expected, the relevance-only algorithm (REL) takes the shortest time among all the algorithms. Computing diversity is a costly operation since it involves computation of distance between all pairs of results. As a result any algorithm that incorporates diversity is much slower as compared to REL; our implementation of MMR is three times slower as REL. Our algorithm (ADA) performs this distance computation over all future navigations and therefore is slower than MMR by a factor of 1.6. While ADA takes more time to compute the set of paginated results, it is very effective in reducing the time or effort incurred by users to navigate such diverse result sets, as shown in Figure 3.7. To calculate user times we mapped the navigation cost obtained during simulation to the time taken by users to perform the actions associated with a given navigation. In Section 3.6 we present the results of the user study that we conducted at Amazon Mechanical Turk and show that cost is linearly proportional with navigation time. More specifically, the relation between navigation time and cost is expressed as, $time = 0.39 * cost + 10.92$ (the trend line in Figure 3.12). Therefore, given the overall navigation cost, we used this formula to calculate the overall navigation time. As seen in Figure 3.7, ADA significantly improves the user navigation time as compared to REL and

MMR. The improvement is by a factor of 1.26 and 1.51 over MMR and REL respectively for the Electronics dataset, and 1.14 and 1.31 over MMR and REL respectively, for the UsedCars dataset. This represents a significant improvement since the user navigation time is orders of magnitude greater than the computation time.



Algorithm	Electronics			UsedCars		
	CPU Time (sec)	User Time (sec)	% CPU Time	CPU Time (sec)	User Time (sec)	% CPU Time
REL	0.02	68.98	0.0287	0.058	49.09	0.1188
ADA (α=1,β=1)	0.106	45.58	0.2312	0.156	37.34	0.4154
MMR (λ=0.5)	0.066	57.52	0.1146	0.107	42.55	0.2505

Figure 3.7: Average CPU, User Navigation Time for Electronics and UsedCars Datasets

3.5.4 Scalability Analysis

We next examine the scalability of the algorithms (for Electronics dataset) in terms of navigation cost and computation time while varying the size of initial result set R_Q (with $\alpha = 1$, $\beta = 1$). For this experiment, we randomly choose 20 different queries from each dataset as our query set.

Figure 3.8(a) shows the average navigation cost incurred by the three algorithms for different initial results set size (500 to 5000) over all the queries of Electronics dataset. The average number of REFINE and NEXT-PAGE actions are also shown on top of the bars. As seen from the figure, our algorithm (ADA) outperforms the other two approaches by a significant margin in all cases. As the result set size $|R_Q|$ increases, this margin also increases. This justifies the effectiveness of our adaptive diversification algorithm (ADA) in handling large amount of results, compared to the state-of-the-art approaches.

Figure 3.8(b) shows the average computation time taken by the three algorithms per REFINE or NEXT-PAGE action. As described in Section 3.5.4, ADA considers all future navigations while computing the result set in a single iteration, thus slower than the other two algorithms. Since the difference is in tens of milliseconds, the readers might consider this as a small amount of price paid for the significant improvement achieved by the ADA algorithm in terms of user navigation cost, and also user navigation time.

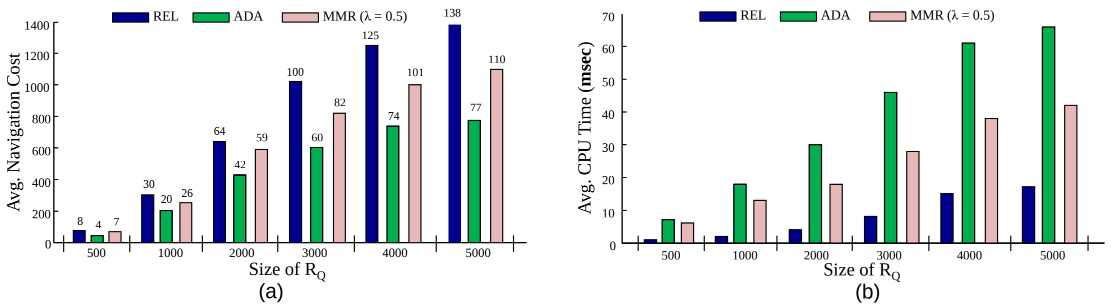
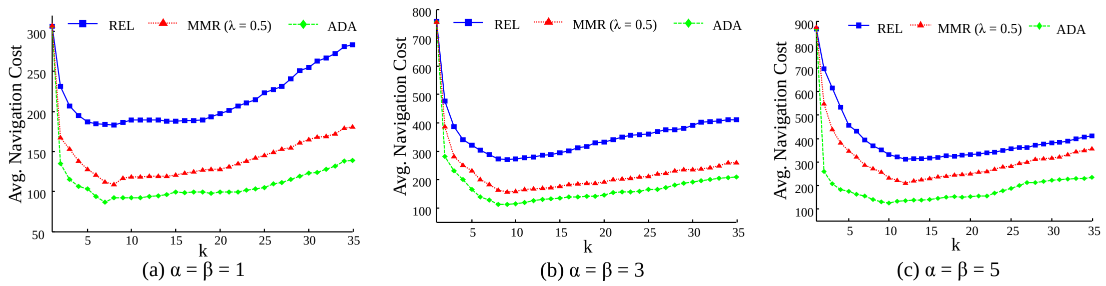


Figure 3.8: For Electronics Dataset, (a) Average Navigation Cost, Average number of REFINE and NEXT PAGE actions (numbers on top of bars), vs. Size of R_Q (b) Average CPU Time vs. Size of R_Q (for $\alpha = 1$, $\beta = 1$)

Varying the Page Size: We also explore the changes in navigation cost while varying the page size (k value). As seen in Figure 3.9, when $k = 1$ the user needs to execute larger number to NEXT-PAGE and READ-RESULT actions to reach the target object, resulting to higher navigation cost. As the k value increases, the chances of getting the target object conditions in a particular page increases. As a result, the user tends to execute more REFINE actions, resulting to fewer READ-RESULT and NEXT-PAGE actions, therefore decreases the overall navigation cost. If we continue to increase the k value, the effect of increasing the page size starts to play a negative role after a certain k, because the user must read many results before refining (our model assumes that the user reads all page results before taking an action). Therefore the total number of READ-RESULT actions, as well as the overall navigation cost increases.

Figure 3.9 shows the average navigation cost incurred by the three algorithms for Electronics dataset, while changing k value from 1 to 35, and REFINE and NEXT PAGE costs, α and β, from 1 to 5 ($\alpha = \beta$), which are the values used in previous user navigation work [60]. Since all the algorithms picked the most relevant object as the first result, they have identical performance for $k = 1$ (no diversity employed). For all other k values, ADA clearly outperforms the other two approaches. The experiments in Figure 3.9 are useful to find the optimal k value for a particular algorithm in a given dataset (e.g. in Figure 3.9(a) the optimal page size of our approach ADA is 7 for Electronics dataset). Also, moving from left to right (from a to c) in Figure 3.9, we observe that the optimal page size increases with increased α and β value. The reason is that as the overhead of a user click action (REFINE and NEXT-PAGE) increases, longer result pages are more effective.



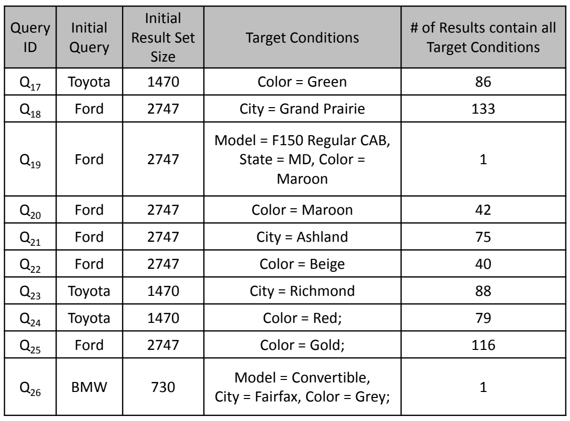
(a) α = β = 1 (b) α = β = 3 (c) α = β = 5

Figure 3.9: For Electronics Dataset, Average Navigation Cost vs. k

3.6 User Study

In this section, we present the results of a user study that we conducted at Amazon Mechanical Turk (MTurk) [1] using the UsedCars dataset. We selected three keyword queries (e.g., 'Ford'), and for each query we created a set of search tasks; each search task specifies a set of target conditions (e.g., find a car with Color = Green). We asked the users, starting from the results of the initial keyword query, to find the best car (according to the relevance score defined in Section 3.5.1) that satisfies all the target conditions.

We repeated the experiment for the four different ranking algorithms: REL ($\lambda = 0$), MMR ($\lambda = 0.5$), ADA (λ-independent) and a diversity-only baseline, DIV, which constructs the k-result subset greedily at each step by maximizing the score function (Equation 3.1); i.e. with $\lambda = 1$. The reason that we asked users to find the best and not any result is to avoid giving an unfair advantage to methods biased towards diversity like DIV. Such methods may help the user to find a satisfying result, but this result may have low relevance. Figure 3.10 shows the list of initial queries, their results' cardinality, the target conditions, and the cardinality of results that satisfy all the target conditions. The page size (k) is set to 10. Each task was completed by 36 MTurk users; we present the average results.



Query ID	Initial Query	Initial Result Set Size	Target Conditions	# of Results contain all Target Conditions
Q_{17}	Toyota	1470	Color = Green	86
Q_{18}	Ford	2747	City = Grand Prairie	133
Q_{19}	Ford	2747	Model = F150 Regular CAB, State = MD, Color = Maroon	1
Q_{20}	Ford	2747	Color = Maroon	42
Q_{21}	Ford	2747	City = Ashland	75
Q_{22}	Ford	2747	Color = Beige	40
Q_{23}	Toyota	1470	City = Richmond	88
Q_{24}	Toyota	1470	Color = Red;	79
Q_{25}	Ford	2747	Color = Gold;	116
Q_{26}	BMW	730	Model = Convertible, City = Fairfax, Color = Grey;	1

Figure 3.10: Query Set for User Study

Figures 3.11(a) and 3.11(b) show the average time taken and average number of actions executed respectively, by users to find the best target object. As seen in Figure 3.11(b), if we have more target conditions (e.g. Q_{19}, Q_{26}), using DIV (diversity-only), the chances of getting a desired target condition on a given page increases. This increases the probability of REFINE action and, therefore, DIV performs better (Figure 3.11(b)) than the other two baselines REL, MMR, and slightly worse that our algorithm ADA, which prefers diversity over relevance during the initial steps. If we decrease the number of target conditions, the performance of DIV degrades, especially if multiple results satisfy all target conditions (as seen for the other queries), since the user needs to find the best and not any object. For Q_{17} and Q_{24}, MMR ($\lambda = 0.5$) and ADA perform similarly, which intuitively shows that 0.5 happens to be the ideal balance between relevance and diversity for these two queries. This is clearly not that case for other queries such as Q_{23}, where MMR takes longer time even compared to REL.

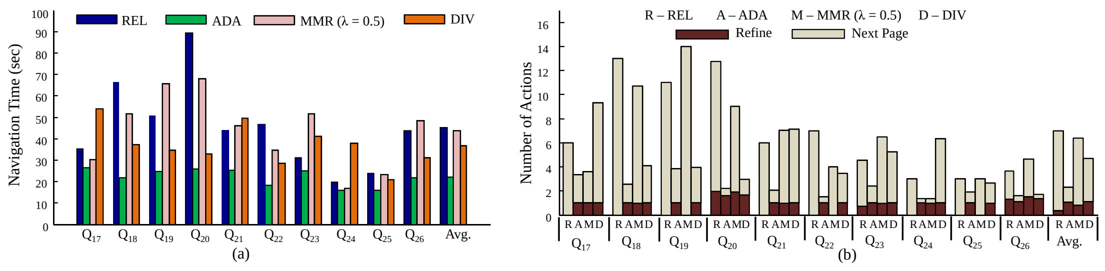


Figure 3.11: (a) Avg. User Navigation Time, (b) Avg. number of Refine and Next Page Actions incurred by the users for 10 different queries using UsedCars dataset (for $\alpha = 1$, $\beta = 1$)

As shown by the average values in Figure 3.11(a), ADA reduces the navigation time significantly compared to all other algorithms. On average, ADA is faster by a factor of 2.04 (p-value 0.004), 1.97 (p-value 0.001) and 1.66 (p-value 0.0002) over REL, MMR and DIV,respectively. This significant improvement is because of the smaller number of actions incurred by ADA compared to the other three algorithms (Figure 3.11(b)).

Figure 3.12 shows the actual user navigation time vs. the estimated cost (using our cost model in Section 3.2) for the 10 different queries using the four different algorithms, where for each query we average over all users. The figure and the trend line show a clear correlation, and specifically a linear relationship, which confirms the validity of the cost model.

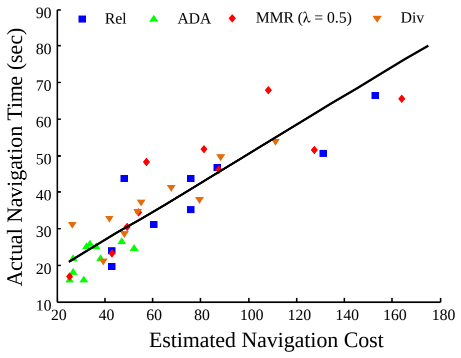


Figure 3.12: Estimated Cost vs. Actual Time

3.7 Related Work

Ranking based on relevance has been investigated and applied in search engines to provide the user top relevant results. [29] uses data and workload statistics and correlations, and apply probabilistic IR models to rank the results. Various other strategies have been proposed in literature to rank the results in keyword search [56][13][11]. Since top relevant objects come from the popular (top) categories, ranking solely based on relevance is not useful to minimize navigation cost, especially for the users interested in less popular objects.

Diversification has recently been introduced in search engines to increase user satisfaction. Various approaches have been developed to diversify search results in different domains. [81] introduces diversity ordering for the attributes of structured data and selects results that are diverse according to the ordering. [52] extends k-nearest neighbors to find the diverse results that are significantly different from each other. [12][63] use coverage approaches to cover diverse aspects of search space. [43] expresses the degree of diversification by a setting a parameter which also determines the size of final result set. [12] operates on Web documents and selects diverse documents to cover many different interpretations of the query. [63] proposes document summarization that highlights different concepts of a document. Diversification has also been proposed in recommendation systems. [84] provides diversity using the content of the recommendations and the past history of the user. [22] presents a method based on medoids clustering to select a set of diverse and highly-ranked items to recommend to a user. Topic diversification using personalized lists in recommendation system has been explored in [89]. Several content based diversifications have been addressed in [44][41]. However, none of these approaches considers the problem of how to minimize the total navigation cost incurred by the user to find the target object by considering subsequent navigation steps. In this chapter, we have addressed this concern by introducing a navigation cost model described in Section 3.1.

Diversification is being used along with *relevance* in [33][39][83][26][41]. Most of these approaches (e.g. [83]) consider diversification as a bi-criteria optimization, which uses a fixed trade-off value for relevance and diversity. In Section 3.5, we have shown that different search tasks require different ideal trade-off value. Threshold based techniques, a variant of the optimization problem, have been proposed in [69][88] to solve the diversification problem. These approaches consider a threshold value of relevance and maximize the diversity between results (or vice versa). Setting a threshold value is hard, and depends on the domains. Our approach does not require the threshold value, and can adaptively set the balance between relevance and diversity for different tasks.

Faceted Search has been shown to be effective in reducing the user effort and time required to navigate large result sets of structured databases. For a given query, these approaches compute the best facet conditions and matching results to display to the user [60][73][27]. However, this model is not suitable for our setting of limited screen size, where we cannot display separately faceted conditions and results. Instead, our results serve a dual purpose, since a user can click on results conditions to refine her navigation.

Navigation Modeling of Search Results has been a subject of intense research in recent years and several methods of reducing user effort have been proposed [73][76][67]. In [73], a navigation model based on a minimum cost decision tree is proposed to minimize navigation effort. [76] proposes a method of rapidly skimming through the results of the query by showing representative tuples from the query result, where the representative tuples are chosen by clustering the results. In contrast, our approach balances relevance and diversity in a principled manner by considering navigation cost. In [67], result navigation based on multi-criteria optimization problem that balances relevance and result set coverage (skyline) is proposed.

Chapter 4

Distributed Diversification

In this chapter, we present the first distributed solution for diversifying large datasets using the MapReduce framework [54]. We propose two approaches for distributed diversification; one optimized for the disk access cost while the second optimized towards the network transfer cost. We present a cost model that can dynamically choose the suitable approach considering the environment parameters (disk rate, network speed, number of cluster nodes) and data size. We also propose an approach to improve the quality of the diversification by iteratively refining the output. The final output is a 2-approximation over the optimal solution.

The rest of the chapter is organized as follows: Section 4.1 describes the problem. Section 4.2 explains the core components of diversification (i.e. diversification approaches, cost model, iterative refinement). Section 4.3 provides an experimental evaluation of the components using two real life datasets. Finally, Section 4.4 describes the related work.

4.1 Problem Definition

Consider a set of n elements $\mathcal{D} = \{e_1, e_2, \ldots, e_n\}$, a query Q and an integer k $(\leq n)$. Each element $e_i \in \mathcal{D}$ has a *relevance* score, $rel : \mathcal{D} \rightarrow \mathbb{R}^+$, to the query Q, where higher relevance score implies the element e_i is more *relevant* to the query Q. The *dissimilarity* between two elements e_i, e_j is defined by the function, $dis : \mathcal{D} \times \mathcal{D} \to \mathbb{R}^+$, where a higher score implies that the elements e_i and e_j are highly *dissimilar* to each other. Our goal is to find a set of k elements, $S_k \subseteq \mathcal{D}$, such that the elements in S_k are highly *relevant* to the query Q and highly *dissimilar* to each other.

Most previous works define top-k diversification as a bi-criteria optimization problem: to each k-element subset $S_k \subseteq D$ a score $\mathcal{F}(S_k)$ is assigned, using both the *relevance* and *dissimilarity* of the elements in S_k. In particular, let $d : \mathcal{D} \times \mathcal{D} \to \mathbb{R}^+$ be the distance metric defined as,

$$d(e_i, e_j) = (1 - \lambda)\, \frac{rel(e_i) + rel(e_j)}{2} + \lambda\, dis(e_i, e_j) \tag{4.1}$$

where, the trade-off parameter, $\lambda \in [0, 1]$, balances the relative weights between *relevance* and *dissimilarity* [83]. Then $\mathcal{F}(S_k)$ is the sum of all pairwise distances between the elements in S_k, namely:

$$\mathcal{F}(S_k) = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} d(e_i, e_j) \tag{4.2}$$

Problem 1 (Top-k Diverse Elements) *Given $\mathcal{D}$, Q and k identify the subset S_k for which $\mathcal{F}(S_k)$ is maximum.*

4.2 Diversification Framework

There are three main components in our diversification framework (Figure 4.1): 1) the *Diversification Stage*, 2) the *Cost Analysis*, and, 3) the *Iterative Refinement*. The *Diversification Stage* component reads data from HDFS and generates the set S_k using a diversification algorithm. In Section 4.2.1 we propose two distributed diversification approaches for this component. In Section 4.2.2 we present a cost model to estimate the execution time of each diversification approach given the environment parameters and data characteristics. Using this model, the *Cost Analysis* component chooses the best approach dynamically at runtime. Finally, the *Iterative Refinement* component iteratively refines the set S_k returned from the diversification stage until it either converges (no further score improvement) or a user time threshold is reached (Section 4.2.3).

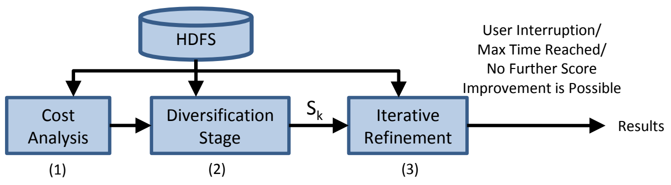

HDFS
Cost Analysis
(1)
Diversification Stage
(2)
S_k
Iterative Refinement
(3)
User Interruption/ Max Time Reached/ No Further Score Improvement is Possible
Results

Figure 4.1: The Diversification Framework Architecture

4.2.1 Diversification Approaches

In the rest of the chapter, we assume a distributed MapReduce framework with m mappers and r reducers. Previous works [35][45] on designing algorithms in a MapReduce framework generally consider the following approach: First the data is divided (mapped) into partitions and each partition is assigned to a single node. Each node solves (reduces) the problem on the assigned partition and generates an output. Then, all outputs are merged together in single node which produces the final output. Our first approach, the **D**ivide and **M**erge based diversification (DM), follows a similar strategy. This approach is disk I/O efficient since it reads the whole data only once from the disk. However, it incurs high network cost to send all the elements through the network for partitioning.

To reduce the network cost, recently another approach, namely "sample and ignore", has been proposed for the problem of clustering large data [35]. It reduces the network cost by first finding the major clusters from a sample of the input data and then ignoring the elements that are contained in the major clusters from passing through the network. The sampling idea is useful for our purposes as it maintains the charasteristics of the dataset; however, we replaced the “ignore” phase with a novel “refine” phase which reduces the network cost significantly compared to the “ignore” phase by sending only k results from each node. This resulted to the **S**ample and **R**efine based diversification (SR) approach.

Note that both of our approaches use a uniprocessor diversification algorithm as a plug-in when each single node performs diversification on its local element set. The choice of this diversification algorithm is independent with our framework. Therefore, any of the algorithms proposed in [81][83][26] can serve the purpose.

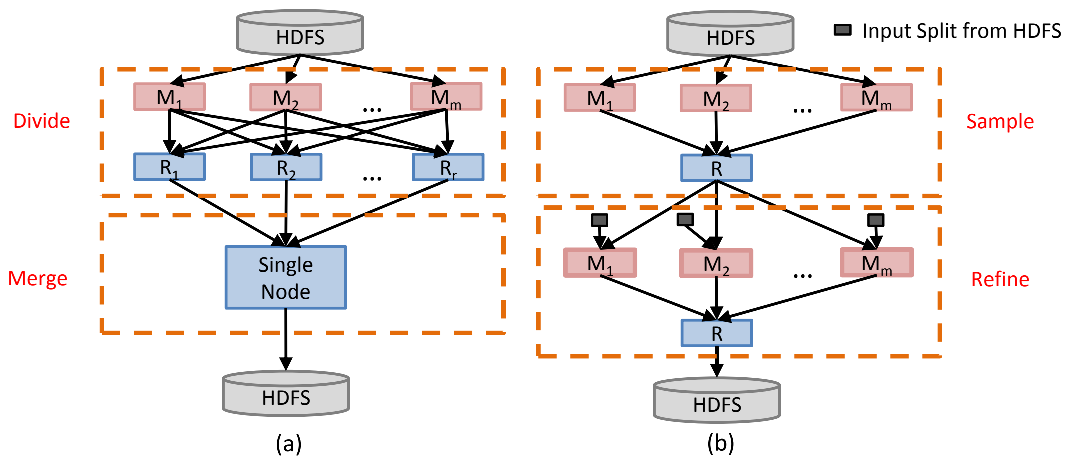


Figure 4.2: Overview of (a)DM and (b)SR Diversification Approaches

4.2.1.1 Divide and Merge (DM)

As the name implies, there are two phases, *divide* and *merge* (Figure 4.2(a)). In the *divide* phase, data is partitioned randomly to different nodes maintaining a balanced load. Each node executes the uniprocessor diversification algorithm on the assigned partition and generates a top-k diversified subset from its own partition. One key assumption in this phase is that each node has enough memory to store its assigned partition (the case that this does not hold is considered in Section 4.2.3). In the *merge* phase, all k-diverse results generated by different nodes are merged in a single node to compute the overall top-k diverse results.

The *divide* phase is implemented by a single MapReduce job. In the map phase, each map task reads a split (block) of $\mathcal{D}$ from HDFS. For each element in the file split, it outputs the pair $< key, element >$. The key denotes the ID of the reducer (between 1 to r). In the shuffle phase, each reducer gets the pairs with the same key, and pairs with distinct key values are forwarded to distinct reducers to be processed separately. In the reduce phase, each reducer executes the uniprocessor diversification algorithm on the assigned elements and generates k-diverse elements. Note that, the value of k is relatively small (in tens) while the value of r is assumed in the hundreds. Therefore, the total number of merged elements (rk) is small enough to fit in the memory of a single node at the *merge* phase. This node executes the uniprocessor diversification algorithm on the rk elements and generates the overall k diverse elements S_k.

4.2.1.2 Sample and Refine (SR)

Although the DM approach reads the input data only once in its divide phase (therefore, is disk I/O efficient), it sends all the elements through the network for partitioning. For a slow network, this might cause a bottleneck. Instead, the SR approach reduces the network load significantly by sending a small subset of the elements through the network.

SR (Figure 4.2(b)) also works in two phases, namely, *sample* and *refine*. In the *sample* phase, each mapper reads a split of $\mathcal{D}$ from HDFS and selects a small random sample from the split. Let α be the sampling ratio. A single reducer collects all the samples, executes the uniprocessor algorithm on the sampled elements and computes the k diverse elements S'_k. Note that, only the selected samples need to be shuffled through the network. Thus the SR algorithm reduces the network cost significantly compared to the DM algorithm.

The key challenge of the *sample* phase is to select a good representative sample from each mapper's file, such that the single node that diversifies the samples, can still produce a good k-diverse result. Since the quality of the diversification depends on the score $\mathcal{F}(S'_k)$

(higher score means more diversified result), we investigate the effect of α on the diversification score. Figure 4.3 shows the $\mathcal{F}(S'_{10})$ score of top-10 diverse tweets computed from 10 million tweets[10] by varying the sampling ratio α. The MMR uniprocessor algorithm [26] was used for diversification. For small k, a sample of about 1% is good enough. However, for higher k, a larger sample is needed (around 30% for $k = 25$).

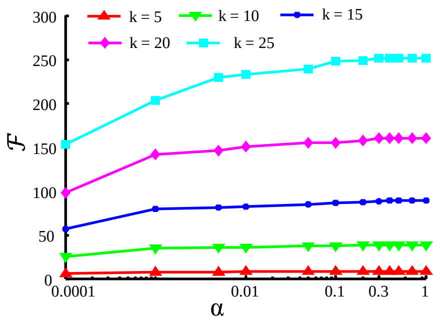


Figure 4.3: Effect of α on $\mathcal{F}$

Note that the SR approach assumes that all samples taken from the mappers will fit in the memory of the single reducer node. If this is not the case (assuming that a limited amount of memory is available for the diversification task), the sampling rate needs to be adjusted, thus reducing the quality of the sample. This motivates us to further refine the k diverse element set (S'_k) generated in the *sample* phase by a novel *refine* phase using the elements in $\mathcal{D}$.

In the *refine* phase, S'_k is broadcasted to all mappers. Each mapper reads a split of data from the disk and tries to refine S'_k using the elements from the split. Since in MapReduce framework the mapper works on a single element at a time, we use the swap strategy [84] for refinement. Each element e in the partition is checked against all the elements e_i in S'_k to see if there exists a replacement operation, e for e_i, that can improve the quality of S'_k. If there exists multiple such operations, e replaces the element e_i in S'_k that improves the quality the most. When all elements in the split are checked, the refined k-diverse element set is forwarded to a single reducer. This reducer combines all refined element sets returned from different mappers, executes the uniprocessor algorithm on the combined element set and computes the final k-diverse results. Note that, the total number of elements shuffled in the *refine* phase is mk. The value of k is in tens (as discussed before) and the value of m is in hundreds depending on the split size and data size. Therefore, the network cost of the *refine* phase is negligible compared to the DM algorithm.

Algorithm 11 shows the pseudocode of the SR algorithm in high level. The sample phase corresponds to the lines (2-3). The refine phase (line 5) is further elaborated by Algorithm 12. In the map phase of Algorithm 12 (lines 2-5), each mapper works on a split of $\mathcal{D}$ and call the subroutine $get\_refined\_set$ for each element in the file split. When all the elements are processed, the refined element set is forwarded to a single reducer. Finally, one reducer merges the refined element sets and computes the final k-diverse results (line 7).

Algorithm 11 $SR(\mathcal{D},k,\alpha)$

```
Require: Element Set $\mathcal{D}$, value of $k$ and Sampling Ratio $\alpha$
Ensure: Return $k$ diverse results
1: // sample
2: In parallel, each mapper reads a split of $\mathcal{D}$ from HDFS, selects some elements with probability $\alpha$ and sends the elements to a single reducer
3: one reducer gets the elements, executes the uniprocessor algorithm and produces a k-diverse results, $S'_k$
4: // refine
5: $S_k \leftarrow Refine(\mathcal{D},S'_k)$
6: return $S_k$
```

Algorithm 12 $Refine(\mathcal{D},S_k)$

```
Require: Element Set $\mathcal{D}$ and $k$-element set $S_k$
Ensure: Return refined $k$ diverse results
1: // map
2: In parallel, each mapper reads a split of $\mathcal{D}$ from HDFS and do the following
3: for each element $e$ in the file split do
4:     $S_k \leftarrow get\_refined\_set(e, S_k)$
5: sends $S_k$ to a single reducer
6: // Reduce
7: one reducer gets the refined result sets, executes the uniprocessor algorithm, and produces k-diverse results $S_k$
8: return $S_k$
```

Algorithm 13 $get\_refined\_set(e, S_k)$

```
Require: an element $e$ and $k$-element set $S_k$
Ensure: Return $k$ diverse results
1: $S'_k \leftarrow S_k$
2: for each element $e_i$ in $S_k$ do
3:     if $\mathcal{F}(\{S_k - e_i\} \cup e) > \mathcal{F}(S'_k)$ then
4:         $S'_k \leftarrow \{S_k - e_i\} \cup e$
5: if $\mathcal{F}(S'_k) > \mathcal{F}(S_k)$ then
6:     $S_k \leftarrow S'_k$
7: return $S_k$
```

In Section 4.3, we show that the quality of the diversified result set produced by each of our approaches, DM and SR, matches the quality of the diversified result set that a uniprocessor diversification algorithm would produce (if it was fed the full data set). Furthermore, in Section 4.2.3 we theoretically prove that combined with some *refine* phases, both of our algorithms produce a diversified result set whose score is 2-approximate to the score of the optimal diversified result set.

4.2.2 Cost Model

The performance of the two approaches depends on various parameters of the distributed environment (disk speed, network speed, number of nodes etc.) For example, Figures 4.7(a), 4.7(c) show the wall clock time needed to compute top-10 diverse results (for twitter[10] and image[9] datasets respectively), by varying the number of reducers. As seen from these experiments, the SR approach performs better for smaller number of reducers while the DM dominates as the number of reducers increases. Ideally, depending on the environment parameters and data characteristics, we would like to choose the best diversification approach. We thus proceed with a cost model that captures the disk I/O, network I/O and CPU cost for DM and SR.

Our cost model is developed on a standard deployment of Hadoop 1.0.4 for massive data. Therefore, we do not assume data piping from memory to memory, instead, pessimistically assume disk is being used in between the mappers and reducers. Table 4.1 summarizes the pa-

Symbols	Definitions
$F_{\mathcal{D}}$	File Size of $\mathcal{D}$ in bytes
D_r	Disk Rate in bytes/sec (average read/write)
N_r	Network Rate in bytes/sec
α	Sampling Ratio
β	Dispersion Ratio
m	Number of Mappers
r	Number of Reducer

Table 4.1: Cost Model Parameters

rameters used. Using an approach similar to [35] we first model the cost of a single MapReduce job which is the sum of costs for the Map and Reduce phases.

Map Cost: Let the cost to start m mappers be $delay(m)$, where $delay(1)$ denotes the time required to start a single task (map/reduce). To read $F_{\mathcal{D}}$ bytes from the disk by m mappers incurs $\frac{F_{\mathcal{D}}}{m.D_r}$ cost. On average, each mapper gets $\frac{F_{\mathcal{D}}}{m}$ bytes. Let $costP(\frac{F_{\mathcal{D}}}{m})$ be the cost to process the $\frac{F_{\mathcal{D}}}{m}$ bytes by each mapper. A factor of α bytes are picked during the processing. To spill the sampled bytes to disk each mapper takes $\frac{\alpha.F_{\mathcal{D}}}{m.D_r}$ cost. Note that, we ignore the additional bytes required to store the key part in the output pairs since this is negligible compared to input bytes $\alpha.F_{\mathcal{D}}$. Therefore the cost to execute the map phase, $costM(F_{\mathcal{D}}, m, \alpha)$, is:

$$\begin{aligned} costM(F_{\mathcal{D}}, m, \alpha) =& delay(m) + \frac{F_{\mathcal{D}}}{m.D_r} + costP(\frac{F_{\mathcal{D}}}{m}) \\ &+ \frac{\alpha.F_{\mathcal{D}}}{m.D_r} \end{aligned}$$

Reduce Cost: In the reduce phase, the data stored in the mapper local disks are copied in the reducer memory. To read $\alpha.F_{\mathcal{D}}$ bytes from m mappers' local disks takes $\frac{\alpha.F_{\mathcal{D}}}{m.D_r}$ cost. Note that, a fraction of these $\alpha.F_{\mathcal{D}}$ bytes are shuffled through the network to reach the other cluster nodes running reduce tasks. Let β is the dispersion ratio, denotes the fraction of mapper output are shuffled though the network. To shuffle $\beta.\alpha.F_{\mathcal{D}}$ bytes to r reducers requires $\frac{\beta.\alpha.F_{\mathcal{D}}}{r.N_r}$ cost.

Once the data is copied in reducer memory, the uniprocessor diversification algorithm is executed on the data. On average each reducer gets $\frac{\alpha.F_{\mathcal{D}}}{r}$ bytes. Let $costD(\frac{\alpha.F_{\mathcal{D}}}{r})$ be the cost to execute the uniprocessor algorithm on $\frac{\alpha.F_{\mathcal{D}}}{r}$ bytes. Finally the output records in written in HDFS. Since each reducer needs to write only k elements, which is small in size, thus the cost to write the output is ignored.

Therefore, the cost to execute the refine phase, $costR(F_{\mathcal{D}}, m, r, \alpha)$, is defined as,

$$\begin{aligned} costR(F_{\mathcal{D}}, m, r, \alpha) =& delay(r) + \frac{\alpha.F_{\mathcal{D}}}{m.D_r} + \frac{\beta.\alpha.F_{\mathcal{D}}}{r.N_r} \\ &+ costD(\frac{\alpha.F_{\mathcal{D}}}{r}) \end{aligned}$$

Note that, in both the DM and SR approaches, each reducer gets the elements with the same key. Therefore, the sorting time is negligible in the reduce phase.

DM Cost: In the *divide* phase, m mappers read the data and do the partitioning with cost $costM(F_{\mathcal{D}}, m, 1)$. The reducers execute the uniprocessor algorithm with cost $costR(F_{\mathcal{D}}, m, r, 1)$. Note that, in the *merge* phase, one single machine runs the uniprocessor algorithm on $r.k$ elements. Since the value of $r.k$ is relatively small (discussed in Section 4.2.1), the cost associated with the merging phase is ignored from calculation. Therefore, the cost to execute the DM algorithm is:

$$costDM = costM(F_{\mathcal{D}}, m, 1) + costR(F_{\mathcal{D}}, m, r, 1) \tag{4.3}$$

SR Cost: The cost to execute SR algorithm consists of the cost associated with the two phases, *sample* and *refine*. In the *sample* phase, m mappers do the sampling with cost $costM(F_{\mathcal{D}}, m, \alpha)$. One reducer processes the remaining data with cost $costR(F_{\mathcal{D}}, m, 1, \alpha)$. In the *refine* phase, m mappers do the refinement with cost $costM(F_{\mathcal{D}}, m, 1)$. Note that, in the map phase, each mapper outputs k elements. Thus the single reducer gets a total $m.k$ elements by m mappers; since this is small, the cost associated in diversifying these mk elements is ignored. Hence, the cost to execute the SR algorithm is:

$$costSR = costM(F_{\mathcal{D}}, m, \alpha) + costR(F_{\mathcal{D}}, m, 1, \alpha) + costM(F_{\mathcal{D}}, m, 1) \tag{4.4}$$

Using Equations 4.3 and 4.4, the cost analysis component in Figure 4.1 estimates the execution times of two diversification approaches DM, SR respectively, and picks the best one at runtime.

4.2.3 Iterative Refinement

Our DM approach assumes that in the *divide* phase each reducer has enough memory to store the assigned partition. However, in case of limited memory, each reducer gets a sample of the partition, thus reduces the quality of diversification (Figures 4.9(a)) by the DM approach. Therefore, we propose an iterative refinement component that further refines the output of the DM approach and guarantees a 2-approximate solution compared to the optimal. Note that this component is also applicable after the SR approach to guarantee the 2-approximate solution.

The iterative refinement component works as a plug-in after the diversification stage and iteratively improves the quality of the k-diverse elements returned by the diversification approaches. Each iteration corresponds to a single mapreduce job and does exactly the same task like the *refine* phase in SR algorithm. This process continues until no further score improvement is possible or a user threshold time is reached. Also note that, the user can stop the execution at any point in the iterative refinement to get the best result set produced at the elapsed period of time. This ensures a possible adaptation of our approach as an *anytime* algorithm.

Algorithm 14 ($DivF$) describes the overall workflow of our diversification framework. At first, the cost analysis component computes the cost of our two diversification approaches DM and SR (line 1). Based on the costs, the best approach is executed in diversification stage (lines 2-5). Finally the iterative component refines the k diverse set returned from diversification stage (lines 6-11).

Algorithm 14 $DivF(\mathcal{D},k,\alpha)$

```
Require: Element set $\mathcal{D}$ and size of $k$, Sampling Ratio $\alpha$
Ensure: Return set $S_k \subseteq \mathcal{D}$ of size $k$
 1: compute $costDM$ and $costSR$ using the cost model proposed in Section 4.2.2
 2: if $costDM < costSR$ then
 3:     $S_k \leftarrow DM(\mathcal{D},k)$
 4: else
 5:     $S_k \leftarrow SR(\mathcal{D},k,\alpha)$
 6: repeat
 7:     $S' \leftarrow Refine(\mathcal{D},S_k)$
 8:     $diff = score(Q,S') - score(Q,S_k)$
 9:     if $diff > 0$ then
10:         $S_k \leftarrow S'$
11: until $diff \leq 0$ or max time elapsed or user interrupts
12: return $S_k$
```

Next, we prove that $DivF$ halts after finite number of iterations, and when it halts (no refinement is possible in the mappers of the *refine* phase), it produces a 2-approximate solution compared to the optimal solution.

Lemma 5 *$DivF$ halts after finite number of iterations.*

Proof 6 *Since the while loop in Algorithm 14 is the only iterative component, it suffices to show that the while loop halts after finite number of iterations. Let S_{opt} denotes the optimal k-diverse element set of $\mathcal{D}$. Each iteration of the while loop changes the current k element set S_k to a new refined set and improves the current score by a positive value $(0,\mathcal{F}(S_{opt}) - \mathcal{F}(S_k)]$. Note that, the total number of k elements subsets of $\mathcal{D}$ is $\binom{|\mathcal{D}|}{k}$, and each subset has a fixed score $\mathcal{F}$. Therefore, $DivF$ halts after at most $\binom{|\mathcal{D}|}{k}$ iterations.*

Although the proof section of Lemma 5 considers the worst case scenario ($\binom{|\mathcal{D}|}{k}$ iterations), in Section 4.3, we show empirically (Figures 4.9(b)) that there is a high probability that $DivF$ halts after $2 \sim 3$ iterations.

To prove that our iterative algorithm $DivF$ produces 2-approximate solution we assume that the distance metric d follows the triangular inequality. We can rewrite the score $\mathcal{F}(S_k)$ as, $\mathcal{F}(S_k) = \frac{1}{2}\sum_{e_i \in S_k}\sum_{e_j \in S_k, e_j \neq e_i} d(e_i, e_j) = \frac{1}{2}\sum_{e_i \in S_k} C_{e_i}^{S_k}$. Here $C_{e_i}^{S_k}$ denotes the contribution of an element $e_i \in S_k$ to $\mathcal{F}(S_k)$ which is the sum of all $d(e_i, e_j)$ between e_i and the other elements e_j in S_k. Let S is the final k element set returned by $DivF$ when it halts.

Theorem 3 $\mathcal{F}(S_{opt}) \leq 2\mathcal{F}(\mathcal{S})$.

Proof 7 *Let us consider the worst case scenario where S_{opt} and S have no elements in common, that means $S_{opt} \cap S = \emptyset$. We will establish a one to one mapping between the elements in S_{opt} and S. Let e_i and e_i' are two arbitrary elements from S and S_{opt} respectively. Since, the total score of an element set is half to the sum of all individual element contributions to the score, to prove Theorem 3, it suffices to show that $C_{e_i'}^{S_{opt}} \leq 2C_{e_i}^{S}$.*

Let e_j' is an element in S_{opt} and $e_j' \neq e_i'$. Also let C_1 (or C_2) denotes the sum of all $d(.,.)$s between the elements in S/e_i and e_i' (or e_j'). Note that C_1 (or C_2) is less than $C_{e_i}^{S}$, otherwise e_i would be replaced by e_i' (or e_j') in the refine phase. By the triangular inequality, we can say that the sum of C_1 and C_2 is at least as $(k-1)d(e_i', e_j')$ (as seen in Figure 4.4). Thus, $(k-1)d(e_i', e_j') \leq C_1 + C_2 \leq 2C_{e_i}^{S}$. Therefore, we can write, $C_{e_i'}^{S_{opt}} = \sum_{e_j' \in S_{opt}/e_i'} d(e_i', e_j') \leq \sum_{e_j' \in S_{opt}/e_i'} \frac{2}{(k-1)} C_{e_i}^{S} = 2C_{e_i}^{S}$.

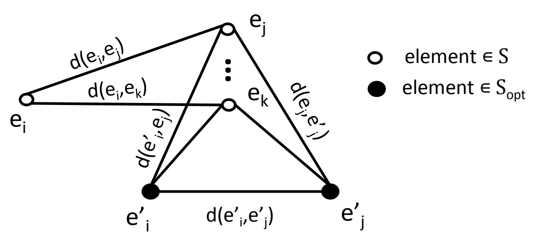


Figure 4.4: 2-approximation of $DivF$

4.3 Evaluation

We proceed with an experimental evaluation of the three main components (*diversification approaches*, *cost model*, *iterative refinement*) of our diversification framework. Section 4.3.1 describes the setup along with the datasets, methodology, cluster parameters used for the experiments. In Section 4.3.2, we evaluate the quality and performance of our two *diversification approaches*, *DM* and *SR*, by changing the environmental and algorithmic parameters, and the dataset size. In Section 4.3.3, we analyze the accuracy of our *cost model*. Finally, Section 4.3.4 provides an empirical evaluation of the *iterative refinement* component using real life datasets.

4.3.1 Setup

All of our experiments are performed on a five node cluster running Hadoop 1.0.4. Each node has 8 cores (3.30GHz Intel Xeon CPU) with 16GB RAM and 1TB of raw disk storage. We configure each node to run 8 tasks (map/reduce) at a time. Thus, at any point of time, we can run at most 40 tasks concurrently on our cluster.

Datasets: We use two datasets for the experiments in this section,

- **TwitterCrawl:** This dataset contains 82,774,328 tweets crawled from Twitter[10]. Each tweet has 8 terms on average. The total size of the dataset on disk is 7.55GB.
- **Image:** This dataset has 79,302,017 feature vectors extracted from collection of images[9]. Each vector has 16 features. The total size on disk is 5.12GB.

Methodology: We randomly select 100 elements from each dataset and use them as the queries. The results shown in this section are averaged over these 100 queries. We use the same distance metric for rel and dis calculation; *euclidean* distance for *Image* dataset, and *cosine* similarity for *TwitterCrawl* dataset. Note that, the relevance features are always taken from the user in the form of a query and the user always expects relevant answers. However, since the user has given a subset of features, diversification is necessary before presenting the potential large set of relevant results. Therefore, in our experiments, a subset of the feature set is used for rel calculation and the whole feature set is used for dis calculation [83]. For the *Image* dataset, first half of the features are used for rel calculation (i.e. $rel = (1 - L_2(1..8))$), and the whole feature set is used for dis calculation (i.e. $dis = L_2(1..16)$). For the *TwitterCrawl* dataset, three random terms are used for rel calculation (i.e. $rel = cosine(terms)$), and the whole tweet is used for dis calculation (i.e. $dis = 1 - cosine(tweets)$).

Uniprocessor Diversification: Several algorithms have been proposed in literature for result diversification in a uniprocessor system [81][83][36][84][26]. For our experiments, we use the MMR [26] diversification because of its efficiency within a single node [83]. MMR picks diverse results using the greedy strategy. The first element is always picked as the most relevant one. Successive diverse results are picked (one at a time) that maximize the $\mathcal{F}$ score with respect to the current selected diverse results. This process continues until k diverse results are picked. Note that, recently two other algorithms (GMC and GNE) have been proposed for diversification[83]. These algorithms may provide better diversification quality but require quadratic running time on the size of the data set $\mathcal{D}$. Instead MMR's running time is linear to the size of $\mathcal{D}$.

Cluster Parameters: We perform several experiments on our cluster to estimate the values of the parameters in Table 4.1. Based on our experiments, the values are estimated as, disk rate $D_r = 20MB/sec$, network rate $N_r = 10MB/sec$, $delay(1) = 0.1sec$, dispersion ratio $\beta = 0.8$. The uniprocessor MMR cost is estimated as $costD(bk) = 3.37E^{-6}bk\ sec$, which is the cost to generate k diverse results from b bytes. The mapper cost during the *refine* phase of *SR* approach is estimated as, $costP(bk) = 2.9E^{-7}bk\ sec$. Note that, the mapper processing cost in the first phase of the diversification approaches (*DM* and *SR*) are ignored from our calculation since the values are estimated as close to zero.

In all of our experiments, the number of mappers m is set by the Hadoop framework and we vary the number of reducers r from 1 to 40 (default value 40). We set the sampling ratio such that the number of elements processed in the single reducer during the *sample* phase of *SR* algorithm is ~ 1 million, i.e. $\alpha = \frac{1million}{|\mathcal{D}|}$. The results shown in this section are averaged over 10 distinct runs. Note that, unless specified explicitly, the default value used for k is 10 and for λ is 0.5

4.3.2 Evaluation of Diversification Approaches

We proceed with an evaluation of our two diversification approaches, *DM* and *SR*, in terms of quality and performance.

4.3.2.1 Qualitative Analysis

The motivation of the qualitative analysis is to answer the question, *Does the distributed implementation of the diversification algorithm degrades the quality when compared to the uniprocessor algorithm?* Therefore, we compare the quality achieved by the top-k result set generated by our two distributed approaches, *DM* and *SR*, with the quality of the uniprocessor *MMR* algorithm (i.e., a naive implementation where all elements are forwarded to a single reducer which computes the top-k diverse results using *MMR*). The comparison with a uniprocessor optimal algorithm has been skipped due to the optimal algorithm's time complexity (given that the dataset we consider is in the millions of records). However, [24] contains a comparison between the uniprocessor MMR and the optimal algorithm for a small dataset ($|\mathcal{D}| = 200$), which shows that MMR produced good quality results with respect to the optimal algorithm. We use score ($\mathcal{F}$) as the measurement of quality[83] for our experiments (i.e. higher $\mathcal{F}$ denotes better quality results).

Due to memory constraints and problem complexity, we could run the uniprocessor *MMR* only on a small set of elements. Figure 4.5 shows the quality comparisons of three algorithms, *DM*, *SR* and uniprocessor *MMR*, using a dataset with 10 million elements. We vary the number of reducers r from 1 to 40 (Figures 4.5(a),4.5(d)), the λ values from 0.1 to 0.9 (Figures 4.5(b),4.5(e)), the k values from 5 to 25 (Figures 4.5(c),4.5(f)) for two datasets *TwitterCrawl*, *Image* (respectively). Note that, since both the *SR* and uniprocessor *MMR* algorithms use only one reducer, the $(\mathcal{F})$ scores are fixed, independent from the changing r values (Figures 4.5(a),4.5(d)).

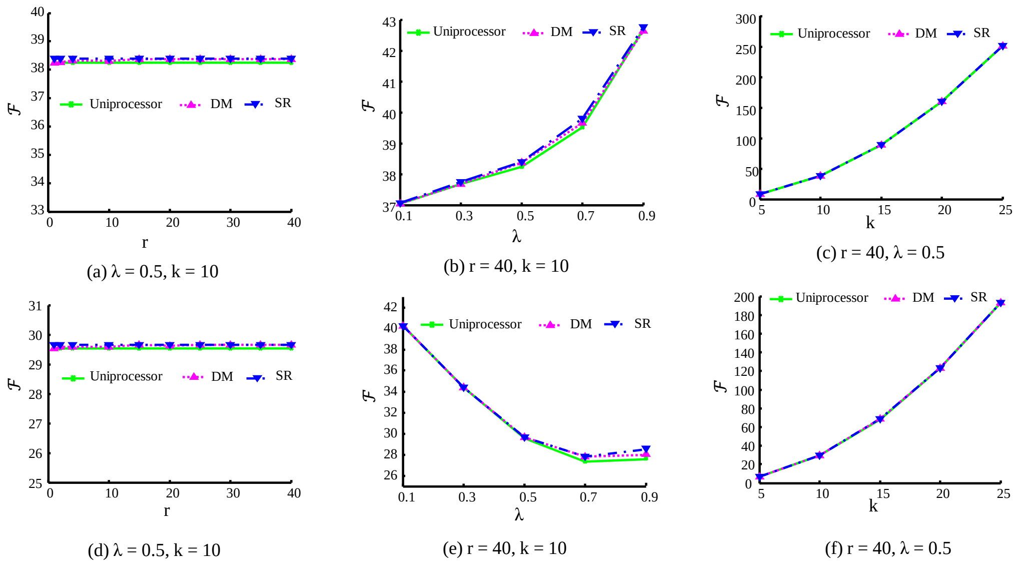


Figure 4.5: For *TwitterCrawl* dataset, (a) Avg. $\mathcal{F}$ vs. r, (b) Avg. $\mathcal{F}$ vs. λ, (c) Avg. $\mathcal{F}$ vs. k. (d),(e),(f) show the same figures for *Image* dataset

In all the experiments of Figure 4.5, the DM and SR approaches produce equal or better quality results when compared to the uniprocessor MMR algorithm. This is due to the nature of the DM and SR approaches. Note that, the quality of the final k diverse results produced by the uniprocessor MMR algorithm depends heavily on the first chosen element. If the first element is not chosen properly then the final k diverse results may be of low quality. In comparison, during the *divide* phase of our DM approach, each reducer uses a separate initial element to compute the k-diverse results from the assigned sample. Thus r different initial elements (one in each reducer) are used to compute a total of rk diverse elements. These rk elements form a better candidate set to be considered for diversification since these elements are selected by a uniprocessor MMR (run on different reducers). Therefore, the final k diverse results computed from these rk elements (during the *merge* phase) would be of better quality when compared to the results produced by the uniprocessor MMR on the whole dataset (as seen in most of the experiments of Figure 4.5).

In the case of the SR approach, a k diverse set is computed using the uniprocessor MMR from a sample of the dataset (in the *sample* phase) which is further refined by an additional scan of the whole dataset (in the *refine* phase). Therefore, the second scan of the dataset improves the quality of the diverse results produced by the SR approach when compared to the results produced by the uniprocessor MMR on the whole dataset.

Figure 4.6 shows the wall clock time needed by the three algorithms (uniprocessor *MMR*, *DM* and *SR*) for the quality experiments. As seen from the figure, our algorithms, *DM* and *SR*, compute diverse results much faster compare to the uniprocessor *MMR*. In fact, *DM* (*SR*) decreases the running time of uniprocessor *MMR* to a factor of 0.037 (0.119) and a factor of 0.039 (0.107) for the *TwitterCrawl* and *Image* datasets respectively.

Dataset	Wall Clock Time (sec)		
	Uniprocessor	DM (r = 40)	SR
TwitterCrawl	3053.94	113.19	363.49
Image	2193.72	85.1	235.63

Figure 4.6: Avg. wall clock time needed by Uniprocessor *MMR*, *DM* and *SR* algorithms for $k = 10$ and $|\mathcal{D}| = 10$ millions

4.3.2.2 Performance Analysis

Figures 4.7(a),4.7(c) show the wall clock time needed to compute top-10 diverse results by varying the number of reducers r, 1 to 40, using 10 million elements from the *TwitterCrawl* and *image* datasets respectively. As described earlier, the *SR* approach uses only one reducer, thus the total time taken by this approach is independent from the increasing number of reducers. However, the total time for the *DM* algorithm is decreasing nonlinearly with the increasing r. Note that, the *SR* algorithm performs better for smaller number of reducers $1 \sim 10$. If we increase the number of reducers, then *DM* algorithm starts to dominate.

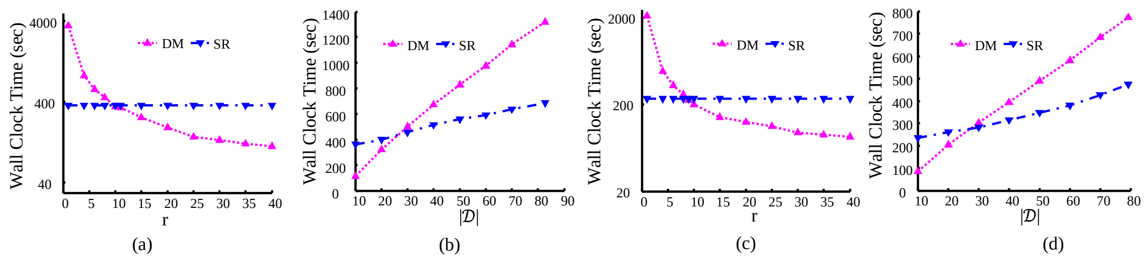


Figure 4.7: For *TwitterCrawl* dataset, (a) Avg. wall clock time vs. r, (b) Avg. wall clock time vs. $|\mathcal{D}|$. (c),(d) show the same figures for *Image* dataset

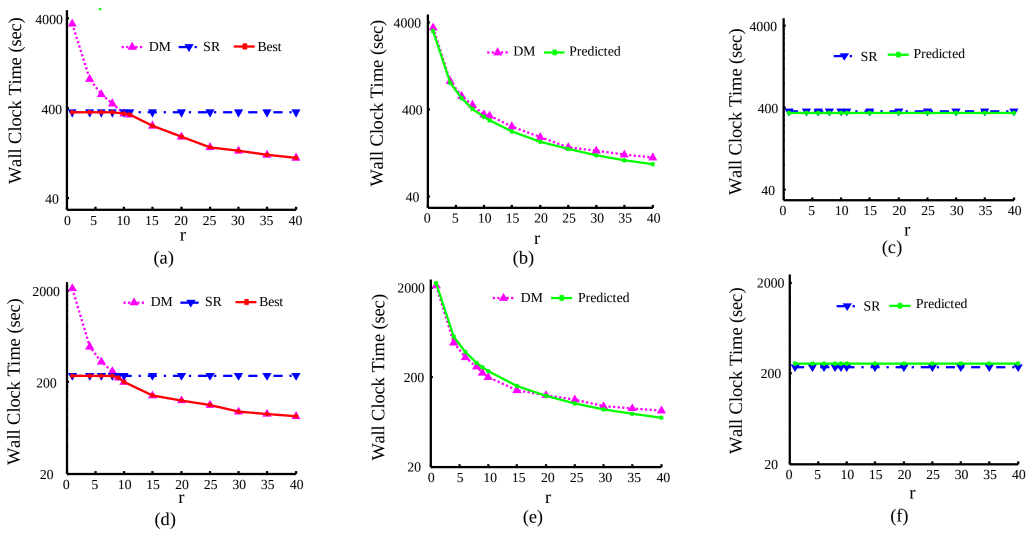


Figure 4.8: For *TwitterCrawl* dataset, (a) the ability of the cost model to select the best strategy (red curve) from *DM* and *SR* (b) actual and predicted times for *DM*, (c) actual and predicted times for *SR*. (d),(e),(f) show the same figures for *Image* dataset

Figures 4.7(b),4.7(d) show the wall clock time needed to compute top-10 diverse results by varying the number of records $|\mathcal{D}|$, 10 to ~80 millions, from the *TwitterCrawl* and *Image* datasets respectively. The number of reducers is fixed to 40. As seen from the figures, both of the approaches, *DM* and *SR*, show linear scale-up with the increasing $|\mathcal{D}|$ value. Unlike to the Figures 4.7(a) and 4.7(c), *DM* algorithm performs better for small number of elements (up to $\sim$ 25 millions), while *SR* works better for larger dataset.

Therefore, Figures 4.7(a)-(d) conclude that none of two diversification approaches, *DM* and *SR*, is a universal winner in all scenarios. *DM* performs better for small dataset and large number of reducers, while *SR* works better for larger dataset and small number of reducers. This provides a strong motivation of our *cost model* which has been analyzed in the next subsection.

4.3.3 Evaluation of Cost Model

In this subsection, we analyze the accuracy of the cost model. Figures 4.8(a), 4.8(d) show the same figures as in Figures 4.7(a), 4.7(c) with one more red curve (denoted as *best*) that, using cost model in Section 4.2.2, can consistently pick the best approach at runtime (as shown from Figures 4.8(a), 4.8(d)). Figures 4.8(b), 4.8(e) (Figures 4.8(c), 4.8(f)) show the actual and predicted running times for *DM* (*SR*) algorithm using two datasets *TwitterCrawl* and *Image* respectively. As seen from the figures, the predicted and actual times are close to each other. This confirms the fact that the cluster parameters estimated in Section 4.3.1 are realistic.

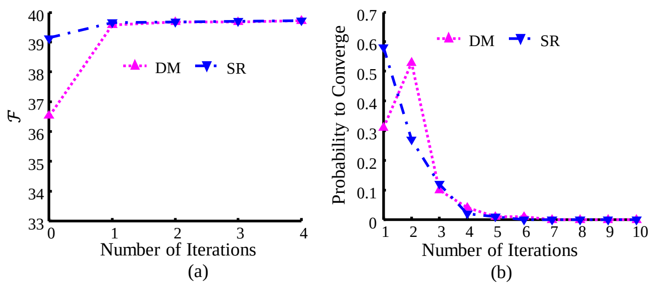


Figure 4.9: For *TwitterCrawl* dataset, (a) Avg. $\mathcal{F}$ vs. Number of Iterations, (b) Probability to Converge vs. Number of Iterations

4.3.4 Evaluation of Iterative Refinement

Our last experiment is to evaluate our iterative refinement component. As discussed in Section 4.2.3, our iterative component is useful when we have limited amount of resources (e.g. number of reducers, memory). Therefore, in the *divide* phase of DM approach, we set r to 8 and assign a random sample of 0.5 millions to each reducer. Similarly, the single reducer in the *sample* phase of SR approach gets a sample of size 0.5 millions.

Figure 4.9(a) shows the average values of $\mathcal{F}$ over iterations (0 to 4) for the top-10 diverse results calculated from the whole *TwitterCrawl* dataset. Note that, the scores at iteration 0 denote the $\mathcal{F}(S_{10})$ scores of the 10 diverse results returned from the diversification stage by the two diversification approaches DM and SR. As seen in Figure 4.9(a), the SR approach produces better quality result compare to the DM approach for limited resources. This is due to the *refine* phase in SR approach which makes a whole pass on $\mathcal{D}$. The DM approach reaches the quality of SR approach at iteration 1. After that the quality remains similar for both of the approaches.

We then estimate the number of iterations needed to converge the *while* loop in Algorithm 14. Figure 4.9(b) shows our estimated probability to converge vs. number of iterations using 100 random queries selected from the *TwitterCrawl* dataset. As seen from the figure, both of the diversification approaches, *DM* and *SR*, have high probability of convergence within $1 \sim 3$ iterations. The maximum number of iterations needed in our experiment was 7. Therefore, empirically we conclude that, with high probability, our iterative refinement strategy guarantees 2-approximate solution after 3 iterations.

4.4 Related Work

Diversification has recently been researched heavily for a uniprocessor environment. Several approaches have been proposed in literature for result diversification in many domains. [81][39][65][43] propose diversification in structured databases. [81] introduces diversity ordering between the result attributes of structured data and proposes diversification based on this ordering. [39] proposes diversification based on the possible interpretation of keyword queries and novelty of the search results. [65] tries to identify a set of differentiating features that help the users to analyze and compare the search results. [43] computes a diverse result set where the results are dissimilar to each other as well as cover the whole dataspace. Diversification has also been applied in recommendation systems [89][84]. [89] proposes topic diversification in the recommendation lists to cover the user's different range of interests. [84] introduces explanation based diversification where the explanation of an item is defined by a set of similar items rated by the user in the past. However, all these works consider online applications of diversification and propose algorithms assuming the size of the dataset is small enough (e.g. thousands of elements) to be stored in a single node's memory, thus fail to diversify large datasets (e.g. millions of elements). In this chapter, we consider diversification on massive datasets.

There are also diversification frameworks [25][37] proposed in literature that generate the diverse result set in polynomial time. [25] uses the query log information to measure the ambiguity of query terms (e.g. java, apple, jaguar) and proposes diversification which retrieves k documents that cover different specializations. [37] proposes diversification by proportionality; the number of documents in the final top-k result set covering a particular query aspect is proportional to the popularity of the aspect. However, all these approaches require some prior knowledge (e.g. query log, specializations, query aspects) which may not be available in all applications (when workloads and query information are not known in advance [83]). In this chapter, we adapt the general diversification framework described in [83] and propose distributed solution to the problem.

One may argue using an existing clustering approach [35][45], that clusters large datasets using the MapReduce framework, to generate k clusters and then pick one representative from each of the clusters, to provide a diversified result. In fact [80] uses clustering technique to generate diverse result set. However, as pointed out in [83] this approach does not guarantee good quality of diversification. This is because it is not trivial to identify which representative point to select from each cluster for the most diversified result. [83] showed experimentally that obvious choices of picking cluster representatives for the diversified result (k-medoids, etc.) provides low diversification score.

Chapter 5

Spatial Top-k Queries

In this chapter, we investigate the problem of answering top-k queries satisfying spatial constraints. Section 5.1 formulates the problem. Section 5.2 describes our index structure and Section 5.3 presents our model. Finally, we evaluate our indexing approach in Section 5.4.

5.1 Problem Definition

Let $\mathcal{D}$ =$\{o_1, o_2, ..., o_N\}$ be a dataset with N objects where each object $o \in \mathcal{D}$ has a pair of attributes $< Loc, Terms >$; $o.Loc$ is a point in a d-dimentional space $\mathcal{R}^d$ and $o.Terms = \{t_1, t_2, ...\}$ is a set of terms including duplicates. Let $V = \{\cup_{o \in \mathcal{D}} o.Terms\}$ is the vocabulary with all terms.

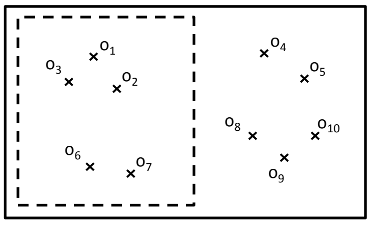
As an example, consider a dataset with 10 objects (Figure 5.1). Figure 5.1(a) shows the locations of the objects in a 2D space and Figure 5.1(b) shows the terms of the objects. $\{\cup_{i=1}^{10} t_i\}$ is the vocabulary with 10 terms.

The frequency of a term $t \in V$ is denoted as $f(t) = \{f_{o_1}(t) + f_{o_2}(t) + ... + f_{o_N}(t)\}$, where $f_o(t)$ denotes the number of times t appears in $o.Terms$. Given a spatial region $R \subseteq \mathcal{R}^d$, the frequency of term t in R is denoted as $f_R(t) = \{\sum f_{o_i}(t) | o_i.Loc \in R\}$.

A top-k spatial query Q is a pair of attributes $< R_Q, k >$, where R_Q denotes the region the user is interested in and k denotes the number of output terms. The goal is to find the k terms whose frequencies are the maximum in R_Q.

Consider the example in Figure 5.1. The dotted region in Figure 5.1(a) denotes the query region R_Q. Let the user is interested in two terms ($k = 2$). Therefore, the goal is to compute the two terms from $\{\cup_{i=1}^{10} t_i\}$ whose frequencies are the maximum in the dotted region (i.e. in the $Terms$ of five objects $\{o_1, o_2, o_3, o_6, o_7\}$).

[TOP-k SPATIAL TERMS]. *Given Q and $\mathcal{D}$, find the k terms: $t_1, t_2, ..., t_k$, whose frequencies $f_{R_Q}(t_1), f_{R_Q}(t_2), ..., f_{R_Q}(t_k)$ are the maximum among all terms in V.*



(a) Locations

Object	Terms	Object	Terms
o_1	$\{t_1, t_2, t_4, t_6\}$	o_6	$\{t_1, t_2, t_5, t_9\}$
o_2	$\{t_2, t_2, t_4\}$	o_7	$\{t_1, t_1, t_4\}$
o_3	$\{t_1, t_3, t_4\}$	o_8	$\{t_4, t_6, t_9, t_{10}\}$
o_4	$\{t_6, t_7, t_8, t_8\}$	o_9	$\{t_2, t_4, t_9\}$
o_5	$\{t_4, t_9, t_{10}\}$	o_{10}	$\{t_2, t_6, t_7\}$

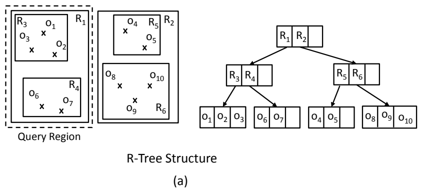
(b) Terms

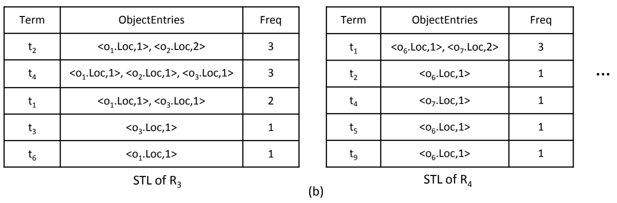
Figure 5.1: Sample dataset containing 10 objects, (a) shows the locations and (b) shows the terms of the objects.

5.2 Index Structure

To solve the top-k spatial terms (k-ST) problem, we consider two main challenges. First, given the dataset $\mathcal{D}$ and the query region R_Q, how can we efficiently identify the objects that are contained in R_Q? Second, given these contained objects, how can we compute the top-k terms efficiently from the $Terms$ of the contained objects? The first challenge has been addressed using a spatial index R-tree[50]. To address the second challenge, we pre-compute some sorted lists of terms for the nodes of the R-tree. We then execute the top-k algorithm on top of the sorted term lists. The details are explained next.

R-Tree Index: We stored the objects in $\mathcal{D}$ using an R-tree index. Each node n of the R-tree corresponds to a region $R_n \subseteq R^d$ and indexes the set objects contained in R_n. Each non-leaf node contains an entry $< ChildPtr, Region >$ for each child node of that node: $ChildPtr$ is the pointer to the child node and $Region \subseteq R^d$ is the space covered by the child node. The leaf nodes contain entries of the form $< ObjPtr, Loc >$: $ObjPtr$ is the pointer to an object and $Loc \in R^d$ is the location of the object in the d-dimentional space. The number of childs of an R-tree node is determined by the page size. Figure 5.2(a) shows the R-tree structure for the sample dataset in Figure 5.1.





Term	ObjectEntries	Freq
t_2	$<o_1.Loc,1>, <o_2.Loc,2>$	3
t_4	$<o_1.Loc,1>, <o_2.Loc,1>, <o_3.Loc,1>$	3
t_1	$<o_1.Loc,1>, <o_3.Loc,1>$	2
t_3	$<o_3.Loc,1>$	1
t_6	$<o_1.Loc,1>$	1

STL of R_3

Term	ObjectEntries	Freq
t_1	$<o_6.Loc,1>, <o_7.Loc,2>$	3
t_2	$<o_6.Loc,1>$	1
t_4	$<o_7.Loc,1>$	1
t_5	$<o_6.Loc,1>$	1
t_9	$<o_6.Loc,1>$	1

STL of R_4

...

(b)

Figure 5.2: Spatial R-Tree for the sample dataset in Figure 5.1 and leaf level STLs

Given the R-tree and the query region R_Q, we first traverse the R-tree to find the leaf nodes whose regions intersect with R_Q. We start from the root node of the R-tree. For a non-leaf node, we check the child entries to determine whether the entries overlap with the query region. The overlapping child entries are then checked further using the same strategy. This process continues until we reach the leaf level where the leaf nodes that intersect with R_Q are identified. The objects contained in R_Q are indexed by these leaf nodes. However, to compute the top-k terms efficiently, we pre-compute some additional sorted term lists for the leaf nodes of the R-tree.

Leaf Level Sorted Term List: For each leaf node n_l of the R-tree, we pre-compute a Sorted Term List (STL) which is a list of term entries sorted based on the frequencies of the terms in R_{n_l}. The total number of entries in STL is equal to $|V_{n_l}|$ where $V_{n_l} = \{\cup_{o \in \mathcal{D}} o.Terms | o.Loc \in R_{n_l}\}$. For each term $t \in V_{n_l}$, we have a term entry t_e of the form $< Term, ObjectEntries, Freq >$.

The first field $t_e.Term$ denotes the term t. The second field $t_e.ObjectEntries$ is a list of object entries of the form $< Loc, Freq > \equiv [\exists o \in \mathcal{D}$: $Loc = o.Loc \in R_{n_l}$ and $Freq = f_o(t) > 0]$. Finally, the third field $t_e.Freq$ is the sum of all $Freq$ values of the object entries in $t_e.ObjectEntries$. The term entries in STL are sorted by their $Freq$ values in descending order. For example, Figure 5.2(b) shows the STLs for the two leaf nodes R_3 and R_4 of the R-tree in Figure 5.2(a).

Once the leaf nodes that intersect with the query region R_Q are identified, we then execute the top-k algorithm on top of the STLs of the intersected leafs nodes. There are several existing top-k algorithms [46][68] in the literature. In this chapter, we use the most popular top-k threshold algorithm [46].

Threshold Algorithm: The threshold algorithm (TA) scans in parallel all the STLs that are involved in top-k computation. At each iteration i, it extracts the i^{th} term entry t_e from each of the involved STLs. The sum of all $t_e.Freq$ values is considered as the threshold θ at iteration i. Each time a new term t is seen, TA scans the other STLs to compute the aggregate $f_{R_Q}(t)$.

Note that in our case, for a given term entry t_e with the term t, if the leaf node n_l of the STL (that contains t_e) is fully contained in R_Q then $t_e.Freq$ is used in the $f_{R_Q}(t)$ computation. Otherwise, $t_e.ObjectEntries$ is scanned to compute $f_{R_Q \cap R_{n_l}}(t)$ which contributes to the $f_{R_Q}(t)$. TA stops when it finds k terms whose frequencies are higher or equal to the threshold θ value.

As an example, consider the dotted query region in Figure 5.1(a). Based on the R-tree in Figure 5.2(a), only the two leaf nodes R_3 and R_4 intersect with the query region. The STLs of the two leaf nodes are shown in Figure 5.2(b). Therefore TA executes on these two STLs. Assume the user is interested on two terms (i.e. $k = 2$). At iteration 1, TA scans the term entries at position 1. The two terms at position 1 are t_1 and t_2. TA scans these terms in all STLs and computes the frequency values: $f_{R_Q}(t_1) = 3+2 = 5$ and $f_{R_Q}(t_2) = 3+1 = 4$. The θ value at this point is $3+3 = 6$. Therefore the algorithm goes on and scans the term entries at position 2. The two terms at position 2 are t_4 and t_2. TA computes the frequency of the new term t_4 which is $3+1 = 4$. At this point, the top-2 terms are t_1 and t_2 (ties are broken arbitrarily). The θ value at position 2 is $3 + 1 = 4$ which is not higher than the frequencies of t_1 and t_2. Therefore TA stops the execution and declares t_1 and t_2 as the top-2 terms.

Solving k-ST with leaf level STLs shows good performance for small query regions. However, as the query region size increases, the number of intersected leaf nodes as well as the number of involved STLs in top-k computation increases. This slows down the performance of TA (Figure 5.10). Therefore, to improve the performance, we enhance our index structure with the STLs for the inner level nodes of the R-tree. Figure 5.3 shows the STLs for two inner level nodes R_1 and R_2. Note that the $ObjectEntries$ fields are removed from the term entries of the inner level STLs. This is to reduce the redundancy (as well as to improve the space efficiency) of the index structure.

Term	Freq
t_1	5
t_2	4
t_4	4
t_3	1
t_5	1
t_6	1
t_9	1

STL of R_1

Term	Freq
t_4	3
t_6	3
t_9	3
t_2	2
t_7	2
t_8	2
t_{10}	2

STL of R_2

...

Figure 5.3: Inner level STLs for the R-tree in Figure 5.2(a)

Using the additional inner level STLs, we consider a modified tree traversal algorithm (Algorithm 16). We start from the root node of the R-tree. If an inner level node is fully contained in the query region, then no further checking is required for the childs of that node. The STL of this fully contained node is used in top-k computation. However, if an inner level node overlaps with the query region then the child nodes are checked. This process continues until we reach the leaf level where the leaf nodes that intersect with the query region are identified. The STLs of these intersected leaf nodes are used top-k computation.

Using the modified tree traversal algorithm, consider the previous dotted query region in Figure 5.1(a). Based on the R-tree in Figure 5.2(a), only the inner level node R_1 is fully contained in the query region. Therefore, no further checking is done and the top-k terms are returned by TA from the STL of R_1. This reduces the number of involved STLs in TA from 2 to 1.

The problem with this modified approach is that it requires large amount of space especially for the STLs of the higher level nodes. Figure 5.4 shows the average number of term entries of the STLs at different levels. As seen from the figure, the number of term entries (as well as the size of STL) increases for the higher levels. Note that, TA scans only a small subset of the entries in general. In the example explained previously, TA stops after scanning two term entries from the STLs. Therefore, we exploit this early stopping property to compute the expected number of term entries (l) to be stored in a STL (discussed in section 5.3). We then use this l value to shrink the list size for the inner level STLs.

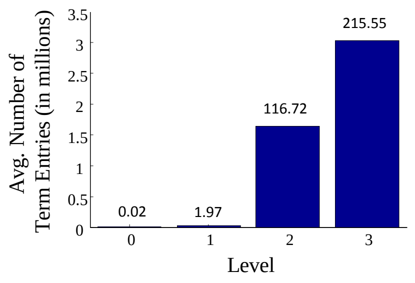


Figure 5.4: Level vs. avg. number of term entries per STL. The numbers on the top of the bars show the avg. sizes of STLs in MB

In the rest of this section, we describe the algorithms to solve the k-ST problem given that the inner level STLs contain l number of term entries while the leaf level STLs contain all term entries. We define this approach as **S**patial **T**erm **I**ndexing (STI) approach. At first, Algorithm 15 finds the R-tree nodes whose STLs are involved in top-k computation (line 1). After that, it executes the TA to compute the top-k term entries using algorithm 17 (line 2). The main component of algorithm 17 is the repeat-until loop (lines 2 - 24). At each iteration i, it scans the i^{th} term entry from each of the involved STLs (lines 5 - 8) and computes the θ value (line 9). Note that, when the index i exceeds l, algorithm 18 is called (line 8) for each involved inner level STL to compute the next term entry sorted order. Algorithm 18 invokes algorithm 17 recursively for the STLs of the child nodes. If a new term t is seen, algorithm 17 looks up the term t in all involved STLs and computes the aggregate frequency (lines 11 - 20). Algorithm 19 is a supporting method used to compute the value of a term entry t_e in R_Q. It scans all the object entries in $t_e.ObjectEntries$, finds the object entries that are contained in R_Q and aggregates their frequencies.

Algorithm 15 $STI(R_Q, k, root)$

```
Require: Query Region $R_Q$, number of output terms $k$ and the root node of R-tree $root$
Ensure: Return top-$k$ terms with highest frequencies in $R_Q$
1: $\mathcal{N} \leftarrow FindCandidateNodes(R_Q, root)$
2: $\mathcal{E} \leftarrow ExecuteTA(\mathcal{N}, R_Q, k)$
3: $\mathcal{T} \leftarrow \emptyset$
4: for each term entry $t_e$ in $\mathcal{E}$ do
5:     $\mathcal{T} \leftarrow \mathcal{T} \cup t_e.Term$
6: return $\mathcal{T}$
```

5.3 Estimation of Expected STL Size

We estimate the expected STL size l in two steps. In the first step, we estimate the expected numbers of STLs (from different levels of the R-tree) that are involved in top-k calculation. Let $M = (m_1, m_2, ..., m_h)$ be the vector; h is the height of the R-tree and m_i denotes the expected number of STLs involved from level i. Using M, we calculate l in the second step.

Algorithm 16 $FindCandidateNodes(R_Q, n)$

```
Require: Query Region $R_Q$, and the node $n$
Ensure: Return the set of candidate nodes from the subtree rooted at $n$ such that the STLs of
    the selected nodes are used for top-$k$ computation
 1: if $IsLeaf(n)$ then
 2:     if $R_Q \cap R_n \neq \emptyset$ then
 3:         return $\{n\}$
 4:     else
 5:         return $\emptyset$
 6: if $R_n \subseteq R_Q$ then
 7:     return $\{n\}$
 8: else
 9:     $\mathcal{N} \leftarrow \emptyset$
10:     for each child $c$ of $n$ do
11:         $\mathcal{N} \leftarrow \mathcal{N} \cup FindCandidateNodes(R_Q, c)$
12:     return $\mathcal{N}$
```

Step 1: Given the query region R_Q, we start from the root level (level h) of the R-tree. At each inner level i ($2 \leq i \leq h$), we estimate the expected number of nodes which are fully contained in query region and the region covered by the contained nodes. The STLs of the contained nodes are involved in the top-k calculation. Thus, m_i for inner levels is equal to the number of contained nodes. The remaining query region (i.e. the query region which is not covered by the contained nodes) is used as the new query region for the next level $i + 1$. This process continues until we reach the leaf level ($i = 1$) where the number of nodes that intersect with the new query region is estimated and considered as m_1.

Now we start our theoretical analysis on how to estimate the M. Table 5.1 describes the model parameters. We consider an d-dimensional unit dataspace ($[0, 1)^d$) which contains the N objects. An R-tree with height h and average node capacity (fanout) f stores these N objects. Let, N_i is the number of nodes at level i and $S_i = (s_{i,1}, s_{i,2}, ..., s_{i,d})$ is the average size of a level i node. First we estimate the R-tree properties (h,N_i,S_i) using the analysis described in [79], then we calculate the M.

Algorithm 17 $ExecuteTA(\mathcal{N}, R_Q, k)$

Require: Set of nodes $\mathcal{N}$, the query region R_Q and the number of output term entries k

Ensure: Execute TA on the STLs of the nodes in $\mathcal{N}$ and return the top-k term entries with highest frequencies

```
1: $\mathcal{E} \leftarrow \emptyset, i \leftarrow 0$
2: repeat
3:   $\theta \leftarrow 0,\ f \leftarrow 0$
4:   for each node $n$ in $\mathcal{N}$ do
5:     $L \leftarrow getSTL(n)$
6:     $t_e \leftarrow i^{th}$ term entry in $L$
7:     if $t_e$ is null and $NotIsLeaf(n)$ then
8:       $t_e \leftarrow GetTermEntry(n, R_Q, i)$
9:     $\theta \leftarrow \theta + t_e.Freq$
10:    $t \leftarrow t_e.Term$
11:    if $t$ has not been seen yet then
12:      $f' \leftarrow 0$
13:      if $isLeaf(n)$ then
14:        $f' \leftarrow ComputeTermFreq(t_e, R_Q)$
15:      else
16:        $f' \leftarrow t_e.Freq$
17:      for each node $n'$ in $\mathcal{N}$ do
18:        if $n' \neq n$ then
19:          do random access for term $t$ on STL of $n'$ and compute $f_{R_Q \cap R'_n}(t)$
20:          $f' \leftarrow f' + f_{R_Q \cap R'_n}(t)$
21:      $\mathcal{E} \leftarrow \mathcal{E} \cup \{< t, \emptyset, f' >\}$
22:  $f \leftarrow$frequency of the $k^{th}$ term entry in $\mathcal{E}$
23:  $i \leftarrow i + 1$
24: until $\theta > f$
25: return top-$k$ term entries in $\mathcal{E}$ with highest frequencies
```

Since N objects are contained in N_1 nodes at leaf level and the average fanout factor is f, the number of leaf level nodes is $N_1 = \frac{N}{f}$. Similarly N_1 nodes are contained in N_2 nodes at level 2, therefore $N_2 = \frac{N}{f^2}$. Thus the number of nodes at level i is,

$$N_i = \frac{N}{f^i} \tag{5.1}$$

The height h of the R-tree is calculated as [79],

$$h = 1 + \lceil \log_f \frac{N}{f} \rceil \tag{5.2}$$

To compute S_i, we assume that the node sides are equal in all dimensions (i.e. $s_{i,1} = s_{i,2} = ... = s_{i,d}$). Let s_i be the average size of a level i node in all dimensions. Since f number

Algorithm 18 $GetTermEntry(n, R_Q, i)$

```
Require: The node $n$, the query region $R_Q$ and the index $i$
Ensure: Return the $i^{th}$ term entry in the region $R_n$
1: $\mathcal{N} \leftarrow \emptyset$
2: for each child $c$ of $n$ do
3:     $\mathcal{N} \leftarrow \mathcal{N} \cup c$
4: $\mathcal{E} \leftarrow ExecuteTA(\mathcal{N}, R_Q, i)$
5: return the $i^{th}$ term entry in $\mathcal{E}$
```

Algorithm 19 $ComputeTermFreq(t_e, R_Q)$

```
Require: The term entry $t_e$, and the query region $R_Q$
Ensure: Return the frequency of $t_e$ in the region $R_Q$
1: $f \leftarrow 0$
2: for each object entry $o_e$ in $t_e.ObjectEntries$ do
3:     if $o_e.loc \in R_Q$ then
4:         $f \leftarrow f + o_e.freq$
5: return $f$
```

of level $(i-1)$ nodes are contained in a single node at level i, the number of level $(i-1)$ nodes that contribute to a single side of level i node is $\sqrt[d]{f}$. Therefore s_i can be computed as,

$$s_i = (f^{1/d} - 1).\frac{1}{(N_{i-1})^{1/d}} + s_{i-1} \tag{5.3}$$

Here $(N_{i-1})^{1/d}$ is the average distance between the centers of two consecutive level $(i-1)$ node projections in a single dimension. The detailed analysis is described in [79].

For simplicity, we assume that the query sides of R_Q are also equal in all dimensions (i.e. $q_1 = q_2 = ... = q_d = q$). Given N_i and s_i, the number of level i nodes that intersect with query region q^d is [79],

$$intersect(N_i, s_i, q) = N_i.(s_i + q)^d \tag{5.4}$$

As stated earlier, in this chapter we are interested in the estimation of the number of fully contained nodes for inner levels (which is a subset of intersected nodes computed in [79]). The next analysis describes how we can estimate the number of contained nodes given the values N_i, s_i and q.

Symbol	Description
h	height of R-tree
$S_i = (s_{i,1}, s_{i,2}, ..., s_{i,d})$	avg. size of an MBR at level i
$N_r = (N_1, N_2, ..., N_h)$	number of MBRs at different levels
$R_Q = (q_1, q_2, ..., q_d)$	size of query region
f	avg. node capacity (fanout)
N	number of objects

Table 5.1: Model Parameters

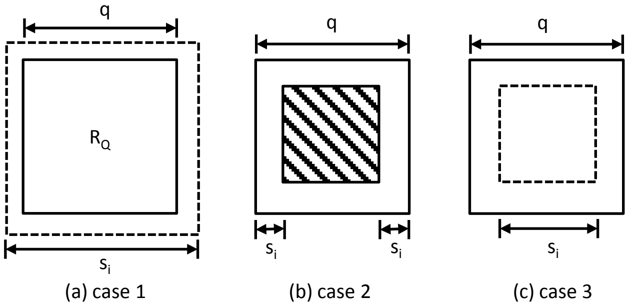


Figure 5.5: Case analysis of s_i and q

Depending on the values of s_i and q, there are three possible cases (Figure 5.5),

Case 1 ($s_i > q$)**:** The node size is greater than the query region. Therefore, no node is contained in the query.

Case 2 ($q \geq 2s_i$)**:** In this case, we consider a d-dimensional rectangle (the shaded region in Figure 5.5(b)) inside the query region where each side of the inner rectangle is s_i distance far away from the query rectangle. We argue that the nodes which intersect with the inner rectangle are the nodes that are contained in the query region. The number of nodes that intersect with the inner rectangle is $intersect(N_i, s_i, q - 2s_i)$. Let A_i is the region covered by the contained nodes. By the similar analysis performed during s_i calculation, we can estimate the average size (say a_i) of a single side of A_i, which is,

$$a_i = (intersect(N_i, s_i, q - 2s_i)^{1/d} - 1).\frac{1}{(N_i)^{1/d}} + s_i \qquad (5.5)$$

The remaining uncovered region (i.e. $q^d - (a_i)^d$) is considered as the new query region for the next level which can be divided into small rectangles of size $(\frac{q-a_i}{2})^d$. The number of small rectangles is estimated as $\lceil (q^d - (a_i)^d)/(\frac{q-a_i}{2})^d \rceil$.

Case 3 $(s_i \leq q < 2s_i)$**:** In this case, only one node can be contained in the query region assuming no overlapping between the nodes at a given level. This assumption is a reasonable property for a good R-tree [19]. Using the similar analysis explained in case 2, the remaining uncovered region (i.e. $q^d - (s_i)^d$) is divided into $\lceil (q^d - (s_i)^d)/(\frac{q-s_i}{2})^d \rceil$ small rectangles of size $(\frac{q-s_i}{2})^d$. These small rectangles are considered as the new query region for the next level.

To compute m_1, we first compute the total number of leaf nodes covered by the contained inner level nodes. We compute this value as $\sum_{i=2}^{h} m_i f^{i-1}$ since each contained node at level i ($2 \leq i \leq h$) covers total f^{i-1} number of leaf nodes. We then subtract this value from the total number of leaf nodes that intersect with the original query q^d. The exact formula used for m_1 computation is,

$$m_1 = intersect(N_1, s_1, q) - \sum_{i=2}^{h} m_i f^{i-1} \tag{5.6}$$

Algorithm 20 shows the pseudocode to compute M. At first, lines $1 - 4$ compute the R-tree properties h, N_i and s_i. Then using the R-tree properties, the second for loop (lines $7 - 21$) computes the M. Each iteration of the for loop corresponds to a level of R-tree. Lines $8-9$ compute m_1 for the leaf level and lines $10-21$ compute m_i for the inner levels ($2 \leq i \leq h$). The variable $factor$ stores the number of query rectangles at a level i.

Step 2: The analysis in this step is based on the assumption that the frequencies of terms in the whole corpus (i.e. the collection of all terms in the dataset) follow the Zipf distribution. This is true[57] for the collection of documents collected from several online sources: *Myspace*[7], *Twitter*[10], *Slashdot*[8].

Let each object has x number of terms on average. Therefore, the total number of terms (including duplicates) in the whole corpus is Nx. The Zipf law states that the frequency of a term is inversely proportional to its rank in the frequency list. That means,

Algorithm 20 $ComputeM(N, f, d, q)$

```
Require: The number of objects $N$, the fanout factor $f$, the dimensionality $d$ and the average size of query region side $q$
Ensure: Return the vector $M$
1: Calculate $h$ using Equation 5.2
2: $s_0 \leftarrow 0$
3: for $i \leftarrow 1$ to $h$ do
4:     Calculate $N_i$ and $s_i$ using Equations 5.1 and 5.3 respectively
5: $factor \leftarrow 1$
6: $q' \leftarrow q$
7: for $i \leftarrow h$ to 1 do
8:     if $i = 1$ then
9:         Calculate $m_1$ using Equation 5.6
10:    else
11:        if $q' < s_i$ then
12:            $m_i \leftarrow 0$
13:        else if $q' \geq 2s_i$ then
14:            $m_i = factor \times intersect(N_i, s_i, q' - 2s_i)$
15:            Calculate $a_i$ using Equation 5.5
16:            $factor \leftarrow factor \times \lceil((q')^d - (a_i)^d)/(\frac{q'-a_i}{2})^d\rceil$
17:            $q' \leftarrow \frac{q'-a_i}{2}$
18:        else
19:            $m_i \leftarrow factor \times 1$
20:            $factor \leftarrow factor \times \lceil((q')^d - (s_i)^d)/(\frac{q'-s_i}{2})^d\rceil$
21:            $q' \leftarrow \frac{q'-s_i}{2}$
22: return $M = \{m_1, m_2, ..., m_h\}$
```

$$p\frac{freq(p, Nx)}{Nx} = c \tag{5.7}$$

Here, p is the rank of the term, $freq(p, Nx)$ is the frequency of the p^{th} term in the frequency list of a dataset containing Nx terms and c is the collection specific parameter. Using Equation 5.7, the frequency $freq(p, Nx)$ of a term at any arbitrary rank p can be computed which is $\frac{cNx}{p}$.

In our case, each level i node of the R-tree contains on average Nx/N_i terms. Therefore, the frequency of the p^{th} term in the STL of a level i node is $\frac{c_i Nx}{N_i p}$. Similarly the frequency of the p^{th} term in the STL of the query region q^d is $\frac{c_q q^d Nx}{p}$.

Note that the top-k threshold algorithm works by computing the threshold value at successive index p which is the sum of all p^{th} frequency values in the STLs that involved in top-k calculation. Given Nx, q and M, the threshold value at an index p is computed as,

$$\theta(p, q, Nx, M) = \sum_{i=1}^{h} m_i \frac{c_i Nx}{N_i p} \tag{5.8}$$

The top-k threshold algorithm stops at an index p when the threshold value equals or drops below the k^{th} frequency value in the query region q^d. Therefore the expected list size is computed as,

$$l(k, q, Nx, M) = \underset{p}{\operatorname{argmin}}\{\theta(p, q, Nx, M) \le \frac{c_q q^d Nx}{k}\} \tag{5.9}$$

5.4 Experimental Evaluation

In this section we present the results of an experimental evaluation of our STI approach. Section 5.4.1 describes the setup including the dataset, baseline approaches and index structures used for the experiments. Section 5.4.2 evaluates our estimated M and l with respect to the actual values. Section 5.4.3 compares the index sizes required by the baseline approaches and our STI approach. Finally, Section 5.4.4 analyzes the scalability of the approaches by varying the query selectivity size. All experiments are performed on a 2.5GHz Intel Core i5 CPU, 8GB RAM machine running Windows 7 OS.

5.4.1 Setup

Dataset: For our experiments, we crawled total 5,362,676 tweets [10] within the continental United States. We remove the stop keywords from the tweet text to get meaningful top-k terms. After removing the stop keywords, each tweet has 6 terms on average.

Baseline Methods: We compare our STI approach with the following two baseline approaches,

Baseline-LL: In this approach, we pre-compute STLs only for the leaf level nodes.

Baseline-FL: As a second baseline, we pre-compute *full* (i.e. the STL of a node n contains term entries for all terms in R_n) STLs for all nodes in the R-tree.

Index Structure: We consider three level indexing scheme for all the methods. First, We compute an R-tree on top of the tweets in the dataset. The page size of the R-tree node is set to $16KB$ (maximum 200 entries). The average fanout factor f is 136 (typical 68% average node capacity [79]). Second, we store the STLs of R-tree nodes in a column family using *Cassandra* $2.0.5$ database management system. To improve the efficiency of TA, we divide each STL into collection of pages of size $16KB$ and load one page at a time. We store each STL page as a separate row in the column family. The row key is the concatenate form of the STL identifier and the page index. Third, to allow random access on the STLs, we store the terms in the STLs in a separate column family. Here, each term is stored as a separate row in the column family. The row key is the concatenate form of STL identifier and the term. The value is the frequency of the term in the STL.

Note that the top-i term entries is a subset of the top-$(i + 1)$. Thus each successive execution of Algorithm 18 performs some repetitive computation which has been done in the previous execution. Therefore, to improve the efficiency, we store the final state of Algorithm 18 (as well as Algorithm 17) after each execution which has been used in the next execution.

The location of the query region is selected based on the number of tweets contained in that region. The dense areas have higher probabilities to be selected as query regions. We varied the query region size as a fraction (selectivity) of the whole space and the results showed in this section are averaged over 100 runs. The default value of k is set to 10 in all experiments.

5.4.2 Evaluation of M and l

Our first experiment is to evaluate the vector M computed by Algorithm 20. Figure 5.6 shows the expected ($\sum_{i=1}^{h} m_i$) and the actual numbers of STLs involved in TA for different selectivity of the query regions. As expected, the number of STLs increases with the query selectivity size. The gaps between the actual and expected numbers are because of our assumption made in Section 5.3. We assume that the objects' locations follow uniform distribution, while in practice, the tweets' locations follow skewed distribution.

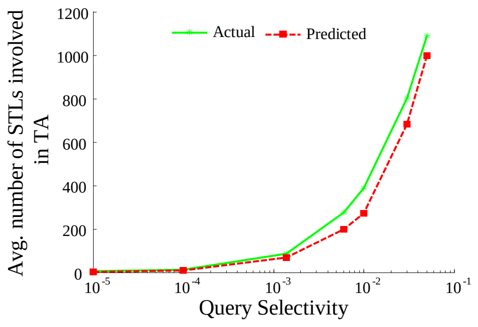


Figure 5.6: Avg. number of STLs involved in TA for different selectivity of query regions

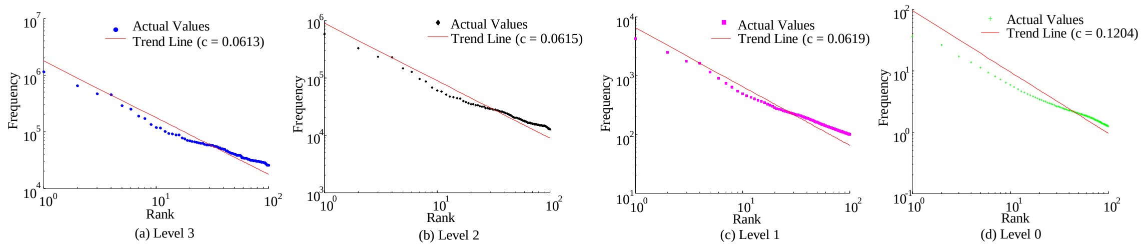


Figure 5.7: *zipf* parameters for different levels of R-tree

We then compute the *zipf* parameters for different levels of the R-tree constructed on top of the tweets. Based on setup described in Section 5.4.1, the R-tree contains four levels. Figure 5.7 shows the average frequency vs. rank values for the STLs at different levels of the R-tree. As seen from the figure, the *zipf* parameters are almost equal (~ 0.061) for the top three levels of the R-tree. The leaf level STLs have different zipf parameter (0.1204). This is because for the less number of tweets contained by the leaf level nodes compared to the top level nodes. For the experiments in this section, we set the query region *zipf* parameter equal to the closest sized R-tree level's parameter.

Our next experiment is to evaluate the expected stopping point (l) of TA computed using Equation 5.8. Figure 5.8 shows the expected and the actual stopping points for different selectivity of query regions. The black line shows the average selectivity of level 2 nodes. Starting from left, at selectivity 0.00001, the query region is less than the average size of leaf nodes. Therefore, TA invokes Algorithm 19 to compute the actual frequencies of the term entries in the query region. Note that the threshold value is computed using the term entry frequency values (line 9 in Algorithm 17). Thus the threshold value has been overestimated compared to the actual values and hence, the stopping point is high. As the query region size increases, the problem caused by the overestimation decreases, so the stopping point also decreases as well (as seen from Figure 5.8). When the query selectivity exceeds the average selectivity of level 2 nodes, some of the leaf level STLs are replaced by inner level STL(s) which start(s) to dominate in top-k computation. Therefore, the threshold value continues to decrease.

During the index build up phase for the STI approach, we first select a query range to be answered. We compute the expected l values for different query regions within the range and find the maximum (l_{max}) of the computed l values. We then store l_{max} number of term entries in the inner level STLs. For the rest of the experiments in this section, we set the query range to be answered from selectivity 0.00001 to 0.05. The l_{max} value is computed as 62 (for selectivity 0.00001 in Figure 5.8).

5.4.3 Index Size

We also compare the three approaches (LL, STI and FL) in terms of the sizes required to store the three different indexes: R-tree index, STL index and term index (Figure 5.9). Since the

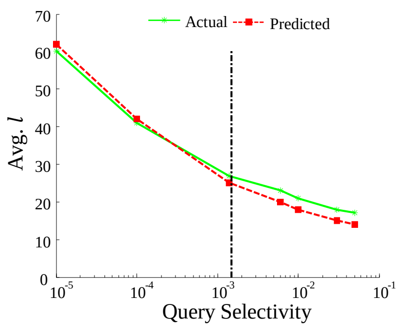


Figure 5.8: Avg. stopping index of TA for different selectivity of query regions

R-trees are identical in all three approaches, the R-tree index sizes are same (as seen in Figure 5.9). The LL approach stores STLs only for the leaf level nodes, thus it requires the least storage (both for the STL and term indexes) among all three approaches. The FL approach stores full STLs for all nodes, therefore the space requirement is the maximum among all three approaches. STI approach stores full STLs for the leaf level nodes and partial STLs (only 62 entries) for the inner level nodes. In comparison with the other approaches, the space requirement of STI is slightly greater than LL (factor of 0.0014) while lot less than FL (factor of 0.2817).

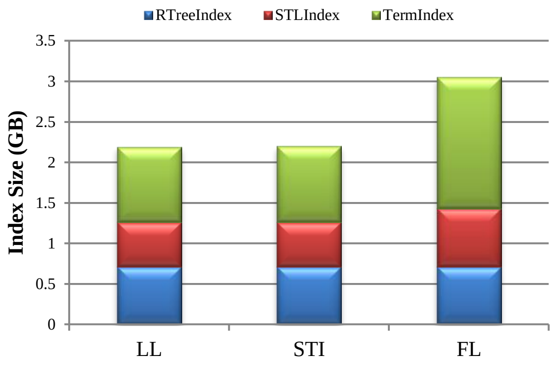


Figure 5.9: Disk sizes required for different index structures

5.4.4 Scalability Analysis

Finally, we perform the scalability analysis for the three approaches, LL, STI and FL (Figure 5.10). As seen from the figure, the processing time increases with the query selectivity size. Note that, when the query region size is less than the size of level 2 nodes, only leaf level STLs are involved in TA. Thus all three approaches perform same. As the query size increases, some leaf level STLs are replaced by the inner level STL(s) both for the STI and FL approaches. Therefore, STI and FL approaches start to perform better compare to the LL approach (Figure 5.10). Our approach STI shows same performance like FL. This is because we have stored 62 entries for the inner level STLs which are sufficient to answer the query regions greater than the size of the level 2 nodes (Figure 5.8).

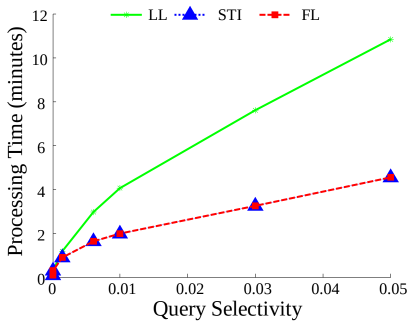


Figure 5.10: Avg. processing time of TA for different selectivity of query regions

Chapter 6

Conclusions and Future Work

6.1 Diversification of Query Results

In this thesis, we have considered *result diversification* on semi-structured data (Chapter 2). For this, we have modified the standard tree edit distance to consider the query-context and the contents for semi-structured result diversification. We have also given a novel heuristic and pruning based technique for speeding up the existing algorithms by factor of two.

Our focus has been to develop methods to diversify *exact* matches to the query. As for future research, we note that our algorithm can be extended to work on approximate matches. In a typical approximate query matching system [15][16], the query is perturbed to generate few approximate queries. Later, *exact* matches for these approximate queries are found. Let S_Q denotes the set of approximate queries of Q. Also let the exact result set for a query $q \in S_Q$ be $\mathcal{T}_q$. Finding the top-k diverse results for Q can then be done in two steps. In the first step, we compute top-k diverse results for each $\mathcal{T}_q$ using our efficient STED and diversification algorithm. In the second step, we find the top-k diverse results from the $k|S_Q|$ matches of the first step. As two matches for two different queries in S_Q may not have a complete mapping between them, we use the modified tree edit distance described in 2.2 instead of STED. The number of distances computed in the second step is not significant as $k|S_Q|$ is in the order of hundreds. All the approximate matches are not equally relevant to the query, which necessitates to use relevance score in diversity calculation. We can use relevance score in [16] or any other function suitable for this purpose.

We also have described a novel framework for adaptive diversification of query results that dynamically adjusts the relevance and diversity of displayed results with the aim to minimize the total expected user navigation cost to reach the desired target objects (Chapter 3). Based on this framework, we prove that the problem is NP-hard and we present an efficient approximate algorithm (ADA) that computes the best set results to display, by dynamically balancing relevance and diversity at each query step. We experimentally evaluate the performance of our proposed algorithm and show that it outperforms state-of-the-art algorithms. A Mechanical Turk user study confirms our findings and validates our navigation model. As a future work, we plan to extend our navigation model where the user can update/delete the selected conditions during the navigation process.

We then propose two distributed approaches, *DM* and *SR*, for result diversification on massive datasets (Chapter 4) . The actual winner of these two approaches depends on the environment parameters and data characteristics. Therefore, we propose a cost model that can choose the best approach at runtime. We also propose an iterative refinement component that iteratively refines the diverse result set returned by the diversification approach and guarantees a 2-approximate solution when converses.

Note that, our discussion in Chapter 4 consider diversification on a static dataset; it is an interesting open problem whether we can iteratively identify a good quality diversified answer if the dataset changes over time (i.e., a new element is added to $\mathcal{D}$ or an element is deleted). For further future work, we are planning to add diversification on the Asterix platform (both as an extension of AQL and on top of the Hyracks engine) [20].

6.2 Spatial Top-k Queries

In Chapter 5, we propose a novel indexing approach (STI) for fast answering of top-k spatial queries using threshold algorithm [46]. Our STI approach uses a model to reduce the size of the index structure without sacrificing the performance of the threshold algorithm. In future, we are planning to enhance our STI approach with distributed threshold algorithm (like [24]) to process large volume of data.

Bibliography

[1] Amazon mechanical turk, https://www.mturk.com.

[2] emarketer report http://www.emarketer.com/article/smartphones-tablets-drive-faster-growth-ecommerce-sales/1009835, 2013.

[3] Facebook, https://facebook.com/.

[4] Facebook statistics, http://www.digitalbuzzblog.com/facebook-statistics-stats-facts-2011/.

[5] http://en.wikipedia.org/wiki/twitter.

[6] https://twitter.com/search-home.

[7] Myspace, http://myspace.com/.

[8] Slashdot, http://slashdot.org//.

[9] Tiny image dataset, http://horatio.cs.nyu.edu/mit/tiny/data/index.html.

[10] Twitter, https://twitter.com/.

[11] B. Aditya, Gaurav Bhalotia, Soumen Chakrabarti, Arvind Hulgeri, Charuta Nakhe, Parag Parag, and S. Sudarshan. Banks: Browsing and keyword searching in relational databases. In *VLDB*, 2002.

[12] Rakesh Agrawal, Sreenivas Gollapudi, Alan Halverson, and Samuel Ieong. Diversifying search results. In *Proc. ACM WSDM*, 2009.

[13] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[14] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing Wu. Structural joins: A primitive for efficient xml query pattern matching. In *ICDE*, pages 141–152, 2002.

[15] Sihem Amer-Yahia, Sungran Cho, and Divesh Srivastava. Tree pattern relaxation. In *EDBT*, pages 496–513, 2002.

[16] Sihem Amer-Yahia, Laks V.S. Lakshmanan, and Shashank Pandit. Flexpath: Flexible structure and full-text querying for xml. In *SIGMOD*, pages 83–94, 2004.

[17] Taku Aratsu, Kouichi Hirata, and Tetsuji Kuboyama. Approximating tree edit distance through string edit distance for binary tree codes. In *SOFSEM*, 2009.

[18] Nikolaus Augsten, Denilson Barbosa, Michael Bohlen, and Themis Palpanas. Tasm: Top-k approximate subtree matching. In *ICDE*, 2010.

[19] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[20] Alexander Behm, Vinayak R Borkar, Michael J Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3), 2011.

[21] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, 2010.

[22] Rubi Boim, Tova Milo, and Slava Novgorodov. Diversification and refinement in collaborative filtering recommender. In *CIKM*, 2011.

[23] Nicolas Bruno. Holistic twig joins: Optimal xml pattern matching, 2002.

[24] Pei Cao and Zhe Wang. Efficient top-k query calculation in distributed networks. In *PODC*, pages 206–215, 2004.

[25] Gabriele Capannini, Franco Maria Nardini, Raffaele Perego, and Fabrizio Silvestri. Efficient diversification of web search results. *VLDB*, 4(7), 2011.

[26] Jaime Carbonell and Jade Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *SIGIR*, 1998.

[27] Kaushik Chakrabarti, Surajit Chaudhuri, and Seung won Hwang. Automatic categorization of query results. In *SIGMOD*, 2004.

[28] D Chamberlin. Xquery 1.0: An xml query language.

[29] Surajit Chaudhuri, Gautam Das, Vagelis Hristidis, and Gerhard Weikum. Probabilistic ranking of database query results. In *VLDB*, 2004.

[30] S Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed xml documents. In *VLDB*, pages 263–274, 2002.

[31] Hung chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*, 2007.

[32] J Clark. Xml path language (xpath) version 1.0.

[33] Charles Clarke, Maheedhar Kolla, Gordon V Cormack, Olga Vechtomova, Azin Ashkan, Stefan Büttcher, and Ian MacKinnon. Novelty and diversity in information retrieval evaluation. In *SIGIR*, 2008.

[34] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. Online aggregation and continuous query support in mapreduce. In *SIGMOD*, 2010.

[35] Robson Leonardo Ferreira Cordeiro, Caetano Traina Junior, Agma Juci Machado Traina, Julio López, U Kang, and Christos Faloutsos. Clustering very large multi-dimensional datasets with mapreduce. In *KDD*, 2011.

[36] Maurice Coyle and Barry Smyth. On the importance of being diverse. In *IIP*. 2005.

[37] Van Dang and W. Bruce Croft. Diversity by proportionality: an election-based approach to search result diversification. In *SIGIR*, 2012.

[38] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*. 2004.

[39] Elena Demidova, Peter Fankhauser, Xuan Zhou, and Wolfgang Nejdl. Divq: Diversification for keyword search over structured databases. In *SIGIR*, 2010.

[40] Marina Drosou and Evaggelia Pitoura. Comparing diversity heuristics. In *Technical Report. Computer Science Department, University of Ioannina*, 2009.

[41] Marina Drosou and Evaggelia Pitoura. Diversity over continuous data. *IEEE Data(base) Engineering Bulletin*, 32:49–56, 2009.

[42] Marina Drosou and Evaggelia Pitoura. Search result diversification. In *ACM SIGMOD Record*, 2010.

[43] Marina Drosou and Evaggelia Pitoura. Disc diversity: result diversification based on dissimilarity and coverage. *VLDB*, 6(1), 2012.

[44] Marina Drosou, Kostas Stefanidis, and Evaggelia Pitoura. Preference-aware publish/subscribe delivery with diversity. In *DEBS*, 2009.

[45] Alina Ene, Sungjin Im, and Benjamin Moseley. Fast clustering using mapreduce. In *SIGKDD*, 2011.

[46] Ronald Fagin. Combining fuzzy information from multiple systems. In *PODS*, pages 216–226, 1996.

[47] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, 2003.

[48] Sreenivas Gollapudi and Aneesh Sharma. An axiomatic approach for result diversification. In *WWW*, pages 381–390, 2009.

[49] Sudipto Guha, H V Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. Approximate xml joins. In *SIGMOD*. ACM, 2002.

[50] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[51] Marios Hadjieleftheriou and Vassilis J Tsotras. (eds.) Result Diversity. *IEEE Data Engineering Bulletin*, 32(4), 2009.

[52] Jayant R. Haritsa. The kndn problem: A quest for unity in diversity.

[53] Mahbub Hasan, Abhijith Kashyap, Vagelis Hristidis, and Vassilis Tsotras. User effort minimization through adaptive diversification. *Submitted for publication*.

[54] Mahbub Hasan, Abdullah Mueen, and Vassilis Tsotras. Distributed diversification of large datasets. In *IC2E*, 2014.

[55] Mahbub Hasan, Abdullah Mueen, Vassilis Tsotras, and Eamonn Keogh. Diversifying query results on semi-structured data. In *CIKM*, 2012.

[56] Vagelis Hristidis, Heasoo Hwang, and Yannis Papakonstantinou. Authority-based keyword search in databases. *ACM Trans. Database Syst.*, 33(1):1:1–1:40, 2008.

[57] Giacomo Inches, Mark J. Carman, and Fabio Crestani. Statistics of online user-generated short documents. In *ECIR*, pages 649–652, 2010.

[58] Anoop Jain, Parag Sarda, and Jayant R Haritsa. Providing diversity in k-nearest neighbor query results. In *PAKDD*. 2004.

[59] Haifeng Jiang, Weii Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins on indexed xml documents. In *Proc. of VLDB*, pages 273–284, 2003.

[60] Abhijith Kashyap, Vagelis Hristidis, and Michalis Petropoulos. Facetor: Cost-driven exploration of faceted query results. In *CIKM*, 2010.

[61] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *SPAA*, 2011.

[62] Vladimir Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–10, 1966.

[63] Kun Liu, Evimaria Terzi, and Tyrone Grandison. Highlighting diverse concepts in documents. In *SDM*, 2009.

[64] Ziyang Liu and Yi Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD*, pages 329–340. ACM, 2007.

[65] Ziyang Liu, Peng Sun, and Yi Chen. Structured search result differentiation. In *VLDB*, 2009.

[66] Abdullah Mueen, Suman Nath, and Jie Liu. Fast approximate correlation for massive time-series data. In *SIGMOD*, 2010.

[67] Danupon Nanongkai, Ashwin Lall, Atish Das Sarma, and Kazuhisa Makino. Interactive regret minimization. In *SIGMOD*, 2012.

[68] Surya Nepal and M. V. Ramakrishna. Query processing issues in image(multimedia) databases. In *ICDE*, pages 22–29, 1999.

[69] Lu Qin, Jeffrey Xu Yu, and Lijun Chang. Diversifying top-k results. *VLDB*, 5(11):1124–1135, 2012.

[70] Filip Radlinski and Susan Dumais. Improving personalized web search using result diversification. In *SIGIR*, 2006.

[71] Davood Rafiei, Krishna Bharat, and Anand Shukla. Diversifying web search results. In *WWW*, 2010.

[72] Alexander Rasmussen, Michael Conley, Rishi Kapoor, Vinh The Lam, George Porter, and Amin Vahdat. *ThemisMR: An I/O-Efficient MapReduce*. Technical Report CS2012-0983, UCSD, 2012.

[73] Senjuti Basu Roy, Haidong Wang, Gautam Das, Ullas Nambiar, and Mukesh Mohania. Minimum-effort driven dynamic faceted search in structured databases. In *CIKM*, 2008.

[74] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.

[75] Rodrygo L.T. Santos, Craig Macdonald, and Iadh Ounis. Selectively diversifying web search results. In *CIKM*, 2010.

[76] Manish Singh, Arnab Nandi, and H. V. Jagadish. Skimmer: Rapid scrolling of relational query results. In *SIGMOD*, 2012.

[77] Barry Smyth and Paul McClave. Similarity vs. diversity. In *ICCBR*. 2001.

[78] Shirish Tatikonda, Srinivasan Parthasarathy, and Matthew Goyder. Lcstrim: Dynamic programming meets xml indexing and querying. In *VLDB*, 2007.

[79] Yannis Theodoridis and Timos Sellis. A model for the prediction of r-tree performance. In *PODS*, pages 161–171, 1996.

[80] Reinier H van Leuken, Lluis Garcia, Ximena Olivares, and Roelof van Zwol. Visual diversification of image search results. In *WWW*, 2009.

[81] E Vee, U Srivastava, J Shanmugasundaram, P Bhat, and S Amer-Yahia. Efficient computation of diverse query results. In *ICDE*, pages 228–236, 2008.

[82] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, 2010.

[83] Marcos R. Vieira, Humberto Luiz Razente, Maria Camila Nardini Barioni, Marios Hadjieleftheriou, Divesh Srivastava, Caetano Traina, and Vassilis J. Tsotras. On query result diversification. In *ICDE*, pages 1163–1174, 2011.

[84] Cong Yu, Laks Lakshmanan, and Sihem Amer-Yahia. It takes variety to make a world: diversification in recommender systems. In *EDBT*, 2009.

[85] Yisong Yue and Thorsten Joachims. Predicting diverse subsets using structural svms. In *ICML*, 2008.

[86] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008.

[87] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *Siam Journal on Computing*, 18:1245–1262, 1989.

[88] Mi Zhang and Neil Hurley. Avoiding monotony: Improving the diversity of recommendation lists. In *RecSys*, 2008.

[89] Cai Nicolas Ziegler, Sean M. Mcnee, Joseph A. Konstan, and Georg Lausen. Improving recommendation lists through topic diversification. In *WWW*, 2005.