EFFICIENT COMPUTATION OF DIVERSE QUERY RESULTS

LI LU

NATIONAL UNIVERSITY OF SINGAPORE

2015

NATIONAL UNIVERSITY OF SINGAPORE

DOCTORAL THESIS

EFFICIENT COMPUTATION OF DIVERSE QUERY RESULTS

Author:
LI Lu

Supervisor:
Prof. CHAN Chee-Yong

A thesis submitted
for the degree of Doctor of Philosophy

in the

Department of Computer Science
School of Computing

2015

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety.

I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Li Lu

July, 2015

ACKNOWLEDGEMENT

This thesis is the majority part of the research work done in my six-year Ph.D. period. In the period, I have received helps from many people. I will take this opportunity to thank them.

I would like to express the deepest appreciation to my supervisor, Prof. Chan Chee-Yong. Without his guidance and persist help, my thesis would not have been finished. During the last six years, he has spent countless time to patiently guide me to build interesting ideas, strengthen the algorithms and improve the writings. As a supervisor, he shows his wisdom, insights, wide knowledge and conscientious attitude. All of these set me a good example to be a good researcher.

I would like to sincerely thank Prof. Tan Kian Lee and Prof. Stephane Bressan, who are members of my evaluation committee and provided me with constructive feedback to refine my research work.

I would also thank the group members from the NUS database research group: Bao Zhifeng, Wang Guoping, Wang Zhengkui, Zhang Dongxiang, Li Hao, Zeng Zhong, Kang Wei, Zhou Jingbo, Wu Huayu, Zheng Yuxin, Tang Ruiming, Song Yi, Zeng Yong, Xiao Qian, etc.

Last but not least, I would like to thank my family: my father Li Junming, my mother Niu Yueling, my brother Li Wei, my sister Li Qi and my wife Wang Haiyan, for their invaluable support and understanding throughout my six-year Ph.D. study.

CONTENTS

Abstract

Query result diversification aims to enhance the quality of query results presented to users by ranking the results based on diversity so that more informative results are presented first. In this thesis, we study three problems related to the efficient computation of diverse query results. Firstly, we study the problem of evaluating diversity queries in the context of relational database systems where query results are diversified with respect to a sequence of attributes (known as the d-order) such that attributes that appear earlier in the d-order have higher priority for result diversification. We design a new indexing technique (termed D-Index), which is based on a trie-like structure, to efficiently evaluate diversity queries. Our experimental evaluation demonstrates that the D-Index not only outperforms the state-of-the-art techniques by up to a factor of 2.7 for diversity queries with static d-orders but also outperforms baseline techniques by up to a factor of 3.5 for diversity queries with dynamic d-orders.

Secondly, we study the optimization problem of evaluating multiple diversity queries in an online environment, and develop three new evaluation techniques. The first optimization technique aims to improve query response time by judiciously reordering queries to increase opportunity for shared index scans. The second optimization is an adaptive query evaluation technique that enables an existing running query to dynamically switch to a different index scan that is used for evaluating a new query. The third optimization is an online index tuning technique that leverages the results of an index scan evaluation to create a new index at the same time. Our experimental evaluation demonstrates that our proposed optimizations can improve performance by up to a factor of 2.

Finally, we study the novel problem of computing diverse query results in the context of spatial keyword search which is useful for applications such as trip-planning. We introduce two new types of spatial keyword queries to compute top-k diversified result groups where each result group is a collection of closely located objects that match the

specified keywords. The first type of query diversifies the result groups based on the semantic diversity of the objects while the second type of query additionally diversifies the spatial locations of the result groups. We propose a novel Quadtree-based indexing technique (termed OQ-tree), which uses both overlapping space decompositions as well as precomputed summary information, to efficiently evaluate both types of spatial keyword queries. Our experimental evaluation demonstrates that the OQ-tree outperforms baseline techniques by up to a factor of 20.

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Query result diversification aims to enhance the quality of query results presented to users by ranking the results based on diversity so that more informative results are presented first. This thesis studies three problems related to efficiently compute diverse query results: efficient indexing for diverse query results, efficient processing of multiple diversity queries and diversified spatial keyword search.

In this chapter, we first present some background on query result diversification. We then state the three studied problems and contributions of this thesis. We finally describe the thesis organization.

1.1 Query Result Diversification

Consider a query with a large number of relevant results. Rather than directly returning all of these relevant results, an effective strategy is to choose and show a set of representative relevant results. One traditional strategy is to find the top-k results based on a pre-defined ranking function which only takes into account the relevance between each result and the query. For a top-k query, they simply assume that the relevance of results is independent with each other. Some of the top-k relevant results, however, could be very similar with each other. Zhai et al. [78] point out that it is insufficient to simply return a set of relevant results, since the correlations among the results are also very important. It has been noticed that a large fraction of search queries are short and thus ambiguous or under-specified [29]. For these queries, the targeted information for the same query could be quite different given different users. For example, a simple ambiguous query "apple" could be relevant to both Apple company and the fruit apple, and a non-ambiguous but under-specified query "laptop" could be relevant to a Lenovo laptop or a Acer laptop. Instead of showing a homogeneous collection of similar results, recently, an amount of existing works [19, 62, 5, 70, 57, 71] study the problem of query result diversification to satisfy different information needs of users, by taking into account both the relevance of each result (wrt the query) as well as the dissimilarity among these results. The result diversification has been studied on many different kinds of databases, such as web document datasets [19, 62, 5, 18, 9, 71, 57, 24, 83], structured databases [42, 70, 34], graph databases [40, 45], streaming data [53], time series data [39], spatial datasets [79], social networks [27], recommender systems [14, 88] and so on. In general, the research area of query result diversification can be broadly classified into *intent-based diversification* (e.g., [77, 88, 18, 57]) which aims to provide search results that cover as many facets of the query as possible to deal with ambiguous queries, and *content-based diversification* (e.g., [71, 62, 9]) which aims to reduce information redundancy in search results.

In the intent-based diversification, the diversified results are covering different intents

to ensure that all users are satisfied, with the hope that the user will find at least one relevant result for his information needs. For example, reconsider the simple ambiguous query "apple". A user might be interested in the fruit apple, while another user could be interested in the products of the Apple company. It would be much helpful for different users to show results, where some are relevant to the fruit apple and some others are relevant to the Apple products. To minimize the average user dissatisfaction, user intents are often modeled as a set of sub-topics (or categories) [5, 77], based on the analysis of collected usage statistics. Some probabilistic models are used to diversify query results to cover as many relevant subtopics as possible.

Differently, the content-based diversification model does not focus on ambiguous queries, and most of existing works [19, 5, 38, 70, 71] in this model attempt to reduce information redundancy by taking into account the dissimilarity (distance) between every two results. For example, reconsider the non-ambiguous but under-specified query "laptop". Instead of showing laptops from only two brands (say Lenovo and Acer), it would be more interesting to display laptops covering a more diverse range of brands (e.g. Lenovo, Acer, Dell, HP, Samsung and so on). To reduce information redundancy, some different distances have been used, such as the Euclidean distance, the explanation-based distance [74], Jaccard dissimilarity function [42], the cosine dissimilarity function, taxonomy-based categorical distance [42], the pre-defined attribute ordering-based distance function [70], and so on. Furthermore, some recent works [67, 79] study the special case of spatially diversifying query results based on the geo-location based distance function (i.e. the Euclidean distance on the geo-locations [67], the road network based distance [79]).

1.2 Research Problems

In this thesis, we study three research problems to efficiently compute the diverse query results, namely, efficient indexing for dynamic diversity queries, evaluation of multiple di-

versity queries, and diversified spatial keyword search. Note that all of the three problems fall under the content-based diversification.

1.2.1 Indexing for Dynamic Diversity Queries

Consider a user who is shopping online for a new laptop from a website which can display a result table consisting of up to 20 laptops that match the user's specification. As the number of matching results is typically much larger than number of display records, it is useful to return a diverse set of results for the user to browse. For example, instead of showing the user 20 laptops from only two brands (say Lenovo and Acer), it would be more interesting to show results covering a more diverse range of brands (e.g., Lenovo, Acer, Dell, HP, Asus, Samsung). If Lenovo and Acer are indeed the only two brands of laptops that satisfy the user's query, then it would be better to show a more "balanced" distribution of the 20 displayed laptops; for example, showing 10 laptops from each of Lenovo and Acer is better than showing 18 laptops from Lenovo and 2 laptops from Acer. Similarly, if the user is interested only in laptops from Dell, then it would be more interesting to show a diverse range of Dell laptops with different screen sizes instead of showing all Dell laptops with the same screen size.

In this problem, we try to diversify query results with respect to a sequence of attributes, referred to as a *d-order*, where the intention is to first diversify the results with as many different values of the first attribute as possible, and for records with the same attribute value of the first attribute, we diversify them with as many different values of the second attribute as possible, and so on. Thus, a d-order determines a priority order for diversifying the query results, where the first attribute has higher priority to diversify than the second attribute, and so on.

Vee et.al. [70] were the first to study the problem of computing diverse query results. They formally define the notion of query result diversity and show that existing score

ID	Brand	#Core of CPU	Screen Size	Battery Life	Color
1	HP	1	13.3	3	Red
2	HP	1	14.1	7	White
3	HP	2	14.1	3	Silver
4	HP	2	14.1	5	Silver
5	HP	2	14.1	7	Black
6	HP	2	15.4	3	Red
7	Acer	2	14.1	6	White
8	Acer	2	15.4	3	Silver
9	Acer	2	15.4	7	Red
10	Acer	4	13.3	3	Black
11	Acer	4	13.3	5	Black
12	Acer	4	14.1	5	Red
13	Acer	4	17.3	5	Black
14	Lenovo	2	14.1	3	White
15	Lenovo	2	14.1	5	Silver
16	Lenovo	2	14.1	7	Black
17	Lenovo	4	13.3	5	Black
18	Lenovo	4	13.3	7	White

Table 1.1: Laptop Example

based techniques are inadequate to guarantee diverse query results. They also propose an inverted-list based approach to evaluate such queries. However, their work addresses only *static diversity queries* (SDQs), where the query results are diversified wrt a static, pre-defined d-order. Clearly, it would be useful to allow users to customize their diversification preference. For example, Alice might be more interested to diversify the results wrt screen size first, followed by brand, whereas Bob might be more interested to diversify the results wrt brand first, followed by the number of CPU cores and screen size. Consider the running example shown in Table 1.1: the attributes Brand, #Core of CPU, Screen Size, Battery Life, and Color represent, respectively, laptop brand (B), number of CPU cores (C), screen size in inches (SS), battery life in hours (BL), and laptop color (LC). Let us simply assume that we can only show four results at a time. Among the two sets of laptops shown in Figure 1.1, it would be helpful for Alice to show the diverse set S_2 that contains four laptops with different screen sizes, rather than returning the other set S_1 that contains four laptops with the same screen size. On the other hand, Bob would be interested in the diverse set S_1, where the four laptops contain three different brands, and the two Acer

laptops are with different screen sizes. The other set S_2, however, is not as diverse since the four laptops only contain two kinds of brands.

RID	Brand	#Core	Screen Size
2	HP	1	14.1
7	Acer	2	14.1
12	Acer	4	14.1
14	Lenovo	2	14.1

(a) S_1

RID	Brand	#Core	Screen Size
7	Acer	2	14.1
13	Acer	4	17.3
14	Lenovo	2	14.1
17	Lenovo	4	13.3

(b) S_2

Figure 1.1: Two example diverse result sets

In this thesis, we examine the more general problem of evaluating *dynamic diversity queries (DDQs)* where each user's query results are diversified wrt a user specified d-order. Although we can extend the techniques designed for SDQs in [70] to evaluate DDQs, the extended techniques are very inefficient for evaluating DDQs. Thus, in this work, we study the problem of efficiently evaluating both SDQs as well as DDQs.

1.2.2 Evaluation of Multiple Diversity Queries

Popular online web services such as online shopping websites need to cope with high transaction throughput. For instance, *Amazon* has to process about 0.44 billion requests from around 53 million customers each day [2], while *Taobao* processes about 0.27 billion requests from around 27.1 million customers per day [3]. For multiple online DDQs, it is suboptimal to independently evaluate each DDQ. Therefore, in this thesis, we study the optimization problem for multiple online diversity queries.

Different from the traditional multiple query optimization techniques [65, 60, 55, 86] for an offline query workload, it is quite challenging to share processing among a newly arrived query and other running queries. Lang et.al. [49] first study the techniques of shared index scan to optimize the evaluations of multiple queries. For example, consider the scenario that a system scans index I to evaluate some queries. As can be seen in Figure 1.2, when picking a newly arrived query Q which can also be evaluated by scanning index I,

the system marks the current accessing point on index I, followed by scanning the index to evaluate both query Q and some other running queries. After reaching the end of index I, the system can complete the evaluation of query Q, by re-scanning index I from the beginning until reaching the marked point. However, sometimes it would be suboptimal to shared scan the current accessing index to evaluate these queries. In this thesis, we study a new technique to switch the query evaluation to scan another index. Instead of shared scanning the current index, we are able to scan a new index to concurrently evaluate these queries, by switching their evaluations to scan the new index. To improve the opportunity of shared index scan, we further present a new framework by allowing each incoming query to be reordered, rather than simply processing them in the first-come-first-serve order.

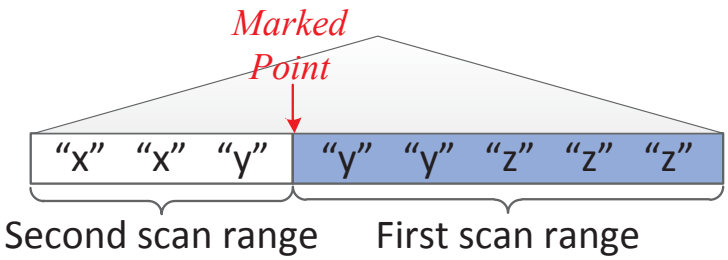


Figure 1.2: Index Sharing Scan

In the online environment, the characteristics of online diversity queries at different time periods could be very different, and a fixed set of indexes might not be optimal for all of these queries. Rather than tuning the set of physical indexes for the offline query workload [7, 22, 6, 32, 89, 8], our system is able to automatically self-tune the index set to improve the performance of future queries. Instead of assuming that the characteristics of recent queries are similar to the characteristics of queries in the near future [15, 63, 64], we generate new indexes by exploiting knowledge of those queries in the waiting queue. Moreover, to minimize the IO cost, the index generation is able to share index scan with other running queries.

In this thesis, we examine the optimization problem of concurrently evaluating multiple

online DDQs. However, evaluating these DDQs independently does not take advantage of the shared index scans. To efficiently evaluate these online DDQs, in this work, we study the optimization problem by introducing some new techniques.

1.2.3 Diversified Spatial Keyword Search

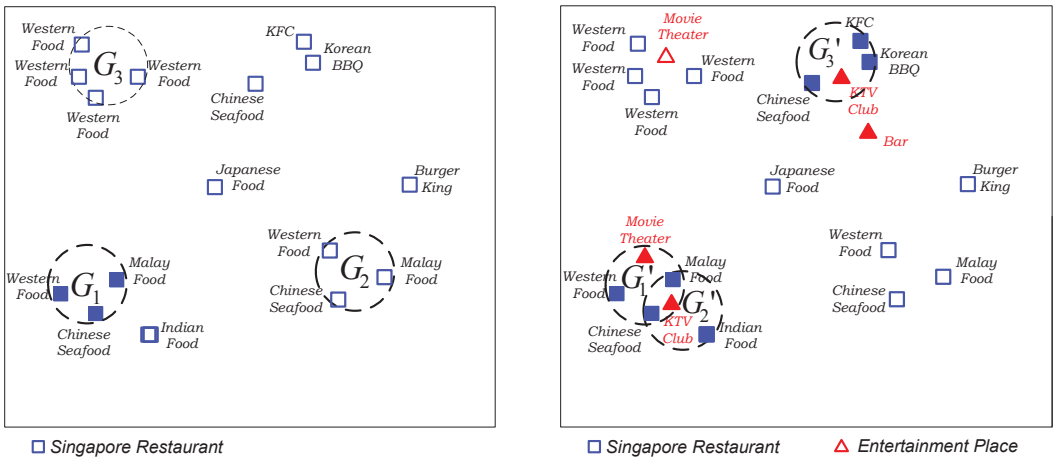
With the prevalence of geo-position devices GPS, a huge number of spatial objects associated with textual information are publicly accessible. In Foursquare, users have contributed millions of venues associated with tips, reviews and category information [1]. Spatial keyword search is now a very popular service that helps users explore local restaurants, hotels and entertainment facilities. Existing works on this topic mainly focus on how to conduct ranking and filtering using spatial and textual attributes. To improve search performance, various indices have been proposed to facilitate spatial and textual pruning simultaneously. However, existing ranking functions for spatial keyword queries do not take into account of the *semantic diversity* of the query results. For a group of spatial objects, the *semantic diversity* refers to the degree of the textual information variation of these group objects.

Some existing works [87, 30] study the spatial keyword search problem to find the top-k spatial objects by considering each spatial object in isolation. These returned spatial objects could be relatively far from each other. If one spatial fails to satisfy users, it could be quite inconvenient to consider some others. Instead, S. Bogh et.al. [12] study the problem of finding the top-k groups of objects that are closely located. These returned groups, however, could contain too many objects with redundant information, since they have not taken into account of the *semantic diversity* of objects in each group. For example, consider group G_3 in Figure 1.3(a). The four "Western Food" restaurants could be much similar with each other. In this thesis, we propose two novel types of diversified queries against spatial object databases that extend the ranking function to incorporate the *seman-*

tic diversity information. To ensure the spatial proximity of each group, a user-specified radius r can be used to guarantee that all objects in each groups are within a circle of radius r. The value of radius r could be determined by the user's transportation modes, such as by car, by bicycle or by foot.

Given a set of query keyword concepts, each of which could contain multiple keywords (e.g. "*Singapore Restaurant*") , a query radius r, the query region R_Q and the limit size k, our first query, named *Diversity Spatial Query* (DSQ), finds the top-k groups of spatial objects in R_Q with high spatial proximity (located within a circle of radius r) and semantic diversity.

Example 1.1: Consider a group of tourists who are planning a trip to Singapore and wish to stay at a convenient hotel so that they could favour different cuisines. The tourists can submit a simple DSQ query with keyword concept "*Singapore Restaurant*", a radius r and the limit size 2. Figure 1.3(a) shows an example query region containing various Singapore restaurants. In Figure 1.3(a), the top-2 groups G_1 and G_2 are returned for the query. Each of the two groups contains three restaurants that provide different cuisines and are located within a circle of radius r. The tourists can choose a desired one from the top-2 groups, and then book a hotel near this selected group of restaurants so that it is convenient for them to try out many different cuisines with minimum transportation overhead. Observe that there is another group G_3 (shown in Figure 1.3(a)) that contains one more restaurant than G_1 and G_2. Group G_3, however, is not preferable since the four results in G_3 provide similar western food. The group might not well satisfy users who do not like western food. Consider another scenario where the tourists would like to attend some entertainment activities after dining. They can issue a DSQ query with keyword concepts "*Singapore Restaurant*" and "*Entertainment Facility*". In Figure 1.3(b), we show the top-2 groups G'_1 and G'_2, and each group contains both Singapore restaurants and entertainment facilities. □



(a) Singapore Restaurants (b) Singapore Restaurants & Entertainments

Figure 1.3: Example spatial objects

Some of the top-k groups for a DSQ query, however, could be highly spatially overlapped. For example, the top-2 groups G_1' and G_2' (shown in Figure 1.3(b)) for the previous DSQ query are closely located, and they share two common restaurants ("*Chinese Seafood Restaurant*" and "*Malay Food Restaurant*") and one common entertainment facility ("*KTV Club*"). In some scenarios, it is less interesting for users to retrieve some highly overlapped similar groups at the same time. Therefore, this motivates us to propose our second query, named *Non-overlapping Diversity Spatial Query* (N-DSQ), that extends DSQ to take into account of spatial diversity for the top-k groups. For example, consider the extended N-DSQ query with keyword concepts "*Singapore Restaurant*" and "*Entertainment*", a radius r and the limit size 2. In Figure 1.3(b), two spatially diversified groups G_1' and G_3' are returned.

In this thesis, we examine the problem of diversified spatial keyword search by designing two novel spatial diversity keyword queries. Unfortunately, existing spatial indexes are inefficient to answer these queries. To efficiently evaluate them, we introduce a new textual-first spatial index, named IOQ-tree, where each inverted postings list corresponding to a keyword concept is organized based on a novel structure OQ-tree with two variants. For each type of spatial keyword queries, we propose two efficient evaluation methods based on the two variants of index.

1.3 Thesis Contributions

In this thesis, we make the following three contributions.

Indexing for dynamic diversity queries. In this work, we study the problem of efficient indexing for diverse query results, and show that extending existing techniques designed for SDQs [70] to evaluate DDQs are inefficient.

Subsequently, we introduce a novel approach for evaluating diversity queries that is based on the concept of computing a core cover of a query. Based on this concept, we design a new index method, D-Index, and introduce two index variants, namely, `D-tree` and `D`$^+$`-tree`.

Finally, we demonstrated with an experimental evaluation, which is based on a PostgreSQL implementation, that our proposed D-Index technique consistently outperforms [70] for both SDQs as well as DDQs.

This work has been published in VLDB 2013 [50].

Evaluation of multiple diversity queries. In this work, we study the optimization problem of concurrently evaluating multiple online DDQs. A new framework is proposed to optimize multiple online queries by allowing each query to be reordered.

To improve the opportunity for shared index scans among multiple queries, we propose a novel technique to dynamically adapt the query plans by switching query evaluations to scan another inactive index. Furthermore, we also introduce a technique of online index tuning to automatically adapt the set of physical indexes by looking-ahead at waiting queries.

Finally, we implemented our approach on PostgreSQL and conducted a comprehensive performance evaluation of multiple online diversity queries to show the efficiency of our proposed techniques.

Diversified spatial keyword search. In this work, we study the problem of diversified spatial keyword search. We first propose two novel spatial diversity keyword queries: DSQ and N-DSQ, and show that existing spatial indexes are inefficient to evaluate these newly proposed spatial queries.

Subsequently, we then introduce a new textual-first spatial index, named IOQ-tree, where each inverted postings list corresponding to a keyword concept is organized based on a novel spatial tree structure OQ-tree with two variants (OQ^+-tree and OQ^*-tree). Based on the two variants of IOQ-tree, two evaluation methods are proposed to efficiently evaluate each spatial query.

Finally, we conducted a comprehensive experimental study to demonstrate the efficiency of our proposed algorithms for these proposed spatial queries. Our experimental results on two real datasets (*Foursquare* and *Tweets*) show that our proposed techniques outperform the state-of-the-art technique [82] by up to one order of magnitude.

1.4 Thesis Organization

The rest of the thesis is organized as follows.

- Chapter 2 describes the related works of the thesis.
- Chapter 3 studies the evaluation problem for DDQs and proposes efficient index-based techniques to evaluate DDQs.
- Chapter 4 studies the optimization problem for multiple online DDQs and proposes efficient techniques to optimize these online DDQs.
- Chapter 5 studies the problem of diversified spatial keyword searching with two newly proposed spatial queries (DSQ and N-DSQ), and proposes efficient indexing techniques for the evaluation of these spatial queries.

- Chapter 6 concludes our thesis and points out some directions for future work.

CHAPTER 2

RELATED WORK

In this chapter, we describe some related works. More specifically, we first present some related works on the query result diversification. Subsequently, we describe some related works on query processing techniques. Finally, we present some related works on diversified spatial keyword search.

2.1 Query Result Diversification

Search result diversification is an active research area that aims to increase user satisfaction in web search and recommender systems (e.g., [4, 37]). The result diversification can be broadly classified into *content-based diversification* (e.g., [71, 62, 9, 14, 36, 78]) which aims to reduce information redundancy in search results, and *intent-based diversification*

(e.g., [77, 88, 18, 57]) which aims to provide search results that cover as many facets of the query as possible to deal with ambiguous queries.

Now we describe some related works of *content-based diversification* and *intent-based diversification*, followed by presenting some diversification models.

2.1.1 Content-based Diversification

The content-based diversification (e.g. [71, 62, 9, 14, 36, 78]) is to reduce information redundancy in the selected results; this is accomplished by avoiding to return results that offer little new information to the user, based on the already examined results.

Most of existing works [19, 5, 38, 70, 71] attempted to reduce information redundancy by taking into account the dissimilarity (distance) between every two results. Based on some traditional IR models, they first generated a candidate set of most relevant results, followed by re-ranking these results based on their own objective functions for different applications.

Furthermore, some existing works [46, 26] used the clustering techniques to avoid the information redundancy. They first grouped these relevant results into different clusters, followed by generating the diverse result set by picking one representative result from each cluster. Sarma et al. [62] studied the diversification approach to analyze click logs by examining both the clicked results and the bypassed results, which are skipped by the user. They proposed several greedy algorithms to minimize the bypass rate of the selected result sets. In these greedy algorithms, similar results are grouped into the same cluster, and at most one result is picked from each one cluster.

Vee et.al. [70] first studied the problem of evaluating *static diversity queries* (SDQs) based on a pre-defined sequence of attributes, where the first attribute has higher priority

to diversify than the second attribute, and so on. They showed that existing score based information retrieval techniques are inadequate for the problem and proposed two indexing methods, `OnePass` and `Probe`. To evaluate SDQs on a relation R, `OnePass` builds an inverted-list index I_j for each attribute A_j in R, where each postings list in I_j is organized using a B$^+$-tree with a pre-determined d-order, $\alpha = (A_1, \cdots, A_n)$, which consists of all the attributes in R, as the index key. Thus, all the B$^+$-trees in `OnePass` use α as the index key. The B$^+$-trees are compressed by replacing each key attribute value with a Dewey encoded value (e.g., replace "Acer" by the value 0). Given a SDQ Q with a selection predicate "$A_j = v$", `OnePass` evaluates Q by an index scan on the B$^+$-tree corresponding to the value v in I_j. A run-time, a main-memory trie structure T is used to organize the retrieved index key values such that each root-to-leaf path in T represents a retrieved α-tuple. Since the index key and query's d-order are both the same (i.e., α) for SDQs, the B$^+$-tree index scan ensures that the retrieved key values are inserted into T "sequentially" by extending T with a rightmost path. This important property enables `OnePass` to conveniently detect when there is a sufficient number of α-tuples in a subtrie to form a diverse result set so that the B$^+$-tree index scan can skip to retrieve tuples for another subtrie in T. As an example, suppose that $\alpha = (A, B, C, D)$ and after inserting a newly retrieved tuple (a_1, b_1, c_1, d_1) into T, `OnePass` detects that the subtrie rooted at (a_1, b_1) has sufficient number of tuples; in this case, the index scan will skip to search for index keys greater than $(a_1, b_1, c_\infty, d_\infty)$, where c_∞ and d_∞ represent the largest domain values for attributes C and D, respectively. To deal with multiple selection predicates on different attributes, `OnePass` invokes a B$^+$-tree index scan for each of the selection attributes and uses an appropriate merge operation to combine the index keys retrieved from the multiple index scans. `Probe` is a variant of `OnePass` that performs a bi-directional B$^+$-tree index scan instead of the single forward scan adopted in `OnePass`. The goal of `Probe` is to reduce the number of *useless* retrieved tuples, which are tuples that are retrieved into T but are later replaced by other tuples. However, `Probe` incurs more random I/Os due to its bi-directional scan and the experimental results in [70] indicate that

both `OnePass` and `Probe` performed similarly.

2.1.2 Intent-based Diversification

Different from content-based diversification, the goal of the intent-based diversification is to return a good variety of results covering different intents to ensure that all users are satisfied, with the hope that the user will find at least one relevant result for his information need.

Zhai et al. [78] studied both novelty and relevancy in the language modeling framework, and proposed an evaluation metrics for subtopic retrieval, based on the metrics of subtopic recall and subtopic precision. In a later work [77], they formalized and proposed a risk minimization approach that allows an arbitrary loss function over a set of returned objects to be defined. The loss function is to determine the dissatisfaction of the user wrt the set of selected objects.

Chen and Karger [24] used the standard IR techniques to improve diversity for ambiguous queries. In this work, objects are selected sequentially according to the object relevance score. The relevance is conditioned on objects having been already selected. Words in the text of previous selected objects are associated with a negative weight to improve novelty.

Agrawal and Gollapudi [5] generated a taxonomy of information, based on the analysis of query logs. Based on the taxonomy, each query and result can be represented as a distribution over a set of categories of the taxonomy. A greedy approach was proposed to minimize the risk of dissatisfaction of the average user, by finding a set of results to capture as many categories of the user intents as possible.

Instead of simply satisfying as many categories as possible, Capannini et al. [18] designed an approach to maximize the weighted coverage of the categories with relevant results. It

is possible that there are many results related to a category that is a dominant interpretation of the query, but there does not exist a result related to some unimportant categories.

Radlinski et al. [58] proposed an approach to directly learn a diverse ranking of results based on users' clicking behavior through online exploration. Since users tend not to click on similar results, online learning produces a diverse set of results naturally. The approach is to maximize the probability that a relevant result is found in the top-k positions.

Since user intents are not well represented in the original results, Radlinski and Dumais [57] proposed an approach to understand the variety of user intents based on the query reformulations instead of the topic categorization. Given a search query, the approach first generates a set of more specific related queries, followed by collecting several results for each generated specific query. Then the diverse set of results can be generated by re-ranking the candidate set which is the union set of results for those specific related queries.

2.1.3 Diversification Models

To improve the probability that a user can find at least one relevant result from the returned results, several different diversification models were proposed by existing works. In this section, we then present some well-known diversification models: *combinatorial optimization models*, *probabilistic language models*, *coverage-based diversification models* and *distance-based diversification models*. Existing works [70, 5] studied the problem of diversifying query result based on one or more of these diversification models. For example, result diversification for SDQs [70] belongs to the *distance-based diversification models*, and the diversification approach proposed in [5] uses both the *probabilistic language model* and the *coverage-based model*.

Our first two works fall under the *distance-based diversification model*, while the third

work falls under both the *combinatorial optimization model* and the *coverage-based diversification model*.

Combinatorial Optimization Models

Result diversification involves a trade-off between having more relevant results and having diverse results in the top positions for a given query [19, 24]. The early work of Gollapudi and Goldstein [19] first studied the result diversification problem, and proposed the Maximal Marginal Relevance (MMR) ranking strategy based on the linear combination of query-relevance and information-novelty. Gollapudi and Sharma [42] proposed an axiomatic approach to characterize the problem of result diversification with several different combination functions: *max-sum diversification*, *max-min diversification* and *mono-objective formulation*. More specifically, the first objective function *max-sum diversification* is to maximize the sum of the relevance and dissimilarity of the selected set, the second objective function *max-min diversification* is to maximize the minimum relevance and dissimilarity of the selected set, and the third objective function *mono-objective formulation* is to maximize the "global" importance (i.e. not with respect to any selected set, but with respect to the set of all relevant results) of each result. The first two objective functions can be reduced to the *p-dispersion* problem [56] which is a NP-Complete problem. In [42], a 2-approximation algorithm was proposed to address the problem. For the third objective function, the optimization algorithm is to compute the weight of each result, followed by picking the top-k results.

Different from the traditional diversification problems where the pair-wise distance is measured by some dissimilarity functions on the feature dimensions of each object, some recent works [54, 67, 69, 21, 79] studied the spatial diversity problem to find a set of relevant spatial objects which are well spread in the search region, by utilizing some spatial distance functions on the object geo-locations (latitude and longitude), such as the

Euclidean distance [69, 21] and the network distance based on a given road map [79]. In this thesis, for a N-DSQ query, we use the Euclidean distance to spatially diversify query result groups.

Probabilistic Language Models

To minimize query abandonment, which is the case that a user can not find any relevant result in the selected results, some existing works [24, 77, 62] studied the result diversification problem based on some probabilistic language models.

Agrawal et al. [5] proposed an objective function based on only the diversity score, which is estimated with the probability that the document set would satisfy the user who issues the query. The probabilities are estimated based on a classification taxonomy.

Santo et al. [61] proposed an objective function based on the relevance of documents to query subtopics and the importance of query subtopics in a probabilistic framework.

Coverage-based Diversification Models

Some existing works [85] studied the diversification problem by computing the diversity score based on the coverage of query subtopics (or named information nuggets and query aspects). To define the coverage function which measures how well a result set covers the information of each query subtopic, Zheng et al. [85] proposed three strategies: *summation-based coverage function*, *loss-based coverage function* and *measure-based coverage function*. The *summation-based coverage function* is to sum up the coverage scores of the individual results. The *loss-based coverage function* is measured by the coverage of results that are not included in the selected result set. The *measure-based coverage function* is based on the evaluation diversity measures over these subtopics (i.e. Precision-IA [5], ERR-IA, α-nDCG [29] and NRBP [29]).

Distance-based Diversification Models

Qin et.al. [5] proposed a framework to handle the diversified top-k search problem. Given a lower-bound threshold τ for similarity, the database can be modeled as an undirect graph, where each node represents a database record. Each edge of two nodes represents that the similarity between these two corresponding records is no less than τ, while the weight of each node represents the relevance of the query result. Therefore, the diversified top-k search problem can be reduced to find an independent set of k nodes with maximum weight sum.

Instead of simply returning k results, Drosou et al. [38] proposed a new model to generate a diverse set of results given a threshold (or named radius) r. For a query, let U denote the set of results, where every two results in U are considered to be similar if the distance between the two results is no greater than r. The diversification model is to find such a diverse result set $S \subseteq U$ that (i) all results in U are similar with at least one result in S and (ii) no two results in S are similar with each other.

2.2 Query Processing Techniques

Some existing techniques were proposed to efficiently process queries. Let us now present some related techniques.

2.2.1 Multiple Query Optimization

The multi-query optimization has been addressed in [65, 60, 41, 55, 84, 66], and these works mainly focused on how to share processing of the common subexpressions, which

frequently appear in complex queries running in OLAP systems. However, these approaches are designed for the optimization of multiple queries in a query batch, but not work well for online queries, since they do not support to share processing among running queries and a newly arrived query.

In the online environment, Candea et.al. [16] proposed an approach to support table scan sharing among different online queries, while Lang and Wong [49] studied the optimization problem for multiple online queries by supporting shared index scanning. In this thesis, we apply the technique of index scan sharing [49] to optimize multiple online diversity queries. However, it could be sub-optimal to shared scan a currently active index to completely evaluate a running query, if there exists another more optimal index for the query. In this thesis, we further optimize the query plan by supporting to switch the query evaluation from scanning an active index to an inactive index.

2.2.2 Adaptive Query Processing

For a query, the traditional query optimization approaches try to find an optimal query plan from a set of potential query plans, and evaluate this query based on the selected query plan. Instead of identifying a query plan, some existing works [48, 52, 10] studied the problem of run-time re-optimization. The principle behind these works is to execute queries and monitor data characteristics simultaneously, and invoke re-optimization to generate better query plans when currently running query plans become sub-optimal.

In some real applications, the data characteristics do not frequently change, but the characteristics of online queries change frequently. Initially, an optimal query plan can be identified for each running query.Some of these running queries could shared scan an index. When the system picks a new query, it could be sub-optimal to shared scan the current index to evaluate both the newly picked query and some other running queries. To optimize the query evaluation, in this thesis, we study an index switching evaluation

technique to shared scan a new index, and we also attempt to maximize the usage of the previous query evaluation on the original index.

2.2.3 Index Tuning Systems

To optimize query evaluation, some existing works [7, 22, 6, 32, 89, 8, 15, 63, 64] studied the automatic physical design tuning in DBMSs. Most of these works [7, 22, 6, 32, 89, 8] focused on the offline physical tuning for a given query workload, while others [15, 63, 64] focused on the physical tuning in the online environment. These approaches for online physical tuning can automatically determine to generate a new index or remove an existing index, based on the analysis of the current physical configuration [15]. Rather than only considering the current physical configuration, in this thesis, our proposed index tuning approach also takes into account the evaluations of running queries, and we attempt to minimize the overhead of new index generation by shared index scanning with these evaluations of running queries. Furthermore, to improve the performance of future queries, these existing approaches [15, 63, 64] generate new indexes based on the historical logs of recent queries, whose characteristics are assumed to be similar with the characteristics of queries in the near future. Instead, our approach generates new indexes by directly looking ahead at those queries in the waiting queue. Although we only focus on the online physical tuning for the partial $\texttt{D}^{+}\texttt{-tree}$ indexes in this thesis, our approaches can also generalize to the physical tunning for the normal B^{+}-tree indexes.

2.3 Diversified Spatial Keyword Search

Some recent works [54, 21, 79] studied the diversified spatial keyword search problem to find a set of relevant spatial objects which are well spread in the search region. For our proposed N-DSQ queries, spatially diversify among different groups of closely spatial

objects. The problem of identifying a highly ranked group of closely located objects is a circle-placement problem.

Let us now present some related works in spatial keyword search and the circle-placement problem.

2.3.1 Spatial Keyword Search

Recently, spatial keyword search is an active research area that aims to find spatial objects that are textually relevant to a search query. Chen et al. [25] provided a detailed survey of those techniques used for processing spatial keyword queries.

Spatial Index

To efficiently process spatial keyword queries, a number of geo-textual indexes [31, 33, 87, 68, 75, 28] have been proposed. In general, all spatial indexes can be classified into three groups: (a) textual-first index [82, 59, 68, 87], (b) spatial-first index [68, 20, 75, 87], and (c) hybrid spatial index [43, 31, 33, 72, 28]. A textual-first index usually maintains an inverted postings list for each keyword, and organizes each inverted postings list in a spatial structure, which can be an R-tree [59], a quadtree [82], a grid or a spatial filling curve. In contrast, A spatial-first index organizes all objects in a spatial structure, whose leaf nodes contain inverted files [75] or bitmaps for the text information of objects contained in the nodes. On the other hand, a hybrid spatial index tightly combines both types of information, by maintaining a text summary into every node of a spatial index [31, 33], or integrating the spatial information into each inverted list [28]. For spatial keyword queries with a small number of query keywords, it was reported [25, 82, 59] that textual-first indexes outperform others when the number of query keywords is small. `S2I` [59] and `I`^{3}`-index` [82] are two of the state-of-the-art textual-first indexes. In a `S2I` [59],

each inverted postings list is organized as a R-tree. On the other hand, in a $\texttt{I}^3\texttt{-index}$ [82], each inverted postings list is organized as a quadtree, where each node corresponds to a rectangular region and the corresponding region of each internal node is partitioned into four non-overlapped sub-regions of the same size. However, none of these existing textual-first indexes can efficiently evaluate our proposed spatial keyword queries, and we will explain it in Chapter 5.

Spatial Keyword Queries

There are two kinds of spatial queries that are mostly related to our proposed queries: (a) the collective spatial keyword search query [17, 51, 80, 81], and (b) the top-k PoI group search query [12]. The collective spatial keyword query attempts to find a group of closely located objects that collectively covers all query keywords. For each query keyword, there exists at least such an spatial object in the group that covers the keyword, and it is not necessary to contain more objects in a group to cover a query keyword. That is, the maximum number of objects in each group is no more than the number of query keywords. On the other hand, the top-k PoI group search query [12] is to find the top-k groups of closely located Points of Interest (PoIs). Instead of collectively covering all query keywords, the objects in each group are independently relevant to the query. Moreover, there is no constraint on the size of each group. That is, for a group of objects, the more objects it contains, the higher ranked it will be.

Similar to [12], we also do not constrain the number of objects in each returned group for our propose spatial query (DSQ or N-DSQ). Differently, the objects in each group collectively cover all query keywords. Additionally, we also take into account the semantic diversity of these objects in each group to improve the user satisfaction.

Spatial Diversity Search

Some recent works [54, 67, 69, 21, 79] studied the spatial diversity problem: given a query region, a set of search keywords and a limit size k, the spatial diversity query on a database of spatial objects is to find a set of k relevant objects that are well spread within the query region. The textual information of each object is used to measure the relevance of each object for a given query. However, the relevance of objects in a spatially diversified set is considered to be independent with each other. Differently, we study a novel spatial diversification problem for a set of object groups by taking into account of the spatial diversity and the semantic diversity information of each group.

2.3.2 Circle Placement Problem

For our proposed spatial query (DSQ and N-DSQ) in this thesis, the problem of identifying top-k result groups that are located within a circle of radius r is a circle placement problem: given a set of points p_i, $i = 1, 2, \cdots, n$, each of weight w_i, in the plane, and a disk of radius r, find a location to place the disk such that the total weight of the points covered by the disk is maximized. This problem is equivalent to the well known maximum weighted clique problem for the circle intersection graph. Figure 2.1 shows an example circle intersection graph for the four points (p_1, p_2, p_3 and p_4). In the graph, all circle are of the same radius r. For the intersection area of the three circles with centers as p_1, p_2 and p_3, any location within the intersection area can be the center of the disk that covers p_1, p_2 and p_3.

Existing work [35] studied the unweighted clique problem, and proposed an algorithm with a time complexity of $\mathcal{O}(n^2 \lg n)$, which can be extended to solve the weighted clique problem. For each circle, the algorithm first sorts the intersection points with other circles in the plane, and then scans these points in clockwise (or anti-clockwise) order. A count of

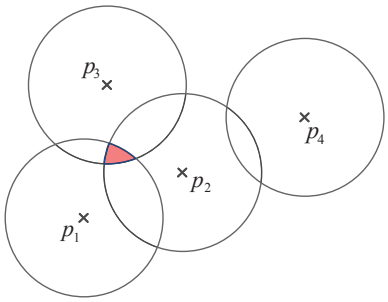


Figure 2.1: Circle Placement Problem

the number of intersecting circles is maintained during the scan. The count is incremented by 1 when we enter the intersection arc of a new circle, and is decremented by 1 when we leave the circle of concern.

To efficiently address the circle placement problem, Chazelle et.al. [23] proposed an algorithm with a time complexity of $\mathcal{O}(n^2)$. Based on the doubly connected edge list presentation of the intersection graph, the algorithm identifies the optimal location by traversing the intersection arcs in the clockwise (or anti-clockwise) order. In this thesis, we apply this algorithm to identify the top-k result groups. Differently, rather than generating the global intersection graph for all spatial objects, our approach restricts the search space based the proposed spatial index, and only generates and evaluates on some local intersection graphs for some objects in some restricted area.

CHAPTER 3

INDEXING FOR DYNAMIC DIVERSITY QUERIES

3.1 Overview

In this chapter, we study the problem of efficiently diversifying query results wrt a sequence of attributes(termed as *d-order*), where the first attribute has higher priority to diversify than the second attribute, and so on. Vee et. al. [70] were the first to study the evaluation for static diversity queries (SDQs) with a fixed d-order. To satisfy more users with different preferences, in this chapter, we attempt to efficiently evaluate dynamic diversity queries (DDQs) with user defined d-orders. A DDQ can be expressed by the following extended SQL syntax: "SELECT ... FROM R WHERE ... DIVERSIFY BY $D_1, \cdots, D_n$ LIMIT k" which retrieves a diverse set of at most k matching records from a

relation R such that the records are diversified wrt a d-order $(D_1, \cdots, D_n)$. The attributes in the SELECT clause must contain all the attributes in the DIVERSIFY BY clause.

In this chapter, we introduce a novel approach for evaluating diversity queries that is based on the concept of computing a *core cover of a query*. Based on this concept, we design a new index method, D-Index, and introduce two index variants, namely, `D-tree` and `D`$^+$`-tree`. Furthermore, we demonstrate with an experimental evaluation, which is based on a PostgreSQL implementation, that our proposed D-Index technique consistently outperforms [70] for both SDQs as well as DDQs.

For convenience, the notation table of this chapter is provided in Table 3.1, and the rest of this chapter is organized as follows. In Section 3.2, we formally define some important concepts for DDQs. Section 3.3 states the challenge of evaluating DDQs, and explains the inefficiency of evaluating DDQs by extending existing techniques designed for SDQs [70]. In Section 3.4, we give an overview of our approach. In Section 3.5, we introduce the proposed index, D-Index, followed by describing the two evaluation algorithms (`D-tree` and `D`$^+$`-tree`) on the two variants of D-Index in Section 3.6. Section 3.7 presents an extension of the proposed algorithm to improve the usage of an D-Index for evaluating more DDQs. We describe an index selection algorithm in Section 3.8. Section 3.9 presents an experimental performance evaluation of the proposed techniques. Finally, we conclude this chapter in Section 3.10.

3.2 Diverse Query Results

In this section, we present the definition of diverse query results used in this work. Our definition is based on that from [70], where a query result is diversified wrt a sequence of attributes $\delta = (D_1, \cdots, D_m)$, referred to as a *d-order*. Essentially, δ specifies a priority order for diversifying the query results with D_i having a higher priority than D_{i+1} such

R	Running example relation	Q	A diversity query
δ	The query d-order	θ	The set of selection predicate attributes
k	The limit size of query	S	A diverse result set
I	A D-Index	T	A result trie
α	The index key of a D-Index	$cover(T)$	The core cover of a diversity query
N	A node in a D-Index	V	A node in a result trie T
T_V	The subtire rooted at V	$ptup_\delta(V)$	The δ-prefix tuple in V
$ptup_\alpha(N)$	The α-prefix tuple of N	$ptup_\delta^{max}(N)$	The maximal δ-tuple of N
$rid(N)$	The RID of some tuple covered by $ptup_\alpha(N)$	$size(V)$	The number of leaf nodes in T_V

Table 3.1: Notation table of Chapter 3

that we maximize the domain values shown for D_i before D_{i+1}. The goal is to maximize the diversity of the attribute domain values shown as well as "balance" the number of records for each attribute value.

RID	B	C	SS
1	HP	1	13.3
4	HP	2	14.1
6	HP	2	15.4
15	Lenovo	2	14.1

(a) S_1

RID	B	C	SS
1	HP	1	13.3
4	HP	2	14.1
14	Lenovo	2	14.1
15	Lenovo	2	14.1

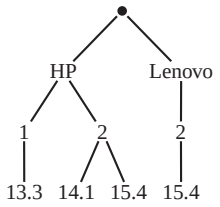
(b) S_2

RID	B	C	SS
1	HP	1	13.3
2	HP	1	14.1
8	Acer	2	15.4
15	Lenovo	2	14.1

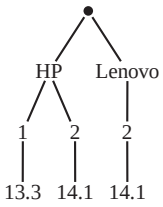
(c) S_3

RID	B	C	SS
1	HP	1	13.3
4	HP	2	14.1
8	Acer	2	15.4
15	Lenovo	2	14.1

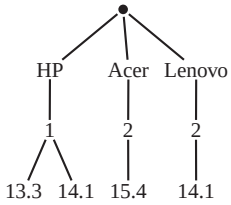
(d) S_4



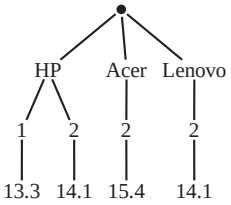
(e) T_1



(f) T_2



(g) T_3



(h) T_4

Figure 3.1: Diverse Query Results, d-order δ = (Brand, #Core, ScreenSzie)

Example 3.1: Consider a query Q on R (Table 1.1) with $k = 4$ and a selection predicate "#Core $\leq$ 2". Figs. 3.1(a)-(d) show four possible result sets (S_1 to S_4) for Q, where only the attribute values for RID, B, C, and SS are shown. If the d-order for Q is $\delta = (B, C, SS)$, we can organize each result set S_i using a trie T_i (wrt δ) as depicted in Figs. 3.1(e)-(g) which provides a more visual and convenient representation for com-

paring result diversity. Observe that T_1 and T_2 are equally diverse wrt the brand attribute (each has two distinct brand values), but S_2 is more balanced than S_1 because S_2 has two records for each brand value, whereas S_1 has three records for HP brand and one record for Lenovo brand. However, compared to T_3, both T_1 and T_2 are less diverse wrt the brand attribute. Finally, we note that T_4 is more diverse than T_3: while both are equally diverse wrt the brand attribute (each has three brand values), T_4 is more diverse wrt the #core attribute because T_4 has two distinct #core values for its two records with HP brand, whereas T_3 has only one distinct #core value for its two records with HP brand. □

In the following, we formalize the above intuition of diverse query results.

Definition 3.1 (attribute ordering)**.** *An attribute ordering of a relation R is a sequence of attributes $(A_1, \cdots, A_n)$, where each A_i is a distinct attribute of R.*

Note that an attribute ordering does not necessarily include all the attributes of R.

Consider an attribute ordering $\alpha = (A_1, \cdots, A_n)$ of R. We use α_i to denote the length-i, $i \in [0, n]$, prefix of α; i.e., $\alpha_i = (A_1, \cdots, A_i)$. We refer to each α_i as a *α-prefix*.

Definition 3.2 (α-tuple, α-prefix tuple)**.** *A tuple t is defined to be an α-tuple if $t \in \pi_\alpha(R)$ for some attribute ordering α. We say that t is an α-prefix tuple if t is an α_i-tuple for some prefix α_i of α.*

Definition 3.3 (matching α_i-tuple)**.** *An α_i-tuple t, $i \in [1, n]$ is defined to be a matching tuple for Q if all the attributes in the selection predicates (i.e., θ) occur in α_i and t satisfies all the selection predicates of Q.*

Note that it is not necessary for a matching tuple to contain all the d-order attributes or all the attributes projected by the query.

Definition 3.4 (tuple cover)**.** *Given a α-tuple t_a and a β-tuple t_b, we say that t_a covers t_b (or t_b is covered by t_a) if $\alpha \subseteq \beta$ and $t_a.A_i = t_b.A_i$ for each attribute $A_i \in \alpha$. We say that a tuple t covers a set of tuples S if t covers each $t' \in S$.*

Let $S \subseteq R$ be a result set for a diversity query Q on relation R wrt d-order δ, and T be the trie representation of S (wrt δ). Each node v in T corresponds to a unique δ-prefix tuple, which we denote by $ptup_\delta(v)$. For example, in Figure 3.1(f), if v refers to the rightmost leaf node in T_1, we have $ptup_\delta(v) = (Lenovo, 2, 15.4)$.

Given a node v in T, we use T_v to denote the subtrie rooted at v representing the subset of records $S(v) \subseteq S$; i.e., $S(v)$ is the set of records contained in T_v. For example, in Figure 3.1(f), if v refers to the node labeled "HP" in T_2, then $S(v)$ contains two records with RID values of 1 and 4.

Consider a subtrie T_v where v has c child nodes, $v_1, \cdots, v_c$. As a measure of the diversity of $S(v)$, define the metric

$$F(S(v)) = c|S(v)| - \sigma$$

where σ is the standard deviation of the set $\{|S(v_1)|, \cdots, |S(v_c)|\}$.

To understand why the above metric is meaningful for comparing result set diversity, consider a query Q to retrieve a result set of k tuples from relation R wrt d-order δ. Consider the trie representations, T_1 and T_2, of two possible result sets, $S_1, S_2 \subseteq R$, where $|S_1| = |S_2| = k$. Let $F(S_1) = c_1k - \sigma_1$ and $F(S_2) = c_2k - \sigma_2$. If S_1 is more diverse than S_2, then either (1) the root node of T_1 has more child nodes than that of T_2 (i.e., $c_1 > c_2$), or (2) the root nodes of both T_1 and T_2 have the same number of child nodes, but the child subtrees in T_1 are more balanced than those in T_2 (i.e., $c_1 = c_2$ and $\sigma_1 < \sigma_2$). Effectively, $F(S_1)$ is larger than $F(S_2)$ if S_1 is more diverse than S_2.

In other words, given a result set $S \subseteq R$ of Q, if for every node v in the trie representation of S, $F(S(v))$ can not be further increased (by replacing some records in $S(v)$ by an equal number of some other records from $R-S$ that are covered by $ptup_\delta(v)$), then the diversity of S can not be increased (without increasing the cardinality of S), and we conclude that S is a diverse result set of cardinality k. Thus, we can define a diverse result set S in terms of maximizing $F(v)$ for each node v in the trie representation of S.

Definition 3.5 (diverse result set)**.** *Let T denote the trie representation of a result set $S \subseteq R$ for a diversity query Q on R wrt d-order δ. Let T_v denote a subtrie of T rooted at v. We define S to be diverse wrt $ptup_\delta(v)$ if $F(S(v))$ is maximized over all sets $S' \subseteq R$ that are covered by $ptup_\delta(v)$ such that $|S'| = |S(v)|$. We define S to be a diverse result set for Q if S is diverse wrt every δ-prefix tuple in S.*

Example 3.2: Consider the trie T_4 in Figure 3.1(h). Let v_0 denote the root node of T_4, and v_1 denote the node in T_4 with $ptup_\delta(v_1) = (HP)$. We have $F(S_4(v_0))$ = $12 - \sqrt{2}/3$ and $F(S_4(v_1))$ = 4. S_4 is a diverse result set for Q following the definition: S_4 is diverse wrt $ptup_\delta(v_0)$ since there are only three brand values in R and v_0 has three child nodes; S_4 is diverse wrt $ptup_\delta(v_1)$ since $|S_4(v_1)| = 2$ and v_1 has two child nodes; and for each of the remaining nodes v in T_4, S_4 is diverse wrt $ptup_\delta(v)$ since $|S_4(v)| = 1$. On the other hand, T_1 in Figure 3.1(e) is not a diverse result set because S_1 is not diverse wrt $ptup_\delta(v_0)$ where v_0 is the root node of T_1: $F(S_1(v_0))$ can be further increased by making the child subtrees of v_0 more balanced by replacing RID6 with RID14 to obtain T_2 in Figure 3.1(f). □

Note that our definition of diverse result set is equivalent to one in [70] in that a set is a diverse result set under our definition if and only if it is also a diverse result set under the definition in [70]. We have chosen to present the definition in terms of the metric $F()$ as we believe that it captures more closely the intuition behind the diversity definition. We should emphasize that our contribution is not on the definition of diverse query result but on the efficient evaluation of diversity queries.

3.3 Challenges for Dynamic Queries

To motivate the need for a new approach to evaluate DDQs, we argue that although existing techniques for SDQs [70] can be extended to support DDQs, their performance is

expected to be poor due to the need to scan a significant portion of the index. This is validated by our experimental results in Section 3.9.

Let us first consider how to extend the basic technique, `OnePass` [70], to form a new variant, termed `OnePass`^{D}, for evaluating DDQs. To make the discussion concrete, suppose that the B$^+$-trees in `OnePass`^{D} have index key $\alpha = (A, B, C, D, E)$ and we are using `OnePass`^{D} to evaluate a DDQ with a d-order of $\delta = (D, E)$ and a selection predicate "$A = a_1$". Similar to `OnePass`, `OnePass`^{D} performs an index scan on the B$^+$-tree corresponding to the value a_1 in the inverted-list index I_a for attribute A. Each retrieved α-tuple from the index scan is converted to a δ-tuple to update the main-memory trie T. Due to the difference between α and δ, there are two extensions required for `OnePass`^{D} to work correctly. First, the tuples inserted into T are now in a "random" instead of a "sequential" order (e.g., the index scan returns $(a_1, b_1, c_1, d_2, e_2)$ followed by $(a_1, b_1, c_2, d_1, e_1)$, where $d_1 < d_2$). Thus, the simple scheme adopted in `OnePass` for detecting when there are sufficient tuples in a subtrie no longer works due to this random order and a more sophisticated detection scheme is required. Second, the Dewey encoding scheme used for compressing index keys does not work correctly when the α-tuples are mapped to δ-tuples (to update T) as the same attribute value could have different Dewey encodings. The second extension is trivial to fix (encode each attribute value with a unique value), but the first extension is more intricate (Section 3.4.5).

Although `OnePass`^{D} can work correctly to evaluate DDQs, its performance could be very inefficient as it might need to scan the entire index. Continuing with the example, suppose that after updating T with a newly retrieved tuple $(a_1, b_1, c_1, d_1, e_1)$, `OnePass`^{D} detects that the subtrie rooted at (d_1, e_1) has sufficient number of tuples. However, `OnePass`^{D} cannot efficiently skip to search for the next value after (d_1, e_1) as the B and C attributes preceding D are not part of the search attributes. Hence, in the worst case, no index skip operation is possible in the `OnePass`^{D} approach. For similar reasons, `Probe` could be extended to correctly evaluate DDQs but would perform even worse than `OnePass`^{D} as

the extended `Probe` would still incur random I/Os for its bi-directional scan but without the benefit of reducing useless tuple retrievals due to the absence of index skip operations.

3.4 Our Approach

In this section, we present the key ideas behind our approach of evaluating diversity queries.

3.4.1 Core Cover

Our approach for computing diverse query results is based on the concept of computing a *core cover* for a query.

Definition 3.6 (core cover)**.** *A set of δ-prefix tuples $C = \{t_1, \cdots, t_\ell\}$, $\ell \in [1, k]$, is defined to be a core cover for a diversity query Q on relation R with d-order δ and limit k if there exists ℓ positive integers $(\beta_1, \cdots, \beta_\ell)$ such that (a) $\sum_{i=1}^{\ell} \beta_i = k$ and (b) for each $t_i \in C$ and for each subset of β_i matching records $S_i \subseteq R$ that is covered by t_i, $\bigcup_{i=1}^{\ell} S_i$ is a diverse result set for Q.*

Thus, each tuple in a core cover C covers at least one tuple in a diverse result set S. We refer to $(\beta_1, \cdots, \beta_\ell)$ as the *core cover assignment for Q*. For the case where $\ell = k$, the core cover assignment for Q is trivially given by $\beta_i = 1$ for each $i \in [1, \ell]$. If $\ell < k$, then there will be duplicate δ-tuples in S and the core cover assignment essentially allocates the distribution of the duplicates among the tuples in C to ensure that S is a diverse result set.

Example 3.3: Consider a query Q on R with $\delta = (B, SS)$, a single selection predicate "#Core = 4", and a limit of 5. Consider a set of (B,C,SS)-tuples, $C = \{t_1, t_2, t_3, t_4\}$, where

$t_1 = (Acer, 4, 13.3)$, $t_2 = (Acer, 4, 14.1)$, $t_3 = (Acer, 4, 17.3)$, and $t_4 = (Lenovo, 4, 13.3)$. Then, C is a core cover for Q with a core cover assignment $(1, 1, 1, 2)$. That is, there exists a diverse result set $S \subseteq R$ for Q where each of the tuples in $\{t_1, t_2, t_3\}$ covers one tuple in S, and t_4 covers two tuples in S. Based on R in Table 1.1 and the core cover assignment $(1, 1, 1, 2)$, there are two possible diverse result sets for Q corresponding to the two sets of RIDs: $\{10, 12, 13, 17, 18\}$ and $\{11, 12, 13, 17, 18\}$. Note that although there are two tuples in R (with RIDs 10 and 11) covered by t_1, $(2, 1, 1, 1)$ is not a core cover assignment for Q as illustrated by the result sets shown above: the result set in (a) is more balanced than that in (b) wrt Brand attribute. □

RID	B	SS
10/11	Acer	13.3
12	Acer	14.1
13	Acer	17.3
17	Lenovo	13.3
18	Lenovo	13.3

(a) Diverse result set for Q

RID	B	SS
10	Acer	13.3
11	Acer	13.3
12	Acer	14.1
13	Acer	17.3
17/18	Lenovo	13.3

(b) Non-diverse result set for Q

Figure 3.2: Query Results in Example 3.3

The concept of a core cover provides a useful design framework to consider techniques for computing diverse query results. Re-examining `OnePass` [70] with this framework, we see that the core cover C computed by `OnePass`, which is organized using a trie, is characterized by the following two properties: (P1) $|C| = k$, and (P2) all the tuples in C are δ-tuples. As `OnePass` is designed for SDQs, δ is the same as a pre-determined index key α, and `OnePass` uses B$^+$-trees to retrieve δ-tuples to compute C. This is a reasonable approach when δ is the same as α. But as we explained in Section 3.3, this index design becomes unacceptable when adapted to `OnePass`^{D} for DDQs as using a B$^+$-tree index scan (with key α) to retrieve diverse δ-tuples could be extremely inefficient when α and δ are very different.

To avoid the pitfall of `OnePass`^{D}, we make the observation that since the tuples in a core

cover are δ-prefix tuples (of which δ-tuples are just a special case), a better index design is to support the retrieval of δ-prefix tuples (instead of δ tuples). Thus, instead of supporting only a single type of index scan with a single index key α, a more flexible index design is to support multiple types of index scans using α-prefixes as keys to efficiently retrieve α-prefix tuples to form δ-prefix tuples for the core cover.

The rest of this section presents our new index technique to evaluate diversity queries. Our approach consists of two data structures: a novel disk-based *diversity index or D-Index*, which supports efficient index scans with α-prefix keys (Section 3.4.2); and a run-time, main-memory structure, called the *result trie*, to organize the tuples in the core cover and guide the index traversal (Section 3.4.3). We give an overview of how these structures operate together to evaluate diversity queries in Section 3.4.4, and establish a sufficient condition for a result trie to be a core cover for a query in Section 3.4.5.

3.4.2 Diversity Index

A D-Index I on a relation R with index key $\alpha = (A_1, \cdots, A_n)$ is a height-balanced trie-like structure on the set of tuples $\pi_\alpha(R)$. The index consists of $n+1$ levels, $L_0, L_1, \cdots, L_n$, where each L_i corresponds to attribute A_i, $i \in [1, n]$. L_0 consists of a single root node, denoted by N_{root}. Each node N at L_i, $i \in [1, n]$, corresponds to a unique α_i-tuple, denoted by $ptup_\alpha(N)$. Thus, each L_i contains $|\pi_{\alpha_i}(R)|$ nodes, $i \in [1, n]$. A node N at L_i, $i \in [1, n-1]$, is the parent node of another node N' at L_{i+1} if $ptup_\alpha(N)$ is a proper prefix of $ptup_\alpha(N')$.

Each node N at L_i, $i \in [1, n]$, consists of the following information: (1) $ptup_\alpha(N)$, the α-prefix tuple corresponding to N; and (2) the RID, denoted by $rid(N)$, of some tuple in R that is covered by $ptup_\alpha(N)$. $ptup_\alpha(N)$ enables the retrieval of descendant index nodes of N while $rid(N)$ enables the retrieval of a tuple that is covered by $ptup_\alpha(N)$.

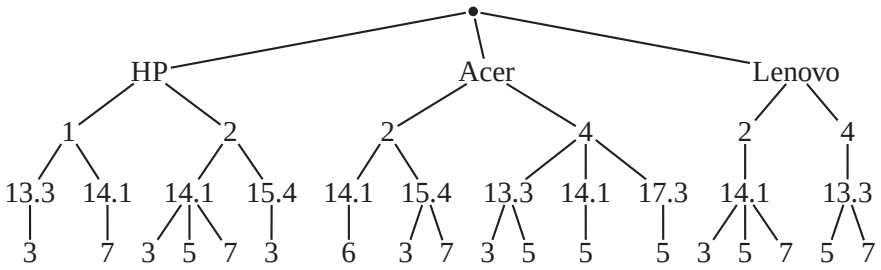


Figure 3.3: D-index on R shown in Table 1.1 with key (Brand, #Core, ScreenSize, BatteryLife)

Example 3.4: Figure 3.3 shows the D-Index with index key (Brand, #Core, ScrnSze, BatLife) on R (Table 1.1) If N denotes the left child node of the node "Acer", then $ptup_\alpha(N) = (Acer, 2)$ and $rid(N) \in \{7, 8, 9\}$. □

In addition, the root node N_{root} of I also maintains statistics on the number of distinct values for each attribute in α, denoted by $count_{N_{root}}(A_j)$; i.e., for each attribute A_j, $j \in [1, n]$, we have $count_{N_{root}}(A_j) = |\pi_{A_j}(R)|$. These statistics are used for checking certain property of the result trie (to be described in Section 3.5.3).

A D-Index I can be used to evaluate a diversity query Q if all the δ attributes α and selection predicate attributes θ occur in the index key α of I.

Definition 3.7 (matching index node)**.** *A node N in a D-Index I is defined to be a matching index node for a diversity query Q if $ptup_\alpha(N)$ is a matching tuple for Q.*

The overall idea of using an index I to evaluate Q is to retrieve δ-prefix tuples from the matching index nodes accessed to progressively compute a core cover for Q. Specifically, for each index node N accessed during the traversal of I, if N is matching index node, the α-prefix tuple corresponding to N (i.e., $ptup_\alpha(N)$) is used to update a core cover for Q. However, since the key α of I and the d-order δ of Q are generally different attribute orderings, we need to transform each α-prefix tuple t retrieved from I to its corresponding δ-prefix tuple to update a core cover for Q. We refer to this transformed tuple as the *maximal δ-prefix tuple of t*.

Definition 3.8 (maximal δ-prefix)**.** *Given two attribute orderings of R, $\alpha_i = (A_1, \cdots, A_i)$ and and $\delta = (D_1, \cdots, D_m)$, we define the maximal δ-prefix of α_i to be $(D_1, \cdots, D_j)$, $j \in [1, m]$, if (1) the set of attributes $\{D_1, \cdots, D_j\}$ occurs in α_i and (2) either $j = m$ or D_{j+1} does not occur in α_i. The maximal δ-prefix of α_i is defined to be nil if D_1 does not occur in α.*

Definition 3.9 (maximal δ-prefix tuple)**.** *Given two attribute orderings of R, $\alpha_i = (A_1, \cdots, A_i)$ and $\delta = (D_1, \cdots, D_m)$, and a α_i-tuple t, we define the maximal δ-tuple of t to be $\pi_{\delta_j}(t)$, where δ_j is the maximal δ-prefix of α_i.*

Given an index node N in I, we use $ptup_\delta^{max}(N)$ to denote the maximal δ-tuple of $ptup_\alpha(N)$.

Example 3.5: Consider $\alpha = (A, B, C, D, E)$ and $\delta = (C, A, E)$. The maximal δ-prefix of α_4 is (C, A). Given a α-tuple $t = (1, 2, 3, 4, 5)$, the maximal δ-tuple of t is $(3, 1, 5)$. Consider a query Q with $\delta = (B, SS, BL)$ and let N denote the parent node of the rightmost leaf node in the D-Index with $\alpha = (B, C, SS, BL)$ in Figure 3.3. Then $ptup_\delta^{max}(N) = (Lenovo, 13.3)$. □

3.4.3 Result Trie

To keep track of the maximal δ-prefix tuples that form a core cover for Q, we use a main-memory structure called the *result trie* (denoted by T).

The result trie T consists of at most $m + 1$ levels, $L_0, L_1, \cdots, L_m$, where each L_i corresponds to an attribute D_i, $i \in [1, m]$, in the d-order δ of Q. L_0 consists of a single root node, denoted by V_{root}. Each node V at L_i, $i \in [1, m]$, corresponds to a δ_i-tuple, denoted by $ptup_\delta(V)$. A node V at L_i, $i \in [1, m-1]$, is the parent node of another node V' at L_{i+1} in T if $ptup_\delta(V)$ is a proper prefix of $ptup_\delta(V')$.

Each node V of T consists of the following information: (1) $ptup_\delta(V)$, the δ-prefix tuple associated with V; and (2) a set of entries, denoted by $entry(V)$, where each entry $e = (\rho, rid)$ corresponds to an index node N such that $\rho = ptup_\alpha(N)$, $rid = rid(N)$, and $ptup_\delta^{max}(N) = ptup_\delta(V)$. Note that $entry(N_{root}) = \emptyset$.

Definition 3.10 (tree size)**.** *The size of a subtree T' of a result trie, denoted by $size(T')$, is defined to be the number of leaf nodes in T'.*

We use $cover(T)$ to denote the set of δ-prefix tuples corresponding to the leaf nodes of T; i.e., $cover(T) = \{ptup_\delta(V) \mid V$ is a leaf node in $T\}$.

Example 3.6: Figure 3.6(h) shows an example result trie wrt a query with $\delta = (Brand, ScrnSze, BatLife)$. We have $cover(T) = \{(Acer, 13.3, 5), (Acer, 14.1, 5), (Acer, 17.3), (Lenovo)\}$. If V denotes the rightmost child node of the node "Acer", then $ptup_\delta(V) = (Acer, 17.3)$. □

Note that our result trie differs from the trie used in [70]: our trie is not necessarily height-balanced, and it requires a more intricate maintenance procedure (Section 3.4.5) as the tuples are inserted into it in a random rather than a sequential order.

3.4.4 Overview of Query Evaluation

Our overall approach to evaluate a diversity query Q using a D-Index I and result trie T works as follows. For each matching index node N accessed in I, we update T with $ptup_\delta^{max}(N)$. Thus, the result trie is used to organize the retrieved $ptup_\delta^{max}(N)$ tuples, which is in turn used to guide the index traversal to construct a core cover for Q efficiently with a small number of index node accesses.

If the result trie satisfies a sufficient condition for $cover(T)$ to form a core cover for Q (discussed in Section 3.4.5), the index traversal terminates and $cover(T)$ is used to derive

a diverse result set for Q as follows. Let $\{V_1, \cdots, V_\ell\}$ denote the set of leaf nodes in T and $(\beta_1, \cdots, \beta_\ell)$ denote the corresponding core cover assignment for Q. Then the rid entries from $entry(V_i)$ will be used to retrieve β_i tuples to form the result set for Q. If $|entry(V_i)| < \beta_i$, then we need to retrieve additional matching tuples by using the ρ entries from $entry(V_i)$ to access additional matching index nodes. The core cover assignment is computed to ensure that the trie representation of the derived result set is as balanced as possible so that it is a diverse result set.

Cover Cover Assignment

Now we discuss how a core cover assignment $(\beta(V_1), \cdots, \beta(V_\ell))$ is computed for a query core cover $cover(T) = \{ptup_\delta(V_1), \cdots, ptup_\delta(V_\ell)\}$ that corresponds to the set of leaf nodes $LN = \{V_1, \cdots, V_\ell\}$ in a result trie T.

For each leaf node $V_i \in LN$, let $NTup(V_i)$ denote the cardinality of the set of matching tuples in R that are covered by $ptup_\delta(V_i)$. Thus, each $NTup(V_i) \geq 1$ and each $\beta_i \in [1, NTup(V_i)]$.

For each non-leaf node V in T, let $NTup(V)$ denote the sum of $NTup(V_i)$ for each leaf node V_i in T_V; and let $\beta(V)$ denote the sum of β_i for each leaf node V_i in T_V. Note that for each node V in T, $size(T_V) \leq \beta(V) \leq NTup(V)$.

Our inductive computation of $\beta(V')$ proceeds top-down as follows. We start with $V' = V_{root}$; clearly $\beta(V') = k$ if there are at least k matching tuples for Q. Let C denote the set of child nodes of V'. We now determine $\beta(V'_i)$ for each $V'_i \in C$ such that $\sum_{V'_i \in C} \beta(V'_i) = \beta(V')$. Let η be the smallest positive integer that satisfies the following inequality: $\sum_{V'_i \in C}(\max(size(T_{V'_i}),\ \min(NTup(V'_i), \eta))) \geq \beta(V')$. Each node $V'_i \in C$ can be categorized into one of three groups: (G1) $NTup(V'_i) < \eta$, (G2) $size(T_{V'_i}) < \eta \leq NTup(V'_i)$, or (G3) $\eta \leq size(T_{V'_i})$. If V'_i belongs to G1, we set $\beta(V'_i) = NTup(V'_i)$.

If V_i' belongs to G3, we set $\beta(V_i') = size(T_{V_i'})$. Finally, for each V_i' that belongs to G3, $\beta(V_i')$ is set to either η or $(\eta - 1)$ as follows.

Let $C = C_{1,2} \cup C_3$, where $C_{1,2}$ denote the set of child nodes of V' that belong to G1 or G2, and $C_3 = \{V_1', \cdots, V_r'\}$ denote the set of child nodes of V' that belong to G3. For convenience, let $size(V_1') \leq \cdots \leq size(V_r')$. Let $\beta(C_3)$ denote $\beta(V') - \sum_{V_i' \in C_{1,2}} \beta(V_i')$. Let $p = (\eta \times r) - \beta(C_3)$. Then for each $V_i' \in C_3$, we set $\beta(V_i') = \eta - 1$ if $i \in [1, p]$; otherwise, $\beta(V_i') = \eta$.

Therefore, by applying the above procedure inductively starting from V_{root}, we compute $\beta(V_i)$ for each $V_i \in LN$, and it can be shown that $(\beta(V_1), \cdots, \beta(V_\ell))$ is a core cover assignment for $cover(T)$.

3.4.5 Sufficient Condition for Core Cover

In this section, we establish a sufficient condition for $cover(T)$ to be a core cover for a query Q with limit of k.

Definition 3.11 (diverse trie)**.** *A result trie T for a query Q on relation R with d-order δ is a diverse trie if for any set of matching records $S \subseteq R$, $|S| = |cover(T)|$, that is covered by $cover(T)$, S is a diverse result set of size $|S|$.* □

Definition 3.12 (expandable node)**.** *We say that a node V in a result trie is expandable if it is possible to add a new child node to V. The new child node must correspond to a yet-to-be accessed matching index node.*

Definition 3.13 (balanced node)**.** *A node V in a result trie is defined to be balanced if for each child subtree T_i of V, the difference between $size(T_i)$ and $size(T')$ is at most one, where T' is the largest child subtree (in terms of $size()$) of V; i.e., $size(T') - size(T_i) \leq 1$.*

Definition 3.14 (balanced-diverse (b-diverse) tree)**.** *A subtree T rooted at a node V in a result trie is defined to be a balanced-diverse (or b-diverse) tree if one of the following conditions hold: (1) V is a leaf node, or (2) V is an internal node and either (a) the number of child nodes of V is equal to $size(T)$, or (b) V is balanced and not expandable, and each child subtree of V is a b-diverse tree.*

The following result states that a b-diverse result trie T is a sufficient condition for T to be a diverse result trie.

Lemma 3.1. *If a result trie T is b-diverse, then T is a diverse trie. In addition, if $|cover(T)| = k$, then $cover(T)$ is a core cover for Q.* □

Definition 3.15 (k-sufficient tree)**.** *A subtree T rooted at a node V in a result trie is defined to be a k-sufficient tree if one of the following conditions hold: either (1) V is the root node and $size(T) = k$; or (2) V is not the root node, the subtree rooted at the parent node V_p of V is k-sufficient, and the difference between $size(T)$ and $size(T')$ is at most one, where T' is the largest child subtree (in terms of $size()$) of V_p (i.e., $size(T') - size(T) \leq 1$).*

The following result states that if a subtree T' in a result trie T is a k-sufficient tree, then increasing $size(T')$ will not improve the diversity of T.

Lemma 3.2. *If T is a k-sufficient result trie for a query Q, then there exists a diverse result set S for Q such that for each k-sufficient subtree T' rooted at V in T, the number of tuples in S that are covered by $ptup_\delta(V)$ is at most $size(T')$.* □

Definition 3.16 (k-optimal tree)**.** *A tree T is k-optimal if T is both b-diverse as well as k-sufficient.*

Example 3.7: Consider a D-Index I on R with $\alpha = (B, C, SS, BL)$, and a query Q on R with $\delta = (B, SS, BL)$, a single selection predicate "#Core = 4" (i.e., $\theta = \{C\}$), and a limit of 4. Figure 3.6 shows a sequence of the states of the result trie as it is updated with

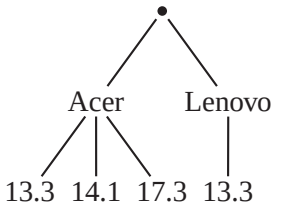
the δ-prefix tuples corresponding to a specific sequence of accessed index nodes. The node "Acer" in Figure 3.6(f) is expandable as it is possible to add a new child node "17.3" to it; however the node "Acer" is not expandable in both Figures 3.6(g) and (k). The root node in Figure 3.6(h) is not balanced since the size of its left subtree is 3 while that of its right subtres is 1; however, the root node in Figure 3.6(k) is balanced since the size of each of its child subtrees is 2. In Figure 3.6(g), the subtree rooted at the node "Acer" is 4-optimal as it is both b-diverse and 4-sufficient; however, the entire trie is 4-sufficient but not b-diverse. In Figure 3.6(i), the subtree rooted at the node "Acer" is 4-optimal, while the subtree rooted at the node "Lenovo" is b-diverse but not 4-sufficient; the entire trie is 4-sufficient but not b-diverse. Finally, in Figure 3.6(k), the entire trie is 4-optimal. □

Based on Lemmas 3.1 and 3.2, we have the following sufficient condition for a result trie to form a core cover for a query.

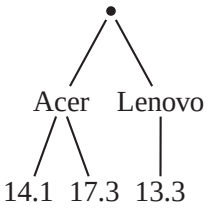
Theorem 3.1. *If for each node V in a result trie T, the subtree rooted at V is k-optimal or V is not expandable, then there exists a subtree T' of T such that $cover(T') \subseteq cover(T)$ and $cover(T')$ is a core cover for Q. In addition, if T is k-optimal, then $T' = T$.* □

Example 3.8: Consider a D-Index I on R with $\alpha = (B, C, SS, BL)$, and a query Q on R with $\delta = (B, SS)$, a single selection predicate "#Core = 4", and a limit of 4. In the result trie T shown in Figure 3.4(a), although T is 4-sufficient, T is not b-diverse and therefore also not 4-optimal. However, observe that Theorem 3.1 applies to T: each node in the subtree rooted at "Acer" is 4-optimal, and the root node as well as each node in the subtree rooted at "Lenovo" is not expandable. Therefore, there exists a subtree T' of T (shown in Figure 3.4(b)) such that $cover(T')$ is a core cover for Q. Indeed, Figure 3.4(c) shows a diverse result set for Q that is covered by $cover(T')$. □

Note that although Lemma 3.1 provides a sufficient condition for $cover(T)$ to be a core cover for Q, it is not efficient to use this condition alone to guide index navigation to



(a) T



(b) T'

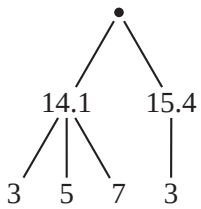
RID	B	SS
12	Acer	14.1
13	Acer	17.3
17	Lenovo	13.3
18	Lenovo	13.3

(c) S

Figure 3.4: Example for Theorem 3.1

compute the query results as it can lead to many useless index access that retrieve δ-prefix tuples that do not contribute to the final result trie. For efficiency reason, we therefore combine the balanced-diverse and k-sufficient properties in Theorem 3.1 as a stronger sufficient condition for $cover(T)$ to be a core cover for Q. The following example illustrates this requirement.

Example 3.9: Consider a D-Index I on R with $\alpha = (B, C, SS, BL)$, and a query Q on R with $\delta = (SS, BL)$, a single selection predicate "#Core = 2", and a limit of 4. In the result trie T_1 shown in Figure 3.5(a), the subtree T' rooted at "14.1" is both b-diverse and 4-sufficient (i.e., 4-optimal). Since T' is 4-sufficient, by Lemma 3.2, it is actually unnecessary to access further index nodes to expand T' since there exists a diverse result set S for Q where the number of records in S covered by (14.1) is no larger than $size(T') = 3$. Indeed, Figure 3.5(b) shows such a diverse result set for Q. If we had not used this k-sufficient property, then we could have access other unnecessary index nodes (e.g., (Acer,2,14.1,6)) that are covered by (14.1). □



(a) T

RID	SS	BL
3	14.1	3
4	14.1	5
6	15.4	3
9	15.4	7

(b) S

Figure 3.5: Example for the k-sufficient property

3.5 D-Index Variants

In this section, we present the key ideas of two instantiations of D-Index: `D-tree` is the simpler variant, which traverses the index in a DFS manner, while `D+-tree` is an improved variant to address the limitations of `D-tree`. The detailed evaluation algorithms for `D-tree` and `D+-tree` will be discussed in Section 3.6. We use I to denote a D-Index on a relation R with index key $\alpha = (A_1, \cdots, A_n)$.

3.5.1 Relevant Index Levels (RI-levels)

A D-Index I can be used to evaluate Q if all the attributes in δ and θ occur in α. Note that the ordering of the attributes in δ and θ can be different from α, and α can contain attributes that do not occur in δ or θ.

In general, not all of the index levels in I are relevant and useful for evaluating Q. We classify an index level L_i (corresponding to attribute A_i) as a *relevant index level (or RI-level)* for Q if it satisfies the following four conditions. First, A_i must be relevant for evaluating Q; i.e., A_i must be a diversity attribute in δ or a selection predicate attribute in θ. Second, α_i must contain all the selection predicate attributes in θ. This is necessary to enable checking whether $ptup_\alpha(N)$ for an accessed index node N at L_i is a matching tuple for Q. Third, if A_i corresponds to a diversity attribute D_j in δ, then α_i must contain all the attributes in δ_j. Recall that each matching tuple $ptup_\alpha(N)$ needs to be transformed to its maximal δ-prefix tuple to update the result trie. Therefore, if α_i does not contain some diversity attribute D_r, $r < j$, then it means that the maximal δ-prefix of α_i is at most δ_{r-1}, which implies that the additional values of attributes $\{D_r, D_{r+1}, \cdots, D_j\}$ retrieved from $ptup_\alpha(N)$ are not utilized at all. In this case, we are better off accessing L_{r-1} instead of L_i. Finally, if A_i corresponds to a selection predicate attribute in θ, then α_i must contain

the first diversity attribute D_1. Otherwise, the maximal δ-prefix of α_i is empty which means that the index nodes accessed from L_i are useless for updating the result trie.

Example 3.10: Consider a D-Index I with $\alpha = (A, B, C, D, E, F, G)$ and a query Q with $\delta = (E, C, G, A)$ and $\theta = \{B\}$. Q can be evaluated using I since α contains all the attributes in δ and θ. Only L_5 and L_7 (corrp. to E and G) are RI-levels. L_1 (corrp. to A) violates the second condition, L_2 (corrp. to B) violates the fourth condition, L_3 (corrp. to C) violates the third condition, and L_4 and L_6 (corrp. to D and F) violate the first condition. □

Trie implementation. As Example 3.10 illustrates, the RI-levels for a query Q are not necessarily consecutive levels in I. Given an index node N, there are two basic access patterns in D-Index: the first is to access the next node after N at the same index level, and the second is to access the first descendant node of N at some RI-level. To efficiently support these access patterns and avoid the overhead of accessing nodes at non-RI levels, we implement each D-Index as a collection of B$^+$-trees. Specifically, for each level L_i in I, the entries in L_i are indexed by a B$^+$-tree with index key α_i; thus, there is one leaf entry in the B$^+$-tree for each level-i index node N in I, and the leaf entry contains its key value $ptup_\alpha(N)$ and $rid(N)$. In this way, the B$^+$-trees corresponding to non-RI levels for Q will not be accessed for evaluating Q.

3.5.2 Definitions & Notations

Before we present the ideas behind the two index variants, we first introduce several additional definitions and notations.

Definition 3.17 (corresponding T-node of N)**.** *Given a node N in a* `D-tree` *index, we say that a node V in the result trie T is the corresponding T-node of N if $ptup_\delta(V)$ is $ptup_\delta^{max}(N)$.*

In this work, we use N to denote an index node in I and use V to denote a node in the result trie T. Given a node V in the result trie I, we use T_V to denote the subtree of the result trie T rooted at V. Given an index node N in I, we use T_N to denote the subtree of the result trie T rooted at the corresponding T-node of N.

Definition 3.18 (heavy/light leaf node)**.** *A leaf node V in T is defined to be a heavy (light) leaf node if for each ancestor node V' of V in T, the subtree rooted at V' is the largest (smallest) subtree (in terms of $size()$) among its sibling subtrees.*

Example 3.11: Let N denote the node in the D-Index in Figure 3.3 with $ptup_\alpha(N) = (Acer, 4, 14.1, 5)$. The corresponding T-node of N in Figure 3.6(g) is the node V with $ptup_\delta(V) = (Acer, 14.1, 5)$. In Figure 3.6(g), the two leftmost leaf nodes are heavy leaf nodes, while the two rightmost leaf nodes are light leaf nodes. □

3.5.3 D-tree Index

In this section, we present the key ideas of evaluating a query Q with a `D-tree` index I. The `D-tree` evaluation algorithm traverses the RI-levels of I in a top-down, depth-first manner. For each matching index node N accessed, we update the result trie with the maximal δ-tuple corresponding to N (i.e., $ptup_\delta^{max}(N)$). If the corresponding T-node of N already exists in T as V, and V is a leaf node in T, then we add an entry corresponding to N into $entry(V)$. On the other hand, if V does not exist in T, we add V into T and update $entry(V)$ as described.

If the update would cause $size(T)$ to exceed k, we first need to select a "victim" tuple from T, denoted by $ptup_\delta(V)$, where V is some leaf node in T, and decide if replacing $ptup_\delta(V)$ by $ptup_\delta^{max}(N)$ would improve the diversity of T. To maximize the diversity of T, we should pick V to be a heavy leaf node. For instance, consider the result trie

shown in Figure 3.6(g) from Example 3.7, where the two leftmost leaf nodes are heavy leaf nodes; clearly, replacing any one of these leaf nodes is better for the diversity of T than replacing any one of the non-heavy leaf nodes.

Having selected a victim tuple $ptup_\delta(V)$, we need to determine whether the replacement would improve the diversity of T. We use a simple sufficient condition to detect whether its diversity would be affected: if V' is the corresponding T-node of N after $ptup_\delta^{max}(N)$ has been inserted into T, V_a is the youngest ancestor node of V with at least two child nodes, and V_a is an ancestor of V', then the replacement does not affect the diversity of T.

Thus, if this sufficient condition holds, we do not update T with $ptup_\delta^{max}(N)$. Continuing with the example trie T_g in Figure 3.6(g), our approach would not update T_g if $ptup_\delta^{max}(N)$ is say $(Acer, 13.3, 7)$ but we would update T_g if $ptup_\delta^{max}(N)$ is $(Lenovo)$. Thus, $size(T)$ does not decrease as the index evaluation progresses and $size(T)$ is at most k.

For each accessed index node N, we proceed with the DFS-traversal from N to its next descendant node (at the next RI-level) if T_N is not k-optimal. Thus, when the index traversal terminates, Theorem 3.1 guarantees that $cover(T)$ is a core cover for Q. A diverse result set for Q is derived from $cover(T)$ as described in Section 3.4.4.

Example 3.12: Consider again Example 3.7. There are three RI-levels corresponding to attributes C, SS, and BL. Figure 3.6 shows the sequence of updates to the result trie as the `D-tree` is traversed to evaluate Q. In each of Figures 3.6(a) to (f), T is not 4-sufficient. The insertion of (Acer,17.3) in Figure 3.6(g) causes T to become 4-sufficient, but T is not b-diverse as V_{root} is still expandable. In Figure 3.6(h), the insertion of (Lenovo) replaces (Acer,13.3,3); and in Figure 3.6(i), the insertion of (Lenovo,13.3,7) replaces (Acer,13.3,5). At this point, T is 4-optimal as it is both 4-sufficient and b-diverse. □

To check if a level-i node V in T is expandable, we use the following sufficient condition:

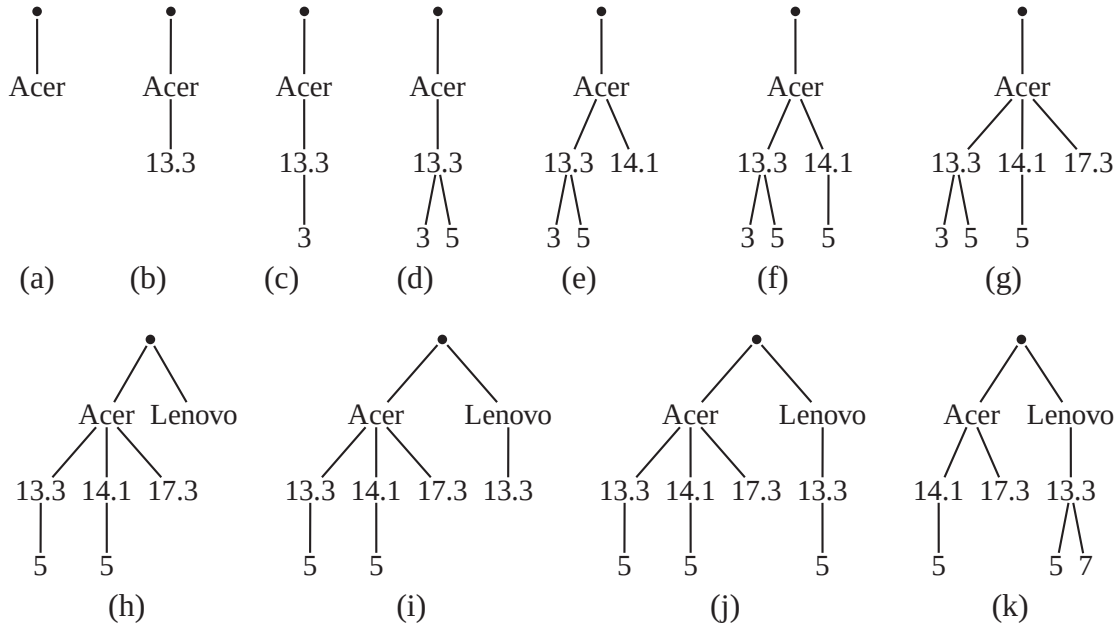


Figure 3.6: Sequence of updates to result trie by `D-tree` evaluation in Example 3.12

if the number of child nodes of V in T is less than the number of distinct values of attribute D_{i+1}, which is obtained from the statistic $count_{N_{root}}(D_{i+1})$ stored in the index's root node, then V is expandable. For the remaining properties (i.e., balanced node, diverse tree, and k-sufficient tree), they can be checked directly based on their definitions or checked more efficiently by incrementally maintaining additional information with each node (e.g., maintaining a flag to indicate whether a node is balanced).

3.5.4 D$^+$-tree Index

One drawback of `D-tree` is that the DFS-traversal of the index nodes could result in the retrieval of many matching index nodes that do not contribute to the eventual query's core cover; we refer to such index nodes as *useless index nodes*. For instance, in Example 3.12, the three index nodes retrieved to form the result subtrie rooted at $(Acer, 13.3)$ in Figure 3.6(d) turn out to be useless index nodes as the subtrie was replaced in the final result trie in Figure 3.6(k).

To reduce the number of useless index node access, we propose an improved variant of D-tree, called the D$^+$-tree, which differs from D-tree in three key ways. First, D$^+$-tree traverses the index nodes in a level-wise manner to alleviate the drawback of a DFS-traveral of the index nodes.

Second, D$^+$-tree uses additional statistics information to optimize the update of the result trie T so that for each accessed index node N, it is possible to not only add a new node V in T (i.e., V is the T-node corresponding to N) but also know about the number of child nodes of V (but not their contents) in T. We refer to such child nodes as *virtual child nodes (or child vnodes)*. This "look-ahead" capability essentially provides a cost-effective means to construct a larger and more informative result trie (with vnodes) without having to first pay the cost to access the index nodes corresponding to these vnodes. If if turns out that a vnode is subsequently replaced (i.e., its correponding index node is actually useless), we would have saved the index access cost for the replaced vnode.

Third, unlike the D-tree where it traverses from one RI-level to the next immediate RI-level, D$^+$-tree uses a cost model to determine the next "best" RI-level to access from a given index node. In this way, D$^+$-tree is able to further optimize performance by judiciously accessing a selected subset of RI-levels.

Additional statistics. To support the look-ahead capability in D$^+$-tree, we extend the statistics information that is stored only with the root node in D-tree to every node in D$^+$-tree. Specifically, for each level-i node N in a D$^+$-tree, we maintain statistics on the number of distinct values for each "descendant" attribute in the index subtree rooted at N, denoted by $count_N(A_j)$; i.e., for each attribute A_j, $j \in [i+1, n]$, we have $count_N(A_j) = |\{t.A_j \mid t \in R, ptup_\alpha(N) \text{ covers } t\}|$. Note that the statistics stored at the index root node are the same for both D-tree and D$^+$-tree.

Example 3.13: Let N denote the node labeled "Acer" in the D$^+$-tree index I in Figure 3.3. We have $count_N(C) = 2$, $count_N(SS) = 4$, and $count_N(BL) = 4$. □

Level-wise traversal. In D$^+$-tree, the top-down traversal of selected RI-levels of the index is carried out in two phases. In the first phase, D$^+$-tree selects a starting RI-level (say level ℓ) to traverse (based on a cost model) and scans for matching level-ℓ index nodes. For each accessed index node N, the result trie is updated with $ptup_{\delta}^{max}(N)$ similar to what is done in D-tree. Let V denote the corresponding T-node of N after the update of T. If T_V is not k-optimal, the evaluation algorithm will determine the maximum number of child nodes of V, denoted by MC, for $cover(T)$ to be a core cover for Q, and insert an appropriate number of child vnodes for V so that the total number of its child nodes in T is MC. Note that vnodes must be leaf nodes in T.

At the completion of the first phase, the result trie T constructed is height-balanced up to level j, where δ_j is the maximal δ-prefix of α_ℓ, with possibly some vnodes at level $j+1$. If T contains vnodes or it is not k-optimal, we begin the second phase of scanning other RI-levels of I which operates by performing a top-down, breadth-first traversal of the result trie starting with level j. Suppose that the algorithm is currently scanning level-i of the result trie, $i \in [j, m)$, and D_i occurs as attribute A_r in α. For each level-i result trie node V accessed, if T_V is not k-optimal or V has child vnodes, then we will start an index scan wrt an index node N. The goal is to retrieve a sufficient number of descendant index nodes of N from I so that their maximal δ-prefix tuples will be inserted into T_V to make V k-optimal (if T_V is not k-optimal), or replace the child vnodes of V (if T_V has child vnodes). To determine N, we pick any one entry (ρ, rid) from $entry(V)$, and let N be the node such that $ptup_\alpha(N) = \rho$. Given N, we use a cost model to select the next "best" RI-level (say ℓ') to access. As before, we update the result trie for each matching level-ℓ' index node N' accessed and if V' is the corresponding T-node of N' and $T_{V'}$ is not k-optimal, we insert an appropriate number of child vnodes for V'.

Since T might have leaf nodes that are vnodes, each update of T should replace a vnode whenever possible. For example, consider the result trie in Figure 3.7(d) where the two leaf nodes of node $(Lenovo, 13.3)$ are vnodes (indicated by $\circ$ nodes). When T is

updated with $(Lenovo, 13.3, 5)$ in Figure 3.7(e), the update replaces one of the vnodes of $(Lenovo, 13.3)$.

At the completion of the second phase, T does not contain any vnodes, if T is k-optimal, Theorem 3.1 guarantees that $cover(T)$ is a core cover for Q, and the diverse result set is constructed following the same procedure described in Section 3.4.4. Otherwise, Theorem 3.1 guarantees that there exists a core cover $cover(T')$, $cover(T') \subseteq cover(T)$.

Now we discuss how to find a core cover $cover(T')$, $cover(T') \subseteq cover(T)$, from a result trie T, where for each node V, T_V is k-sufficient or V is not expandable.

For each node V in T, let $NTup(V)$ denote the cardinality of the set of matching tuples in R that are covered by $ptup_\delta(V_i)$, and let $\beta(V)$ denote the number of matching tuples in a diverse result set that are covered by $ptup_\delta(V_i)$.

Our inductive refinement from T to T' proceeds top-down as follows. We start with $V' = V_{root}$; clearly $\beta(V') = k$ if there are at least k matching tuples for Q. If $\beta(V') = NTup(V')$,or $\beta(V') = size(T_{V'})$ and $T_{V'}$ is b-diverse, we do not need to remove any subtrees in $T_{V'}$. On the other hand, we need to check and refine the subtree $T_{V'}$. Let C denote the set of child nodes of V'. If $|C| \geq \beta(V')$, we can only keep $\beta(V')$ child nodes and remove other nodes in the subtree $T_{V'}$. Otherwise, we now determine $\beta(V'_i)$ for each $V'_i \in C$ such that $\sum_{V'_i \in C} \beta(V'_i) = \beta(V')$. Let η be the smallest positive integer that satisfies the following inequality: $\sum_{V'_i \in C}(\min(NTup(V'_i), \eta)) \geq \beta(V')$. Each node $V'_i \in C$ can be categorized into one of two groups: (G1) $NTup(V'_i) < \eta$, or (G2) $\eta \leq NTup(V'_i)$. If V'_i belongs to G1, we do not need to remove any nodes in $T_{V'_i}$. If V'_i belongs to G2, $\beta(V'_i)$ is set to either η or $(\eta - 1)$ as follows. Let $C = C_1 \cup C_2$, where C_1 denote the set of child nodes of V' that belong to G1, and $C_2 = \{V'_1, \cdots, V'_r\}$ denote the set of child nodes of V' that belong to G2. For convenience, let $size(V'_1) \leq \cdots \leq size(V'_r)$. Let $\beta(C_2)$ denote $\beta(V') - \sum_{V'_i \in C_1} \beta(V'_i)$. Let $p = (\eta \times r) - \beta(C_2)$. Then for each $V'_i \in C_2$, we set $\beta(V'_i) = \eta - 1$ if $i \in [1, p]$; otherwise, $\beta(V'_i) = \eta$.

After removing some nodes in T, we get such a result trie T' that $cover(T')$ is a core cover for Q.

Example 3.14: Consider again Example 3.7 but using D$^+$-tree as the D-Index. Figure 3.7 shows the sequence of updates to the result trie as the D$^+$-tree is traversed to evaluate Q, where the vnodes are indicated by $\circ$ nodes. The first RI-level that D$^+$-tree chooses to access is the level corresponding to attribute SS; i.e., the RI-level corresponding to attribute B is skipped. Thus, for the evaluation of Q, only two (i.e., corresponding to SS and BL) out of the three RI-levels are accessed. Figure 3.7(h) shows the result trie at the completion of scanning index nodes at the level for SS. Observe that the D$^+$-tree evaluation incurs only one useless index node access (i.e., $(Acer, 13.3)$)) compared to three useless index node access using D-tree in Example 3.12. □

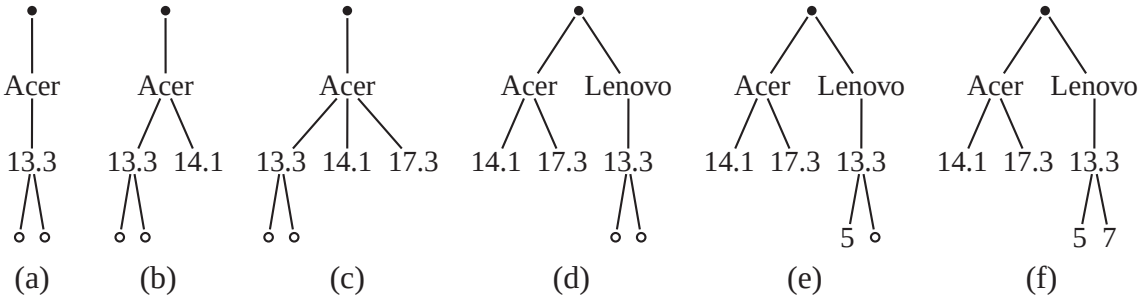


Figure 3.7: Sequence of updates to result trie by $\boldsymbol{D^+}$-tree index evaluation in Example 3.14

Besides using the additional statistics to determine the number of child nodes of a result trie node, the additional statistics can also be used to more accurately determine whether a trie node is expandable. Instead of using the approximate statistics in the root node for this purpose (as in D-tree), we perform the following for D$^+$-tree: whenever we update the result trie with $ptup_{\delta}^{max}(N)$ to create a new level-j leaf node V in T, we copy the statistic $count_N(D_{j+1})$ from N to V for this purpose, which is more accurate than $count_{N_{root}}(D_{j+1})$.

Cost model for RI-level selection. We now outline how D$^+$-tree uses a cost model to

select the next "best" RI-level to access wrt a level-ℓ index node N. This RI-level selection problem arises in three cases: (C1) N is the index root node (i.e., selection of the starting RI-level); and (C2) N is the index node corresponding to some trie node V accessed during the breath-first traversal of T, where (a) T_V is not k-optimal or (b) V has some child vnodes. For (C2b), since T_V is already k-optimal, we can simply select the next RI-level below the level of N. For (C1) and (C2a), the procedure is more elaborate as the goal is to pick an RI-level to minimize the overall index access cost to retrieve a target number of index nodes (denoted by num). For (C1), num is equal to the query limit k, while for (C2a), num is equal to maximum possible size of T_V for $cover(T)$ to be a core cover for Q. The maximum subtree size is computed by the `MaxSubtreeSize` function in Section 3.6.2. For simplify, we assume that the tree generated by accessing each RI-level is balance, and then we present a simplified version of our cost model which is to find the smallest RI-level i such that $i > \ell$ and the estimated number of level-i matching index nodes is at least num.

Now we discuss about the estimate function of the size of tree T by accessing a RI-level. Consider query Q with τ predicates and d-order $\delta = (D_1, \cdots, D_m)$, and a RI-level L_ℓ. Let δ_i, $i \in [1, m]$, be the longest proper prefix of δ that each attribute D_j, $j \in [1, i]$, occurs in α_ℓ.

For simplify, we first assume that $\tau = 0$. Since the order of δ_i does not affect the size of tree by accessing L_ℓ, we reorder δ_i as $\delta'_i = (D'_1, \cdots, D'_i)$ in the same order with α_ℓ. We use $\phi(D'_j, D'_{j+1})$, $j \in [1, i)$, to denote the average distinct number of D'_{j+1} at the subtree rooted at each node on the $\mathtt{D^+\text{-}tree}$ level wrt D'_j. After that, we can estimate the size of T as follow.

$$M = count_{N_{root}}(D'_1) \cdot \prod_{j=1}^{i-1} \phi(D'_j, D'_{j+1}) \tag{3.1}$$

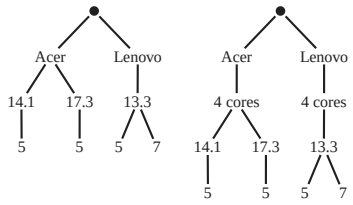
Now we consider the scenario that $\tau > 0$. We reorder the τ selection predicate attributes

in the same order with α_ℓ, and denote them as $(SPA_1, \cdots, SPA_\tau)$. For simplify of the description, we assume that all of these predicates are equality predicates. Let τ' ($\tau' \in [0, \tau]$) be the maximal integer that $(SPA_1, \cdots, SPA_{\tau'})$ is a proper prefix of α_ℓ. We have a naive case that $\tau' = \tau$. Let N be the particular node in `D+-tree` identified by the τ' predicates. In such case, we can easily estimate it in Equation 3.2. The only difference with Equation 3.1 is that we use a more precise statistic information stored in node N.

$$M = count_N(D'_1) \cdot \prod_{j=1}^{i-1} \phi(D'_j, D'_{j+1}) \tag{3.2}$$

At last, we consider about the most complex case that $0 < \tau$ and $\tau' < \tau$. It is too hard to directly estimate the size of T wrt δ'. Let $\gamma = (A_1, \cdots, A_m)$ be the order among the attribute set of each SPA'_o, $o \in (\tau', \tau]$, and D'_j, $j \in [1, i]$, in the same order with α_ℓ. Instead of directly estimate the size of T wrt δ', we consider about the tree T' wrt γ. But we should note that for a selection predicate attribute A_p, $p \in [1, m]$, all nodes at level L_p of T' labeled by the same value. Then we have that the size of T' is the same with that of T. Let's take an example to illustrate it.

Example 3.15: Reconsider again Example 3.12. Fig. 3.8(a) shows the tree T for Q, while Fig. 3.8(b) shows the tree T' which contains the C level. Even though the structures of the two tree are different, they have the same size. □



(a) (b)

Figure 3.8: Comparison of two tree size

Now we discuss about the estimation of the size of T'. If we do not consider about these predicate constraints on T', we can easily estimate the size by using Equation 3.1 and

Equation 3.2. Based on these constraints, we can cut some branches in T'. For a level of T' wrt a selection predicate attribute A_p, $p \in [1, m]$, there could be nodes labeled by $count_N(A_p)$ different A_p values, but we only keep subtrees rooted at nodes labeled by the predicate value. Now we can estimate the size of T' as follow.

$$N(L_i) = \frac{count_N(A'_1) \cdot \prod_{j=2}^{q} \phi(A'_{j-1}, A'_j)}{\prod_{j=\tau'+1}^{\tau} count_N(SPA_j)} \tag{3.3}$$

3.5.5 Implementation Issues

Insufficient RID problem. Note that it is possible that the D-Index might not have sufficient RIDs to answer a query even though there are adequate number of records in the relation R being indexed. This is due to the design of D-Index which stores only a single RID in each index node. To address this problem, one way is to change the design of the last index level (i.e., L_n) so that each level-n index node N now stores the RIDs of all the records in R that are covered by $ptup_\alpha(N)$ instead of just a single RID. With this design, we can retrieve more RIDs associated with a leaf node V in T by first accessing some entry (ρ, rid) from $entry(V)$ and use the α-prefix tuple ρ to retrieve appropriate level-n descendant nodes in I to obtain their RID-lists.

Index key compression. To optimize the performance of the constituent B$^+$-trees of a D-Index, we compress each index's key values by using a mapping table to map the original attribute values of the keys into compressed forms.

3.6 Evaluation Algorithms

In this section, we present the detailed evaluation algorithms for both `D-tree` and `D`$^+$`-tree`.

Algorithm 3.1: `D-tree-Eval` (Q, I)

Input: query Q with $\delta = (D_1, \cdots, D_m)$ and limit k, index I with $\alpha = (A_1, \cdots, A_n)$
Output: diverse result set S for Q

```
1 initialize result trie T with V_root;
2 DFSIndexScan (Q, I, N_root, T);
3 GetTuples(T, k);
4 initialize S to be empty;
5 foreach leaf node V in T do
6 |   foreach (ρ, rid) in entry(V) do
7 |   |   add rid into S;
8 return S;
```

Algorithm 3.2: `DFSIndexScan` (Q, I, N, T)

Input: query Q, index I with n levels, index node N, result trie T
Output: updates result trie T

```
1 ℓ ← NextRILevel (Q, I, N);
2 foreach matching level-ℓ descendant node N' of N in I do
3 |   UpdateTrie (Q, I, N', T);
4 |   if (T_N' is not k-optimal) and (ℓ ≠ LastRILevel(Q, I)) then
5 |   |   DFSIndexScan (Q, I, N', T);
6 |   if T_N is k-optimal then
7 |   |   break;
```

3.6.1 D-tree Index

The main algorithm for evaluating a query Q with a `D-tree` index I is shown in Algorithm 3.1. After initializing the result trie T, the function `DFSIndexScan` (Algorithm 3.2) is invoked to traverse the RI-levels of I in depth-first order starting at the root node N_{root}.

In `DFSIndexScan`, the function `NextRILevel`(Q, I, N) returns the next RI-level of I (wrt Q) that is the closest level below the level of an index node N, while the function `LastRILevel`(Q, I) returns the last RI-level of I (wrt Q). For each accessed index node N, the result trie will be updated using the function `UpdateTrie` (Algorithm 3.3) if N is a matching node for Q. The DFS-traversal at N is terminated if the corresponding T-node of N is k-optimal or N is located at the bottommost RI-level.

Algorithm 3.3: `UpdateTrie` (Q, I, N, T)

Input: query Q with limit k, `D-tree` index I, index node N, result trie T
Output: updates result trie T

1 let V be the node in T such that $ptup_\delta(V)$ is the longest prefix of $ptup_\delta^{max}(N)$;
2 **if** ($ptup_\delta(V)$ *is a proper prefix of* $ptup_\delta^{max}(N)$) **and** ((V *is a leaf node*) **or** ($size(T) < k$)) **then**
3 insert $ptup_\delta^{max}(N)$ into T to form a new leaf node V';
4 $entry(V') \leftarrow \{(ptup_\alpha(N), rid(N))\}$;
5 **else**
6 **if** ($|entry(V)| + size(T) < k$) **then**
7 add $(ptup_\alpha(N), rid(N))$ to $entry(V)$;
8 **else**
9 **if** $ptup_\delta(V)$ *is a proper prefix of* $ptup_\delta^{max}(N)$ **then**
10 let V_h be a heavy leaf node in T, and V_a be the youngest ancestor node of V_h with at least 2 child nodes;
11 **if** V_a *is not an ancestor node of* V **then**
12 delete $ptup_\delta(h)$ from T;
13 insert $ptup_\delta^{max}(N)$ into T to form a new leaf node V';
14 $entry(V') \leftarrow \{(ptup_\alpha(N), rid(N))\}$;

The function `GetTuples` (step 3) takes the query's core cover computed by `DFSIndexScan` to derive a diverse result set for Q by retrieving an appropriate number of additional tuples into $entry(V)$ for each leaf node V in T (Section 3.4.4). Finally, steps 4 to 7 collect the RIDs from the leaf nodes of T to return a diverse result set for Q. The details of function `GetTuples` are omitted due to lack of space.

3.6.2 D$^+$-tree Index

The main algorithm for evaluating a query Q with a `D+-tree` index I is shown in Algorithm 3.4. The function `ScanIndexLevel` (Algorithm 3.5) is invoked wrt an index node N and to access matching index nodes at a single RI-level (below the level of N). This RI-level, denoted by ℓ, is selected using the the function `NextBestRILevel` (step 2). The function `NextBestRILevel`(Q, I, N) returns the next "best" RI-level to traverse, based on some cost model.

In `ScanIndexLevel`, the function `IndexLevel`(I, C) returns the level in I that cor-

responds to a given attribute C; i.e., $\texttt{IndexLevel}(C) = i$ if C occurs as A_i in the index key α, $i \in [1, n]$.

In `RefineTrie`, we first determine the maximum number of child nodes of V, denoted by MC, for $cover(T)$ to be a core cover for Q. The number of vnodes to be inserted for V is then given by $MC - c_t$, where c_t is the existing number of child nodes of V. MC is the minimum of $count_N(A_\ell)$, which represents the maximum number of child nodes of V in T, and $maxsize$, which is the maximum value of $size(T_V)$ for $cover(T)$ to be a core cover for Q.

The function $\texttt{MaxSubtreeSize}(T, V)$ (Algorithm 3.8) computes $maxsize$. This is computed inductively by computing $\texttt{MaxSubtreeSize}\ (T, V')$ for each ancestor node V' of V in T. Let V_p denote the parent node of V, and let k_p denote the computed maximum possible value for $size(V_p)$. Then $M = size(T_V) + k_p - size(V_p)$ denote the size of T_V after inserting $k_p - size(V_p)$ entries into T_V to enlarge T_{V_p} to its maximum possible size. So long as there is some sibling node V' of V with $size(T_{V'}) > size(T_V)$, we can further enlarge T_V with new insertions into T_V that replace entries in $T_{V'}$. For sibling nodes of V whose subtrees are shrunk in this way, they all must be reduced to the same size which we denote by λ, and the enlarged size of T_V must be either λ or $\lambda + 1$. Given this, let S denote the set of sibling nodes of V in T, and E denote $\lambda + \sum_{V' \in S} \min(size(T_{V'}), \lambda)$. Then $\texttt{MaxSubtreeSize}(T, V')$ is computed by finding the largest integer value for λ such that $\lambda \geq M$ and $k_p - 1 \leq E \leq k_p$. If $E = k_p - 1$, then $\texttt{MaxSubtreeSize}(T, V) = \lambda + 1$; otherwise, $\texttt{MaxSubtreeSize}(T, V) = \lambda$.

The function `InsertChildVNodes` (Algorithm 3.7), which is a slight variation of `UpdateTrie`, updates the result trie T by inserting up to num number of virtual child nodes to a trie node V; for each vnode V' added, $entry(V')$ is initialized to an empty set.

Algorithm 3.4: `D+-tree-Eval` (Q, I)

Input: Query Q with $\delta = (D_1, \cdots, D_m)$ and limit k, index I with $\alpha = (A_1, \cdots, A_n)$
Output: diverse result set S for Q

```
1  initialize result trie T with V_root;
2  ℓ ← ScanIndexLevel(Q, I, N_root, T);
3  let δ_j be the maximal δ-prefix of α_ℓ;
4  for i = j to m − 1 do
5      if T is k-optimal and has no vnodes then
6          break;
7      foreach level-i node V in T that is not k-optimal or has some child vnode do
8          pick an entry e = (ρ, rid) from entry(V);
9          let N be the index node in I with ptup_α(N) = ρ;
10         ScanIndexLevel(Q, I, N, T);
11         if V is an internal node in T then
12             remove e from entry(V);
13         if T is k-optimal and has no vnodes then
14             break;
15 construct S following steps 3-7 in Algorithm 3.1;
16 return S;
```

3.7 Extended Evaluation Method

Consider a diversity query Q, we have described two methods to evaluate Q on I, if all the attributes in δ and θ occur in α. However, it is not necessary to ensure that all attributes in δ occur in α. In this section, we discuss an extended evaluation method to evaluate Q on I even if some attributes on δ do not occur in α.

The following result states that a k-optimal result trie T for query Q is still k-optimal for another query Q', whose d-order is an extension from the d-order of Q.

Lemma 3.3. *Consider two diversity queries Q and Q' with the same selection predicates and the same k, δ_Q is a proper prefix of $\delta_{Q'}$. Let $cover(T)$ be a core cover for Q. If T is k-optimal, then $cover(T)$ is also a core cover for Q'.* □

Let's take an example to illustrate it.

Example 3.16: Consider again Example 3.7, and another diversity query Q' with $\delta =$

Algorithm 3.5: `ScanIndexLevel` (Q, I, N, T)

Input: Query Q with $\delta = (D_1, \cdots, D_m)$ and limit k, index I with $\alpha = (A_1, \cdots, A_n)$, N is an index node, T is result trie
Output: updates result trie T, returns scanned RI-level

```
1  let V be the corresponding T-node of N;
2  ℓ ← NextBestRILevel (Q, I, N);
3  if T_N is not k-optimal then
4      foreach matching level-ℓ descendant node N' of N do
5          UpdateTrie⁺(Q, I, N', T);
6          if (T_N' is not k-optimal) and (ℓ ≠ LastRILevel(Q, I)) then
7              RefineTrie (Q, I, N', T) ;
8          if T_N is k-optimal then
9              break;
10 else
11     if V has some child vnode then
12         i ← IndexLevel(I, D_|ptup_δ(V)|+1);
13         c_v ← number of child vnodes of V;
14         if c_v ≤ count_N(A_i) then
15             foreach level-ℓ descendant node N' of N do
16                 UpdateTrie⁺(Q, I, N', T);
17                 if V has no child vnodes then
18                     break;
19 return ℓ;
```

Algorithm 3.6: `RefineTrie` (Q, I, N, T)

Input: Query Q with limit k, `D-tree` index I, N is a level-i index node, result trie T
Output: updates result trie T

```
1  let V be the corresponding T-node of N;
2  let c_t be the number of child nodes of V in T;
3  let ℓ = IndexLevel(I, D_|ptup_δ(V)|+1);
4  if count_N(A_ℓ) > max(1, c_t) then
5      maxsize ← MaxSubtreeSize(V, T);
6      MC ← min(count_N(A_ℓ), maxsize);
7      if MC = maxsize then
8          entry(V) ← ∅;
9      InsertChildVNodes(Q, I, V, MC − c_t, T);
10 insert (ptup_α(N), rid(N)) into entry(V);
```

Algorithm 3.7: `InsertChildVNodes` (Q, V, num, T)

Input: Query Q with limit k, V is a node in result trie T, num is the number of child vnodes to add to V
Output: updates result trie T

for $i = 1$ **to** num **do**
  if $size(T) < k$ **then**
    add a child vnode V' to V in T;
    $entry(V') \leftarrow \emptyset$;
  else
    let V_h be a heavy leaf node in T;
    let V_{anc} be the youngest ancestor node of V and V_h;
    if *V_{anc} is not the parent node of V_h* **then**
      delete $ptup_\delta(h)$ from T;
      add a child vnode V' to V in T;
      $entry(V') \leftarrow \emptyset$;

Algorithm 3.8: `MaxSubtreeSize` (V, T, k)

Input: node V in result trie T, k is the limit of query
Output: maximum possible value of $size(T_V)$

if *V is V_{root}* **then**
  $x \leftarrow k$;
else
  let V' be the parent node of V in T;
  $y \leftarrow size(T_V) +$ `MaxSubtreeSize` $(V', T, k) - size(T_{V'})$;
  $x \leftarrow y$;
  let $V_1, \cdots, V_s$ be the sibling nodes of V in T such that $size(T_{V_1}) \geq \cdots \geq size(T_{V_s})$;
  for $i \leftarrow 1$ **to** s **do**
    if $x > size(T_{V_i})$ **then**
      break;
    else
      $x \leftarrow \lceil \frac{y + \Sigma_{j=1}^{i} |size(T_{V_i})|}{i+1} \rceil$;
return x;

(B, SS, BL, LC) and the same selection predicate and limit size as Q. In Figure 3.6(k), the entire trie T is 4-optimal for both of Q and Q'. Therefore, $cover(T)$ is a core cover for both queries. □

As can be seen in Example 3.16, the two queries share the same core cover, and thus the D-Index I with $\alpha = (B, C, SS, BL)$ can be used to execute Q', even though LC does not occur in α.

Consider index I and query Q. Let δ' be the longest proper prefix of δ that each attribute occurs in α, and let Q' be the query with d-order δ'. Based on the cost model, we estimate the result trie T for Q', and we can get the cover core of Q by evaluating Q' on I if T is k-optimal.

In summary, instead of the strict constraint of the two evaluation methods, the extended evaluation method ensures that an index I can be used to evaluate more diversity queries.

3.8 Index Selection

In this section, we consider the index selection problem of recommending a set of `D+-tree` indexes to optimize the performance of a given query workload W with respect to a space constraint. We discuss two variants of the problem. The first variant, *full* `D+-tree` *selection*, treats each collection of B$^+$-trees for a `D+-tree` as an atomic unit while the second variant, *partial* `D+-tree` *selection*, treats each B$^+$-tree in a `D+-tree` as an atomic unit.

3.8.1 Full D$^+$-tree Selection

The optimization for the *full* `D+-tree` *selection* problem is NP-complete, since it can be reduce from Set-Cover problem. Now we present a heuristic approach for the *full*

D$^+$-tree *selection* problem. Consider a query Q_i with τ predicates and d-order $\delta_{Q_i} = (D_1, ..., D_m)$. For each permutation of selection predicate attributes ordering $(SPA_1, .., SPA_\tau)$, we generate a set of candidate index I_j, $j \in [1, m]$, with index key $\alpha_j = (SPA_1, .., SPA_\tau, D_1, ..., D_j)$. Therefore, for query Q_i, we can generate at most $m \cdot \tau!$ different candidate indexes.

After generating a set of candidate indexes for each query in the workload, we can generate more candidate indexes by merging two candidate indexes. Let C be the set of candidate indexes for queries in the workload. For a candidate index $I \in C$, we use α'_I to denote the ordering for the selection predicate attributes in α_I, and α''_I to denote the ordering for the remaining attributes in α_I. Let $S(\alpha)$ be the set of attributes in α_I.

Consider two candidate indexes I_1 and I_2 in C. We generate two more candidate indexes by merging them together, if $|S(\alpha_{I_1}) \cup S(\alpha_{I_2})| \leq |S(\alpha_{I_1}) \cap S(\alpha_{I_2})| + 1$. The intuition behind is to minimize the evaluation time on the merged index for all queries which can be evaluated on either I_1 or I_2. Let β be the attribute ordering $(\alpha'_{I_1}, \alpha'_{I_2}, \alpha''_{I_1}, \alpha''_{I_2})$. We generate an attributes ordering α by removing all duplicate attributes in β. More precisely, consider an attribute A_i in β, we can remove it if A_i appears in the proper prefix β_{i-1}. Subsequently, we generate a new candidate index with the index key α. On the other hand, let β' be the attribute ordering $(\alpha'_{I_2}, \alpha'_{I_1}, \alpha''_{I_2}, \alpha''_{I_1})$. We can generate another candidate index with the index key α' which is obtained by removing all duplicate attributes in β'.

We repeatedly merge two candidate indexes in C until that C reaches a fixed point. We now discuss the heuristic for selecting the indexes. Now we use the greedy algorithm used in GreedyCube [44] to select a set of full D$^+$-tree indexes under the space constraint. Let IS be the current set of D$^+$-tree indexes. We use $Cost(IS, W)$ to denote the cost of queries in W executed by using the indexes in IS, and use $B(I, IS)$ to denote the benefit of index I relative to IS. Then we have $B(I, IS) = Cost(IS, W) - Cost(\{I\} \cup IS, W)$.

At the beginning, we initialize IS as an empty set, and repeatedly select the index with the maximal benefit until that there is no more space.

3.8.2 Partial D$^+$-tree Selection

Similarly, the optimization for *partial* `D+-tree` *selection* problem is also NP-complete. Now we discuss the heuristic approach for partial `D+-tree` selection. Like above, we use the same strategy to generate a set of partial `D+-tree` indexes, and also use the greedy algorithm to select a set of partial `D+-tree` indexes. However, we should note that the cost of a query executing on a partial `D+-tree` index is different from that of executing on a full `D+-tree` index, since we can only access on the single level of a partial `D+-tree` index.

After obtaining a set of partial `D+-tree` indexes, we need to organize these selected indexes as a DAG, since we need to probe access to optimize the result trie if it is not enough to just accessing the current partial `D+-tree` index. In order to probe access between these indexes, we consider each index pair (I_1, I_2) ($S(\alpha_{I_1}) \subset S(\alpha_{I_2})$) which are directly connected in DAG. If the first attribute A_1 of α_{I_2} is not in the attribute set $S(\alpha_{I_1})$, we modify I_1 by appending A_1 in the tail of α_{I_1}; otherwise, we do not need to modify I_1.

3.9 Performance Study

We conducted an experimental study to evaluate the effectiveness of our proposed techniques. Sections 3.9.1 and 3.9.2 compare the performance of SDQs and DDQs, respectively, using synthetic datasets. Section 3.9.4 reports the comparison using real datasets.

Our results show that `D+-tree` has the best performance. For synthetic datasets, `D+-tree` is on average $2\times$ and up to $4.4\times$ faster than `OnePass` for SDQs, and on average $5\times$ and up to $35\times$ faster than `OnePass`^{D} for DDQs. For real datasets, `D+-tree` is on average $1.8\times$ and up to $2.7\times$ faster than `OnePass` for SDQs, and on average $2.2\times$ and up to $3.5\times$ faster than `OnePass`^{D} for DDQs.

Data sets. We generated four synthetic tables, $R_1, \cdots, R_4$, by computing the join of the *lineitem*, *part*, *customer*, and *orders* relations from the TPC-H benchmark using four different scale factors (SF). The properties of these tables are as follows: Each R_i consists

Relation	SF	Size (GB)	No. of tuples (million)
R_1	0.75	1.03	4
R_2	4.4	4.83	18.73
R_3	16	9.9	38.35
R_4	36	15	56.35

Table 3.2: Information on Synthetic Tables

of 10 attributes; for convenience, we use $A, \cdots, J$, respectively, to denote the attributes *linenumber*, *discount*, *tax*, *returnflag*, *container*, *shipinstruct*, *shipmode*, *linestatus*, *nationkey*, and *orderstatus*.

The synthetic datasets are evaluated using the following 5 SDQs (Q_1 to Q_5) and 5 DDQs (Q_6 to Q_{10}): All the SDQs share the same d-order $\delta = (A, B, C, D, E, F, G, H, I, J)$.

Query	θ
Q_1	A
Q_2	C
Q_3	F
Q_4	C,F
Q_5	A,C,F

(a) SDQ

Query	θ	Diversity Ordering, δ
Q_6	A	A,F,B,C,D,E,J,G,H,I
Q_7	A	B,C,D
Q_8	A	B,D,C
Q_9	A	C,D,B
Q_{10}	A	D,B,C

(b) DDQ

Figure 3.9: Diversity Query

Recall that θ represents a query's set of selection predicate attributes (SPA). To be fair to `OnePass` [70], we used only equality selection predicates for all queries.

Algorithms. We compared our proposed `D-tree` and `D`$^+$`-tree` against `OnePass` [70] and `OnePass`^{D}. Recall from Section 3.3 that `OnePass`^{D} is an extended variant of `OnePass` to evaluate DDQs; we incorporated D-Index's result trie structure into `OnePass`^{D} to support the random order of trie updates. Since `Probe` performed similarly to `OnePass` for SDQs [70] and is expected to be worse than `OnePass`^{D} for DDQs

(Section 3.3), we omit the comparison against `Probe` and its extension. We also evaluated the performance of two sequential scan techniques: *TableScan* scans the relation while *DIndexScan* scans the last RI-level of a D-Index. However, as these two techniques performed significantly worse than `D+-tree` (`D+-tree` is about $50\times$ and $100\times$ faster than *DIndexScan* and *TableScan*, respectively), we omit these two techniques in this work.

All the algorithms were implemented in PostgreSQL 9.0.2: we extended PostgreSQL's GIN index to support the skip operations for `OnePass` [70] and `OnePass`^{D}, and both `D-tree` and `D+-tree` were implemented as a collection of B$^+$-trees (Section 3.5.1).

For each table R_i, we built a `D-tree` and `D+-tree` with index key $\alpha = (A, \cdots, J)$, and built the B$^+$-trees of `OnePass` and `OnePass`^{D} with α as the index key. Our implementation shows that `D+-tree` index is about 4 times smaller than the GIN index used in `OnePass` and `OnePass`^{D}: As an example, for the 15GB table, the size of the `D+-tree` is only 1.9GB while the size of the GIN index is 8.5GB.

Parameters. We varied the following four experimental parameters: (1) the size of dataset with the default size of 10GB using R_3, (2) the query limit k with a default value of 10, (3) the number of selection predicate attributes (SPA) with a default value of 1, and (4) the position of a SPA with a default value of 1.

For comparing DDQs, we also varied two additional parameters: (1) length of query d-order (i.e., $|\delta|$), and (2) the ordering of the attributes for a given set of diversity attributes.

The experiments were conducted on a PC with a Qual-Core Intel Xeon 2.66Ghz processor, 8GB of memory, one 500G SATA disk and another 750GB SATA disk, running Ubuntu 10.04.4. Both the operating system and PostgreSQL were built on the 500GB disk, while the database was stored on the 750GB disk.

In our experiments, each execution time reported refers to the total running time for a query. Each running time is measured with the query running alone in the database sys-

tem, and the database system is restarted between queries. Each query is run 5 times, and the reported running time is the average of 3 values excluding the minimum and maximum values.

3.9.1 Static Diversity Queries

Effect of data size

Figure 3.10 compares the performance for different data sizes on Q_1. The results show that `D+-tree` gives the best performance and it outperforms `OnePass` by an increasing factor of 1.7, 2.4, 2.7, and 3.0 as the data size increases. Observe that while `D+-tree` performs similarly for the different data sizes, `OnePass`'s performance worsens with increasing data size. The results demonstrate that `D+-tree`'s level-wise index traversal is more effective and scalable than the depth-first traversal of `D-tree`. The results also show that `D-tree` generally outperforms `OnePass`: the reason is that while it is possible for `D-tree` is to terminate its DFS-traversal at any index level, `OnePass` can only terminate its scan at the leaf level.

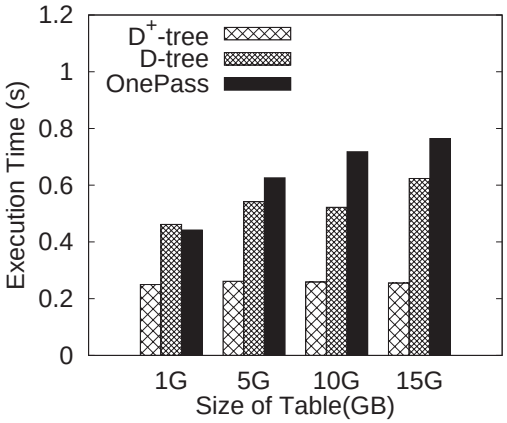


Figure 3.10: Effect of data size on Q_1

Effect of query limit, k

Figure 3.11 compares the performance for different values of the query limit k on Q_1. Here again, the results show that D$^+$-tree gives the best performance which outperforms OnePass by up to a factor of 3. The number of index entries accessed by OnePass increases from 211 to 17723 as k increases from 10 to 150, while that for D$^+$-tree only increases from 11 to 297. Note that the performance fluctuations for D-tree is due to the fact that as k increases, although the number of accessed pages increases, the I/O access pattern also becomes more sequential.

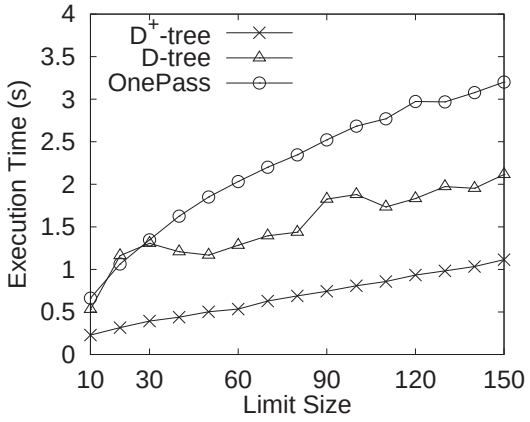


Figure 3.11: Effect of limit size k on Q_1

Effect of number of SPA

Figure 3.12 compares the performance as the number of selection predicate attributes is varied. We used queries Q_3, Q_4 and Q_5, which have 1, 2, and 3, SPAs, respectively, and query selectivity factors (denoted by sel) of 20%, 2%, and 0.5%, respectively.

The results show that D$^+$-tree gives the best performance and it outperforms OnePass by an increasing factor of 1.7, 4.1, and 4.4, as sel decreases. For both D-tree and D$^+$-tree, their performance improves (as expected) when sel decreases. However, OnePass actually performs worse when sel drops from 20% to 2%, and then improves

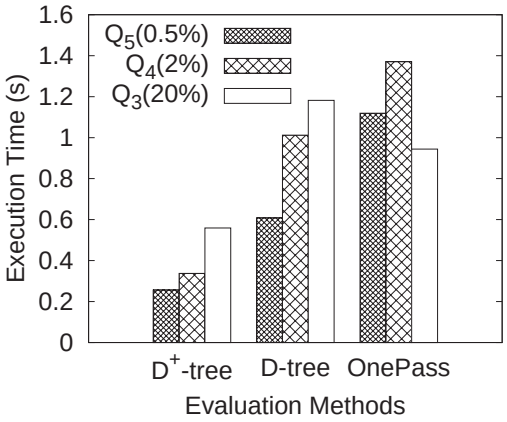


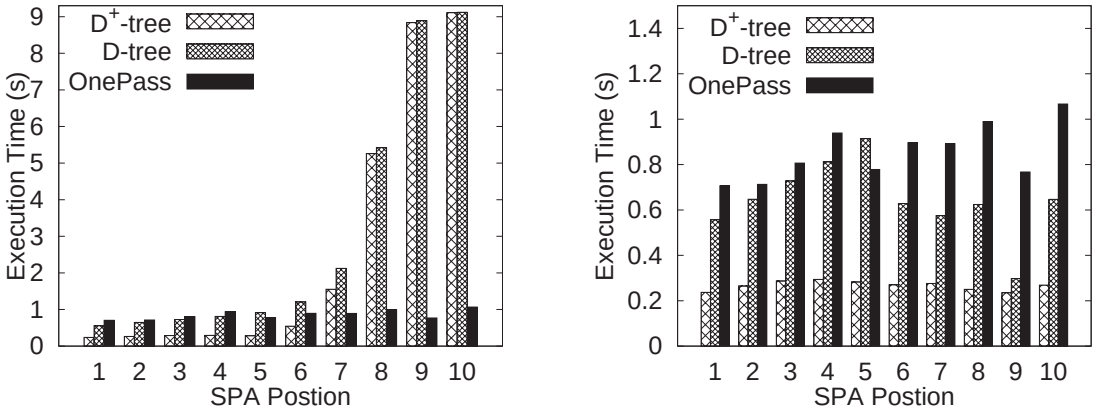
Figure 3.12: Effect of the number of SPA

when sel drops further to 0.5%. There are two factors affecting the performance of `OnePass` when there are multiple SPAs: one is the increase in number and cost of index scans with more SPAs, and the other is the more effective index skips with more SPAs. Thus, `OnePass` performs worse for Q_4 compared to Q_3 as the first factor dominates the second factor; however, it performs better for Q_5 compared to Q_4 as the second factor dominates the first factor.

Effect of SPA position

Figure 3.13(a) compares the performance of 10 SDQs with the same d-order of δ and a single SPA whose position varies from 1 to 10. The results show that `OnePass` performs similarly for all of the 10 queries as it is insensitive to the SPA position. In contrast, while both `D-tree` and `D+-tree` perform similarly for the first six queries (i.e., with SPA position between 1 and 6) their performance deteriorate significantly for the last three queries (i.e., when the SPA position is at least 8). The reason is that the size of the first RI-levels for the last three queries are very large.

However, this performance issue with using a single D-Index to evaluate a set of workload queries can be addressed by selecting a set of indexes (wrt to some space constraint) to evaluate the workload. Indeed, we have developed an efficient heuristic for this index



(a) Compare with a single D-Index (b) Compare with a set of D-Index

Figure 3.13: Effect of the SPA Position for SDQs

selection problem, and for this workload of ten queries, it turns out that building an additional D-Index with index key $\alpha' = (A, F, B, H, J, G, I, C, D, E)$ is sufficient to address the performance issue. The total size of the two D-Indexes is only 36% of the size of the single index used by `OnePass`. Figure 3.13(b) shows the performance comparison with both `D-tree` and `D+-tree` using this two-index configuration (i.e., each query is evaluated using the more efficient index between the two). The results show that `D+-tree` is consistently the most efficient method. Note that since all the static queries have the same d-order δ, the index key used in the single `OnePass` index (which is equal to δ) is already the optimal index key for evaluating each of the static queries. Therefore, unlike the D-Index, the performance of `OnePass` will remain the same even if additional indexes are created for the `OnePass` approach.

3.9.2 Dynamic Diversity Queries

Effect of query limit, k

Figure 3.14 compares the performance for different values of the query limit k on Q_6. The results show that `D+-tree` outperforms `OnePass`^{D} by up to a factor of 35. Comparing

Figure 3.14 for DDQs with Figure 3.11 for SDQs, we observe that the performance of both D$^+$-tree and D-tree do not vary too much, but the performance of OnePass^D for DDQs is worse than that of OnePass for SDQs. This demonstrates that it is not effective to extend OnePass, which was designed for SDQs, to handle DDQs. For example, when $k = 10$, OnePass^D scans a total of 1761346 index entries of which only 61 of them are used to update the result trie. This result concurs with our explanation of OnePass^D's expected poor performance in Section 3.3.

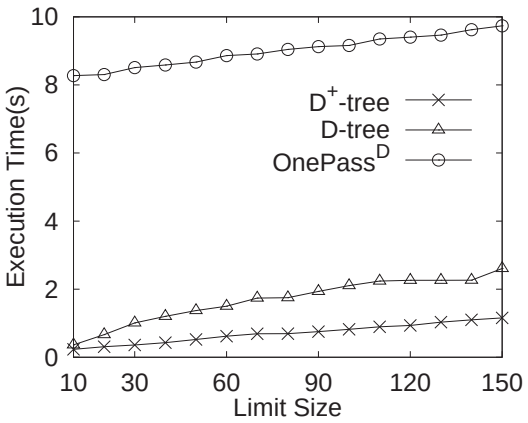


Figure 3.14: Effect of limit size k on Q_6

Effect of length of query d-order, $|\delta|$

In this experiment, we examine the effect of varying the length of the query d-order. We generated 8 DDQs, $Q_1^3, \cdots, Q_1^{10}$, from Q_1, where each of these queries is the same as Q_1 except that the d-order of Q_1^i is the length-i prefix of that of Q_1; thus, Q_1^{10} is the same as Q_1.

The results in Figure 3.15 show that D$^+$-tree consistently outperforms OnePass^D by up to a factor of 2.2. Observe that the performance of D$^+$-tree is very similar for all the queries; indeed, D$^+$-tree selects the same initial RI-level of 3 for all the queries. The performance of OnePass^D is also not too sensitive to $|\delta|$ as it does not seriously affect the number of index pages accessed. For D-tree, its performance becomes worse for

the last four queries due to an increase in the number of index node access: the number of index pages accessed by `D-tree` for the 8 queries are 3, 4, 8, 10, 20, 27, 37, and 47, respectively.

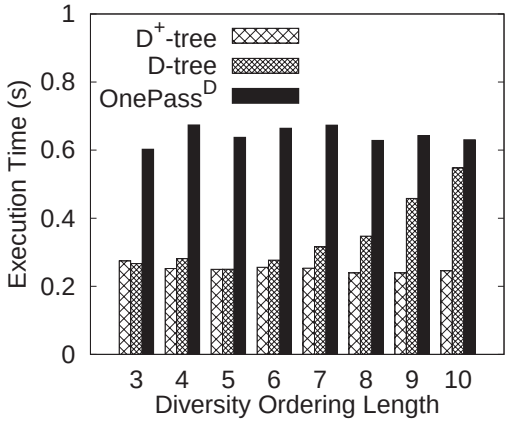


Figure 3.15: Effect of the length of query d-order

Effect of ordering of diversity attributes.

In this experiment, we examine the effect of different orderings of a same set of diversity attributes. Figure 3.16 compares the performance for the queries Q_7, Q_8, Q_9, and Q_{10} which all share the same set of diversity attributes $\{B, C, D\}$. In the following discussion, we use δ_{Qi} to denote the d-order for Q_i.

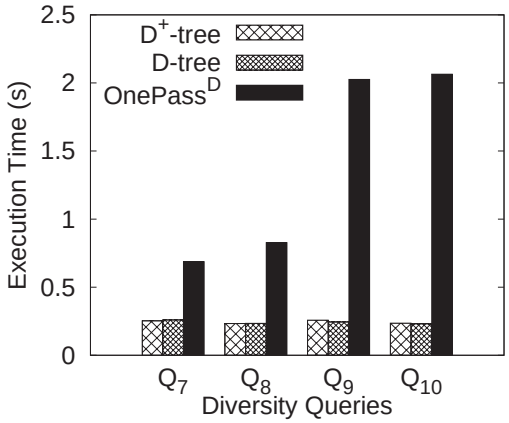


Figure 3.16: Effect of the attribute ordering

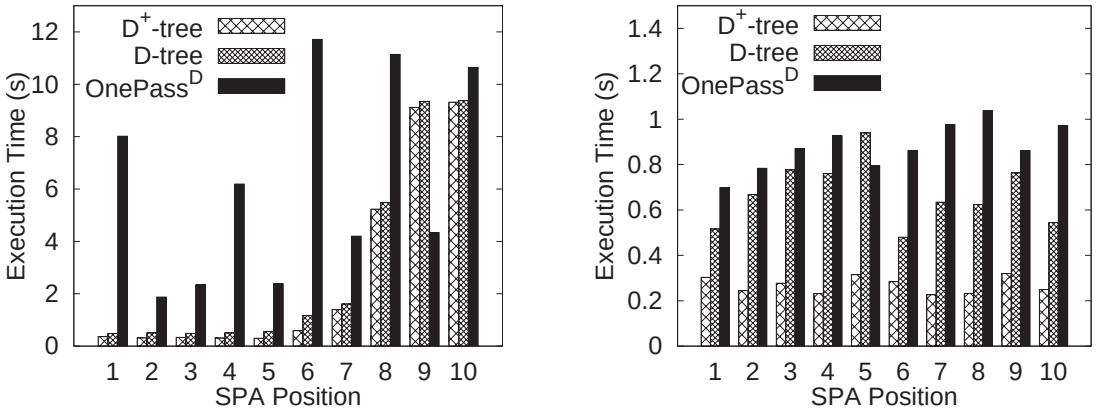
The results show that the performance of both `D-tree` and `D+-tree` are not sensitive to the attribute ordering. This is because the number of RI-levels for these four queries

are small: they are 3, 2, 2 and 1 levels, respectively. More importantly, the sizes of these RI-levels are also small. In contrast, the performance of $\texttt{OnePass}^D$ varies rather widely. $\texttt{OnePass}^D$ performs the best for Q_7 with $\delta_{Q7} = (B, C, D)$ because together with the selection attribute A, (A, B, C, D) forms a proper prefix of the index ordering α which enables $\texttt{OnePass}^D$ to perform efficiently. For Q_8 with $\delta_{Q8} = (B, D, C)$, the performance of $\texttt{OnePass}^D$ is slightly worse relative to that for Q_7 because δ_{Q8} with selection attribute A now forms a shorter proper prefix (A, B) of α and its evaluation now requires more skip operations compared to that for Q_7. However, for queries Q_9 and Q_{10}, the performance of $\texttt{OnePass}^D$ becomes significantly worse because both δ_{Q9} as well as δ_{Q10} are ordered drastically differently from α which is not conducive at all for the performance of $\texttt{OnePass}^D$ as explained in Section 3.3. Thus, $\texttt{OnePass}^D$ performs equally poorly for the last two queries.

Effect of SPA position

Figure 3.17(a) compares the performance of 10 DDQs with the same d-order as that of Q_6 and a single SPA whose position varies from 1 to 10. Comparing the performance for DDQs in Figure 3.17(a) with that for SDQs in Figure 3.13(a), we have two key observations. First, the performance behaviour of D-Index (i.e., `D-tree` and $\texttt{D}^+\texttt{-tree}$) is similar for both SDQs and DDQs; and $\texttt{OnePass}^D$ outperforms D-Index when the SPA position is 9. Second, while `OnePass` performs efficiently for for all the SDQs in Figure 3.13(a), $\texttt{OnePass}^D$ performs poorly for DDQs in Figure 3.17(a). Note that the performance of D-Index depends very much on the size of the starting RI-levels while that of $\texttt{OnePass}^D$ depends on the size of the selected inverted lists. Thus, if the size of the starting RI-levels is much larger than that of the inverted lists, $\texttt{OnePass}^D$ could outperform D-Index.

However, similar to our discussion for SDQs in Figure 3.13(b), the performance for evaluating a set of queries could be improved by using more than one index. In Figure 3.17(b),



(a) Compare with a single D-Index (b) Compare with a set of D-Index

Figure 3.17: Effect of the SPA Position for DDQs

we compare the performance of the methods using a set of two indexes. For `OnePass`^{D}, the optimal index has key (A,F,B,C,D,E,J,G,H,I), while for both `D-tree` and `D`$^+$`-tree`, the optimal set of two indexes have keys (A,F,B,C,D,E,J,G,H,I) and (A,J,F,G,H,B,I,C,D,E). Comparing the results in Figure 3.17(a) and Figure 3.17(b), it is clear that the performance of each of the methods improve with an additional index, and `D`$^+$`-tree` significantly outperforms `OnePass`^{D} in Figure 3.17b). Note that the total size of the two D-Indexes is only 26% of the size of the single index used by `OnePass`^{D}.

3.9.3 Performance on Index Sets

After investigating the performance on one single D-tree Index, now we compare the performance on a full `D`$^+$`-tree` index set and a partial `D`$^+$`-tree` index set for a given diversity query workload.

We first need to generate a query workload. Since not all attributes are equally important for users, we group the ten attributes into three small clusters: $\{A, B, C, D\}$, $\{E, F, G, H\}$ and $\{I, J\}$. We randomly generate a workload of 30 diversity queries, each one contains 1-3 predicate and 5-8 diversity attributes. The limit size of each query is in the range [10, 100]. For each query, when generating a selection predicate attribute or a

diversity attribute, the probability of choosing an attribute from the first cluster is 2 times of that from the second cluster, and 4 times of that for the third cluster.

After obtaining the query workload, we generate a set of full D$^+$-tree indexes and a set of partial D$^+$-tree indexes under the same space constraint (20% of the size of original table), based on the heuristics in Section 3.8. Let I be the single full D$^+$-tree index, IS$_f$ be the set of full D$^+$-tree indexes, and IS$_p$ be the set of partial D$^+$-tree indexes.

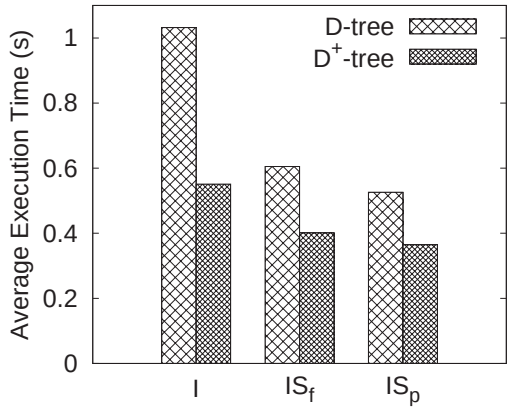


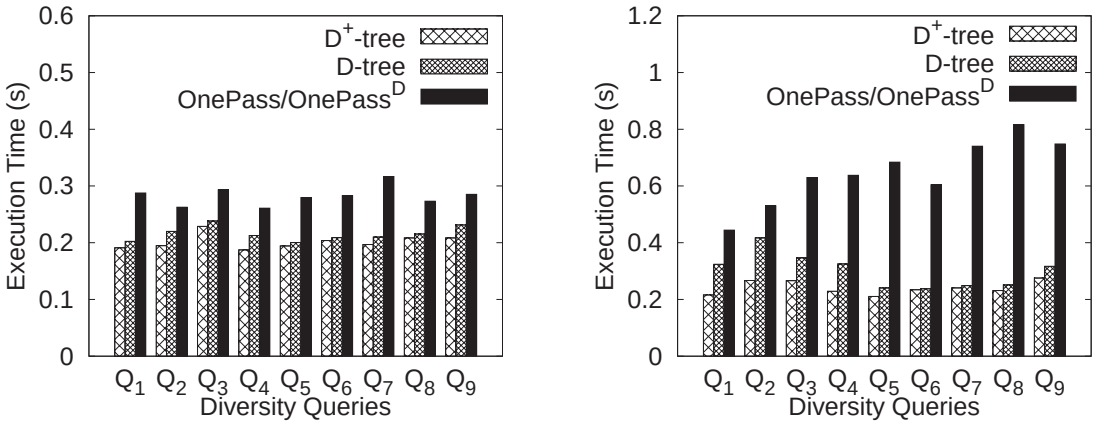
Figure 3.18: Comparison on different index sets

We run each query of the workload on I, IS$_f$ and IS$_p$, respectively. Fig. 3.18 shows the average execution time of D-tree and D$^+$-tree for queries in workload on I, IS$_f$ and IS$_p$. As can be seen in Fig. 3.18, both of D-tree and D$^+$-tree give the best performance on IS$_p$. D$^+$-tree on IS$_p$ outperforms D$^+$-tree on I by a factor of 1.4, and outperforms D$^+$-tree on IS$_f$ by a factor of 1.1.

3.9.4 Comparison on Real Data Sets

In this section, we present performance results using a real dataset on laptop products extracted from eBay. The original dataset (denoted by Laptop$_1$) is a relation with 11 attributes containing 39,411 laptop records (24MB). We created a larger dataset (denoted by Laptop$_2$) from Laptop$_1$ by duplicating it 100 times. For each of these two datasets,

we created four indexes, OnePass, OnePass^D, D-tree, and D$^+$-tree, all with the same index key (B, T, C, M, D, S, P, O), where B, T, C, M, D, S, P, and O denote attributes *brand*, *type*, *condition*, *memory*, *disk*, *screen size*, *processor type* and *operating system*, respectively. We used the following nine diversity queries for this experiment: queries Q_1 to Q_4 are SDQs, while queries Q_5 to Q_9 are DDQs.



(a) Laptop$_1$ (24MB) (b) Laptop$_2$ (2.3GB)

Figure 3.19: Comparison with laptop data sets from eBay

The performance results in Figures 3.19(a)-(b) shows that the performance gain of D$^+$-tree over OnePass and OnePass^D increase with the data size. For the Laptop$_1$ dataset, Figure 3.19(a) shows that D$^+$-tree outperforms OnePass by up to a factor of 1.5 for SDQs and outperforms OnePass^D by up to a factor of 1.6 for DDQs. For the Laptop$_2$ dataset, Figure 3.19(b) shows that D$^+$-tree outperforms OnePass by up to a factor of 2.7 for SDQs and outperforms OnePass^D by up to a factor of 3.5 for DDQs.

3.10 Summary

In this chapter, we have examined the problem of computing diverse query results. We have proposed a novel indexing technique, D-Index, that is based on the concept of computing a core cover, for evaluating both static as well as dynamic diversity queries. We also

Q	Selection Predicates	Diversity Ordering, δ	k
Q_1	B = 'HP'	B, T, C, M, D, S, P, O	10
Q_2	B = 'HP'	B, T, C, M, D, S, P, O	20
Q_3	C = 'New'	B, T, C, M, D, S, P, O	10
Q_4	B = 'HP' and C = 'New'	B, T, C, M, D, S, P, O	10
Q_5	B = 'HP'	T, M, C, S, D, P	10
Q_6	B = 'HP'	T, M, C, S	10
Q_7	B = 'HP'	M, D, S, C, T, P	10
Q_8	B = 'HP' and C = 'New'	T, M, S, D, P	10
Q_9	B = 'HP'	T, M, C, S, D, P	20

Table 3.3: The query workload for real laptop data sets from ebay

have designed two instantiations of the D-Index, `D-tree` and `D`$^+$`-tree`. Our comprehensive performance study comparing against the state-of-the-art technique for static diversity queries, `OnePass`, and its extended variant for dynamic diversity queries, showed that `D`$^+$`-tree` outperforms existing techniques on average by a factor of 2.

CHAPTER 4

EVALUATION OF MULTIPLE DIVERSITY QUERIES

4.1 Overview

In this chapter, we study the optimization problem of evaluating multiple online diversity queries. For each diversity query, we apply the most efficient evaluation algorithm $\texttt{D}^{+}\texttt{-tree}$ proposed in Chapter 3 based on a given set of partial $\texttt{D}^{+}\texttt{-tree}$ indexes. Rather than independently evaluating each individual query, in this chapter, we concurrently evaluate multiple queries by applying the techniques of the shared index scan [49], the switched index evaluation and the online index-tuning.

In this chapter, we propose a new framework, where all online diversity queries are modeled as a sequence in order of their arrival time. All of these queries are maintained in a

waiting queue. Instead of simply applying the first-come-first-serve strategy, each waiting query is allowed to be reordered to improve the opportunity of index scan sharing. The system can also adaptively change an index scan for an existing running query to use a different index scan that could be shared scan with the query evaluation for a new query. Furthermore, the running system can automatically self-tune the set of partial `D+-tree` indexes to improve the evaluations of future queries. Consequently, we demonstrate with an experimental evaluation, which is based on a PostgreSQL implementation, that our proposed techniques consistently outperform the independent concurrent evaluations of multiple queries.

For convenience, the notation table of this chapter is provided in Table 4.1, and the rest of this chapter is organized as follows. In Section 4.2, we describe the proposed framework. Section 4.3 presents the concurrent evaluations for multiple online diversity queries by sharing the index scan among a set of partial `D+-tree` indexes. In Section 4.4, we introduce the self-adaptive component of automatically update the set of indexes. Section 4.5 presents the implementation of our optimization system. Section 4.6 presents an experimental performance evaluation of the proposed techniques. Finally, we conclude this chapter in Section 4.7.

R	Running example relation
Q	A diversity query
δ	The query d-order
k	The limit size
I, I'	A partial `D+-tree` index
IS	The set of partial `D+-tree` indexes
e, e', e_i, e'_i	An entry in an index
S_r	The set of running diversity queries
S_w	The set of waiting diversity queries in the queue
N	The maximum number of concurrent evaluated queries

Table 4.1: Notation table of Chapter 4

4.2 Framework

In this section, we describe our framework for the optimization of multiple online diversity queries. Figure 4.1 shows the diagram of the framework. Different users can issue and submit their own diversity queries into the system. As the system could receive a large number of queries at the same time, it is impossible to evaluate all of them immediately. Therefore, a waiting queue is designed to maintain these queries. Several queries will be picked from the waiting queue for processing, when the running system is available. Let us assume that the running system can concurrently evaluate at most N queries. The index selection algorithm mentioned in Section 3.9.3 is applied to generate a set of partial $\mathtt{D}^{+}\mathtt{-tree}$ indexes to evaluate diversity queries. Additionally, the system automatically self-tunes the index set to improve the performance of query evaluations.

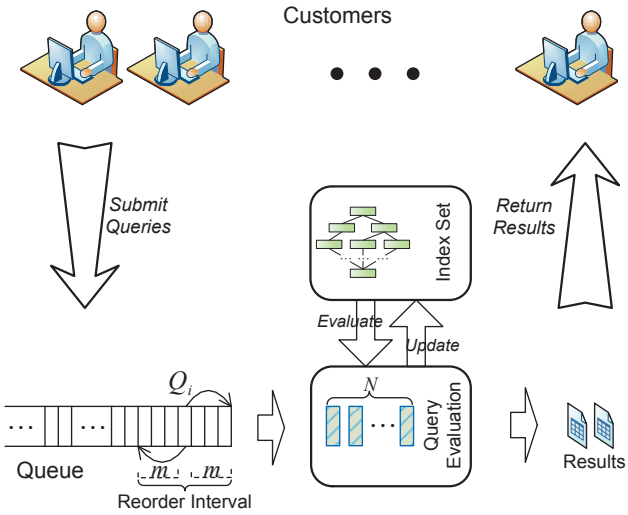


Figure 4.1: The framework for multiple online diversity queries

When the running system is available to evaluate one more query, the straight-forward way is to directly pick the first query in the waiting queue. However, the picked query might not share index scan with other running queries. To improve the opportunity of shared index scan and improve performance, each waiting query is allowed to be reordered in the queue. However, for a waiting query which is not able to share index scan with other queries, it could be infinitely delayed. To avoid such kind of infinite delay phenomenon,

we can constrain that the execution of each query in the queue can be delayed at most m times. In other words, the reordering allows at most m queries that are waiting behind the first query Q to be executed before Q is executed. More specifically, for a query Q_i in the queue, let $\psi(Q_i)$ be the unique logical timestamp of Q_i the waiting queue, where $\psi(Q_i) \in 1, 2, \cdots$. Then Q_i can be reordered within the interval $[\psi(Q_i) - m, \psi(Q_i) + m]$.

After picking a waiting query from the queue, the running system then concurrently evaluates both the newly picked query as well as other running queries. In this thesis, we apply the technique [49] to support the shared index scan among these queries. For a set of diversity queries, we use a plan-bipartite-graph to represent the set of potential query plans, and select the optimal query plan from these potential query plans. For a newly picked query Q, the system can shared scan the current `D+-tree` index to evaluate both Q and other running queries if the current index can be used to evaluate Q. This query plan, however, could be suboptimal if there exists another switchable `D+-tree` index for these queries. Therefore, our system can dynamically adapt the query plans by switching these query evaluations to scan another `D+-tree` index.

Under a space constraint, a fixed set of partial `D+-tree` indexes can be generated for an offline diversity query workload, based on the algorithm mentioned in Section 3.8.2. In the online environment, the fixed set of partial `D+-tree` indexes, however, could not be universally optimal for the online query workload at different periods of time, since the characteristics of queries at different time could be much different. To efficiently evaluate multiple online queries, our system can automatically adjust the set of partial `D+-tree` indexes, by looking ahead at those queries in the waiting queue. Furthermore, to minimize the overhead of the automatic index tuning, the index generation can shared index scan with the evaluations of running queries.

4.3 Multiple Diversity Query Evaluation

In this section, we introduce the evaluation of multiple online diversity queries, given a set of partial $\texttt{D}^+\texttt{-tree}$ index set IS. As mentioned in Section 4.2, each query in the waiting queue is allowed to be reordered to improve the opportunity of index scan sharing with other running queries. Let S_r be the set of currently running diversity queries, and S_w be the set of queries waiting in the queue. If $|S_r| < N$, it indicates that the system is available to evaluate more queries, and then several queries in S_w would be picked to be evaluated.

Let Q^{min} ($Q^{min} \in S_w$) denote the query in the queue with the minimum timestamp, and S_c represent the candidate set of waiting queries that are allowed to be picked by the running system at this time. Thus, we have $S_c = \{Q \in S_w \mid \psi(Q) \geq \psi(Q^{min}) + m\}$. In this thesis, the system picks a candidate query Q', $Q' \in S_c$, to minimize the average remaining time that is needed to complete the evaluation of the queries in the set $S_r \cup \{Q'\}$. Intuitively, the system can efficiently evaluate this newly picked query, and the execution of newly picked query will not delay or slow down the evaluations of other running queries in S_r.

After discussing the metric used for the selection of the next waiting query, let us now introduce the concurrent evaluation for multiple running queries. We first introduce the plan-bipartite-graph to represent the set of potential query plans for multiple diversity queries. More specifically, given a set of partial $\texttt{D}^+\texttt{-tree}$ indexes IS, a plan-bipartite-graph can be generated for the running diversity queries S_r ($|S_r| \leq N$). In the plan-bipartite-graph, the two sets of nodes are S_r and IS; and there exists an edge between query Q, $Q \in S_r$, and index I, $I \in IS$, if index I can be used to evaluate query Q. For example, given a query set $S_r = \{Q_\alpha, Q_\beta, Q_\gamma\}$ and an index set $IS = \{I_1, I_2, I_3, I_4\}$, an example plan-bipartite-graph is shown in Figure 4.2. As can be seen in Figure 4.2, the two edges (Q_α, I_1) and (Q_α, I_2) indicate that both of index I_1 and I_3 can be used to evaluate to query Q_α.

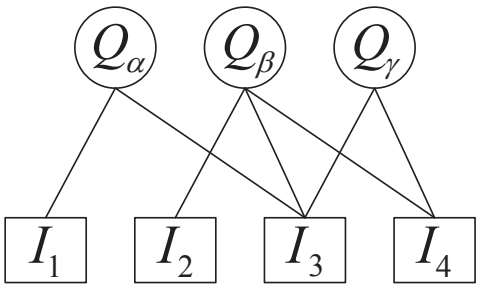


Figure 4.2: An example plan-bipartite-graph for multiple diversity queries

To evaluate the set of diversity queries, one of the straight-forward methods is to independently identify the optimal index for each individual query. In practice, several different indexes could be selected for different queries. When concurrently scanning these indexes which are allocated on the same disk, the performance could be very inefficient to evaluate these queries, due to the random seeks among these disk-based indexes [49].

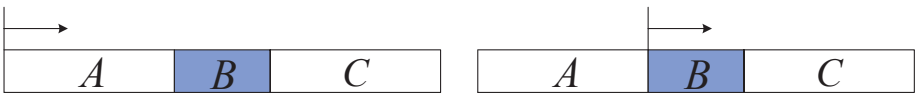
Instead of independently evaluating these queries, we identify the optimal query plans to concurrently evaluate them, by utilizing the techniques of shared index scan and adaptive query evaluation, which are described in the following two subsections.

4.3.1 Query Evaluation Sharing

In this section, we introduce the concurrent evaluation of multiple diversity queries by sharing scans of partial `D`$^+$`-tree` indexes. From the set of potential plans that are represented by the plan-bipartite-graph, an optimal query plan can be identified to evaluate multiple queries. For example, based on the plan-bipartite-graph shown in Figure 4.2, the optimal query plan could be to concurrently evaluate the three queries (Q_α, Q_β and Q_γ) by shared scanning index I_3. The optimization problem of identifying an optimal set of indexes for the set of running queries is a NP-complete problem, since it is reduced from the weighted set cover problem, which is a well-studied NP-complete problem. In this chapter, we use the approximate algorithm proposed in [47] to address our problem, and which is an $\ln\ell$-approximation algorithm, where ℓ is the number of indexes in the system.

For multiple diversity queries that are able to be evaluated by scanning index I, let us now discuss the concurrent evaluation of these queries by shared scanning index I. We first consider a simple case of evaluating multiple queries without picking a new query, and then discuss a more complicated scenario of evaluating multiple queries including a newly picked query. For simplicity, in later sections we only discuss the simple case of concurrently evaluating only two queries by sharing scan of index I. Without loss of generality, we can also generalize to evaluate more than two queries by sharing scan of index I.

Consider a partial $\texttt{D}^+\texttt{-tree}$ index I and two diversity queries Q and Q' that are able to be evaluated by scanning index I. The system can concurrently evaluate both queries by shared scanning I from the beginning. For each accessed entry e_i of I, the system can concurrently evaluate both queries by pipelining entry e_i into the evaluations of the two queries. However, it could be suboptimal to shared scan I from the beginning. Let us consider the case that the evaluation of query Q only needs to scan a small middle portion of index I and that of query Q' needs to scan the whole index I. As can be seen in Figure 4.3(a), the shaded part B of I is the relevant part for query Q. If the scan of I is started from the beginning as shown in Figure 4.3(a), the system is unable to evaluate Q until the scan has reached the part B as shown in Figure 4.3(b).



(a) Scan from part A (b) Scan from part B

Figure 4.3: Shared index scan

To avoid the delay for the evaluation of Q, one possible solution is to concurrently evaluate both queries by shared scanning I started from the beginning of part B as shown in Figure 4.3(b). After the completion of scanning part B, the evaluation of query Q is completed, but the evaluation of query Q' has not be finished. To complete the evaluation of query Q', the system then scans part C and part A of I. Comparing with the previous query plan of scanning I from the beginning, the response time of query Q' does not

change too much, but the response time of query Q has been reduced by avoiding to wait in the period of scanning part A.

Let us take an example to illustrate the shared index scan.

Example 4.1: Consider a partial D$^+$-tree index I on the example laptop relation R (shown in Figure 1.1) with index key (B,C,SS) as shown in Figure 4.4, where e_i is an index entry relevant to the key value, and two following diversity queries Q_1 and Q_2.

Q_1 : **select** $*$ **from** R **where** B = "*Acer*" **diversify by** C, SS **limit** 4

Q_2 : **select** $*$ **from** R **diversify by** B, C **limit** 4

e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}	e_{11}
HP 1 13.3	HP 1 14.1	HP 2 14.1	HP 2 15.4	Acer 2 14.1	Acer 2 15.4	Acer 4 13.3	Acer 4 14.1	Acer 4 17.3	Lenovo 2 14.1	Lenovo 4 13.3

Figure 4.4: D-Index I on R (shown in Figure 1.1) with index key (B,C,SS)

The evaluation of Q_2 needs to fully scan I, while that of Q_1 only needs to access the relevant part of I from entry e_5 to entry e_9, where each entry is of the same brand ("*Acer*"). Assuming that t presents the time of accessing each entry in I. If both queries are evaluated by scanning I from the beginning, the response time of Q_1 is $9 \cdot t$ which is equal to the time of scanning I from e_1 to e_9, while the response time of Q_2 is $11 \cdot t$. On the other hand, if both queries are evaluated by scanning I from e_5, the response time of Q_1 is $5 \cdot t$ which is equal to the access time of the five entries from e_5 to e_9, while the response time of Q_2 is equal to $11 \cdot t$. □

We now consider a more complex one where the shared index scan is among a newly picked query and the running queries. Consider a running query Q_α that is currently evaluated by scanning index I, and a newly picked query Q_β that can also be evaluated by scanning I. Let us assume that the system picks Q_β after accessing entry e in I. Then

the system concurrently evaluates both queries by shared scanning I from the next entry of e. When reaching the last entry of I, the system continues with the evaluation of Q_β by re-scanning I from the first entry of I to entry e.

Let us consider an example to illustrate the shared index scan for a newly picked query.

Example 4.2: Reconsider the two queries Q_1, Q_2 and index I in Example 4.1. At the very beginning, there is only one running query Q_1 in the system, and the system then evaluates Q_1 by accessing I from e_5. Let us assume that the system picks a new query Q_2 after accessing e_6. The system can concurrently evaluate both queries by shared scan I from the next entry e_7. The evaluation of Q_1 will be completed after accessing e_9. To complete the evaluation of Q_2, the system continues to scan e_{10} and e_{11}, followed by accessing I from e_1 to e_8. Therefore, the response time of Q_1 is $5 \cdot t$, where t denotes the time of accessing each entry in I, while the response time of Q_2 is $11 \cdot t$. □

4.3.2 Query Evaluation Switching

While the shared index scan evaluation presented is an improvement over the independent index scan evaluation, the former could also be sub-optimal. Consider a diversity query Q with d-order $\delta = (D_1, \cdots, D_m)$, an index I with index key $(A_1, A_2, \cdots, A_n)$ and another index I' with index key $(A'_1, A'_2, \cdots, A'_\ell)$. Each of the two indexes can be used to evaluate query Q. At the beginning, query Q is evaluated by scanning index I. Assume that the evaluation of Q using I has just completed scanning the gray portion of index I (labeled E) in Figure 4.5(b). Instead of continuing the index scan on the original index I, let us introduce the technique to switch the query evaluation of Q to scan the new index I'. Thus the challenging problem is how to deal with the evaluation of Q on the original index I. Rather than simply giving up this previous evaluation, the better strategy is to re-use it by mapping the previous evaluation into the evaluation on the new index I'.

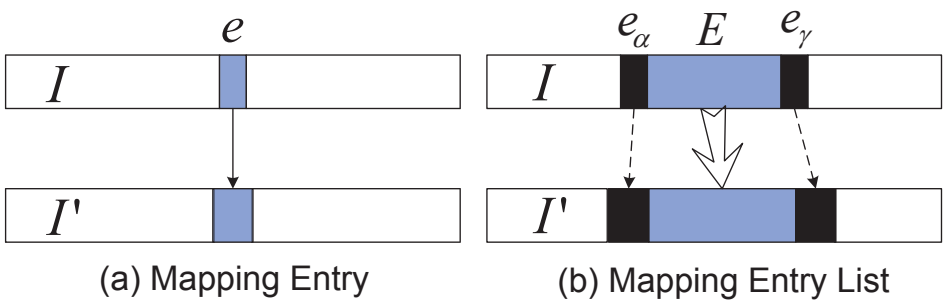


Figure 4.5: The mapping function

e'_1	e'_2	e'_3	e'_4	e'_5	e'_6
HP 1	HP 2	Acer 2	Acer 4	Lenovo 2	Lenovo 4

Figure 4.6: D-Index I' on R (shown in Figure 1.1) with index key (B,C)

Let τ present the length of the maximum common prefix of the two index keys of I and I'. Now we first define a mapping function $M_{I\to I'}(e)$ to map entry $e : (a_1, a_2, \cdots, a_n)$ on I into a list of consecutive entries with key values $(a_1, \cdots, a_\tau, -, \cdots, -)$ on I' as shown in Figure 4.5(a). For example, consider index I with the index key (B, C, SS) (shown in Figure 4.4) and another index I' with the index key (B, C) (shown in Figure 4.6). For entry $e_7 : (Acer, 4, 13.1)$ on I, the mapped entry $M_{I\to I'}(e)$ is $e'_4 : (Acer, 4)$ on I'.

The following result states the sufficient condition of identifying an equivalent entry on I' for each accessed entry e on I.

Lemma 4.1. *Consider two index I and I' that could be used to evaluated some query Q. For any accessed entry e in an index I, there exists such an entry $e' \in M_{I\to I'}(e)$ that result of evaluating Q by using entry e is equivalent to that using entry e'.*

For a list of consecutive accessed entries E on index I, we further define a general mapping function $M_{I\to I'}(E)$ to map E into a contiguous list of entries on I' as shown in Figure 4.5. Let e_α be the last entry in the left of E on index I, e_γ be the first entry in the

right of E on index I. Therefore, $M_{I \to I'}(E)$ can be identified as follows:

$$M_{I \to I'}(E) = \bigcup_{e \in E} M_{I \to I'}(e) - M_{I \to I'}(e_\alpha) - M_{I \to I'}(e_\gamma) \tag{4.1}$$

For example, consider the list of entries (e_7, e_8 and e_9) on I (shown in Figure 4.4). The mapped partition $M_{I \to I'}(E)$ is e'_4.

The following result states the sufficient condition of reusing the previous evaluation on I.

Lemma 4.2. *The evaluation of Q by only scanning I will be equivalent to the evaluation of Q by scanning both of partition E on I and partition $I' - M_{I \to I'}(E)$ on I'.*

Let us consider an example to illustrate the adaptive index evaluation technique.

Example 4.3: Reconsider Example 4.2. At the very beginning, there is only one running query Q_1 in the system, and the system then evaluates Q_1 by accessing I from e_5. The system picks Q_2 after access e_6 on I, and then shared scans I to evaluate both queries. After accessing e_9, the evaluation of Q_1 has been completed. Subsequently, instead of continuing to scan I to evaluate Q_2, the system can map the list of accessed entries $\{e_7, e_8, e_9\}$ into the entry $\{e'_4\}$, and then evaluate Q_2 by scanning the list of entries $\{e'_5, e'_6, e'_1, e'_2, e'_3\}$ on I'. The response time of Q_2 will be $8 \cdot t$, which is less than the response time of Q_2 if we continue the scan on I. □

Furthermore, it is observed that we have $M_{I \to I'}(E) = \emptyset$ if $\tau = 0$. That is, the previous evaluation on index I can not be re-used if there is no common prefix between the two index keys of I and I'. For example, consider another index I'' with index key (C, B). Although I'' can be used to evaluate Q_2 (shown in Example 4.1), the previous evaluation of Q_2 by scanning $\{e_7, e_8, e_9\}$ on I can not be re-used if the system decides to continue the evaluations of Q_2 by scanning I''.

4.4 Online Index Tuning

After discussing the evaluation of multiple online diversity queries given a fixed set of partial D$^+$-tree indexes, we now introduce the technique of automatic index tuning to further optimize these evaluations. In the Section 3.8.2, we have presented an offline algorithm to generate a set of partial D$^+$-tree indexes for a workload of diversity queries under a space constraint, but this algorithm does not work very well in the online environment. In our system, we automatically self-tune the set of physical indexes by looking ahead at those future yet-to-be evaluated queries in the waiting queue. More specifically, the online index tuning in our system consists of two parts: (a) index candidate generation, and (b) index selection for materialization. The first component is to generate a set of candidate indexes, while the second component is to select some candidate indexes for materialization. Figure 4.7 shows the diagram of the index tuning component in our system. For each accessed entry e (shown in Figure 4.7), the system can push it into both the query evaluation component and the index generation component. As mentioned in Section 4.3, the system can concurrently evaluate several different queries by scanning an index I. Based on the accessed index I, a large set of candidate indexes can be generated on-the-fly. Instead of generating and materializing all of them, we look ahead at the waiting queries in the queue, and only generate and materialize a small subset of candidate indexes to improve the performance of these waiting queries.

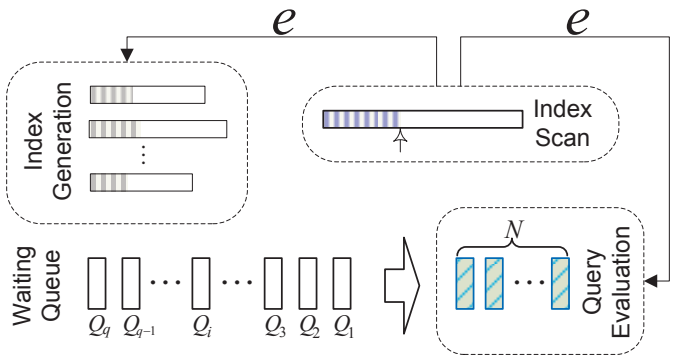


Figure 4.7: The diagram of index tuning component

4.4.1 Generation of Candidate Indexes

In this section, we describe the generation of candidate indexes in detail. We assume that the system is scanning a partial $\mathtt{D}^+\text{-}\mathtt{tree}$ index I with index key $(A_1, A_2, \cdots, A_n)$. To minimize the overhead of generating a new index, we only consider the case that the system is evaluating some queries by entirely scanning the whole index I. In such case, it is guarantee that the index generation component does not need to scan extra parts of index I or additional indexes, since it will slow down the evaluations of current running queries by scanning these extra parts or indexes.

As can be seen in Figure 4.7, after accessing an entry e on index I, the system can push e into the evaluation component for evaluating the currently running queries, and also can simultaneously push it into the index generation component for generating new candidate indexes. Consider a candidate index I' with index key $(A'_1, A'_2, \cdots, A'_\ell)$, where $\{A'_1, A'_2, \cdots, A'_\ell\} \subseteq \{A_1, A_2, \cdots, A_n\}$. Let us now describe how to generate index I' by entirely scanning index I. In the generation, we maintain a hash table $HTable$ to avoid creating duplicate entries for index I'. We initialize $HTable$ to be empty. For each accessed entry $e(a_1, a_2, \cdots, a_n)$ in I, we map it into a new entry $e'(a'_1, a'_2, \cdots, a'_\ell)$, where the key value $(a'_1, a'_2, \cdots, a'_\ell)$ is equal to $\pi_{A'_1, A'_2, \cdots, A'_\ell}(a_1, a_2, \cdots, a_n)$. Based on $HTable$, we can easily check whether there already exists an entry $e''(a''_1, a''_2, \cdots, a''_\ell)$ such that $(a'_1, a'_2, \cdots, a'_\ell) = (a''_1, a''_2, \cdots, a''_\ell)$. If e'' does not exist, we can directly insert e' into $HTable$; otherwise, we can simply discard it. After the completion of accessing index I, we then sort all entries in $HTable$ in order of the index key of I', and organize those sorted entries as the new index I'.

Let IS_I^c present the set of candidate indexes (excluding I itself). In theory, the cardinality of IS_I^c is $\Sigma_{k=2}^{n} P(n,k) - 1$, where $P(n,k)$ is the number of k-permutations of n. For instance, consider accessing an index I with index key (B, C, SS, BL, CL). There are 204 different candidate indexes (e.g. $I_\mu(B, C, SS)$, $I_\nu(B, C, BL, CL)$, $\cdots$).

4.4.2 Index Selection

As mentioned above, the cardinality of IS_I^c will be very large when n is large. So it is not feasible to generate and materialize them all. Let us now discuss how to select several candidate indexes to be generated and materialized. In the online system, the intuition behind index selection is to materialize some indexes to improve the performance of queries in the future. Users are likely to submit a series of similar queries when shopping online. Therefore, in some sense, the waiting queries in the queue represent the future queries. By looking ahead at the waiting queries in the queue, we try to select and materialize several indexes to improve the performance of incoming queries in the future.

Let us now formalize the above intuition of index selection. Let IS be the set of existing available partial $\mathtt{D^+\text{-}tree}$ indexes. In the waiting queue, we assume that there are μ queries, denoted as Q_1, Q_2, $\cdots$, Q_μ. For a waiting query Q_i, $i \in [1, \mu]$, let $C_{IS}(Q_i)$ be the estimated minimum evaluation cost of Q_i based on the index set IS.

Definition 4.1 (Benefit of a Candidate Index)**.** *For a candidate index I', $I' \in IS_I^c$, $I' \notin IS$, and a waiting query Q_i, $i \in [1, \mu]$, we define the benefit of I' for Q_i wrt IS as* $B_{IS}(I', Q_i) = \frac{\mu - i + 1}{N}(C_{IS}(Q_i) - C_{IS \cup \{I'\}}(Q_i))$.

The function $B_{IS}(I', Q_i)$ is to measure the performance improvement based on the new index set including the additional candidate index I'. The response time of Q_i is directly reduced by the period of time $t = C_{IS}(Q_i) - C_{IS \cup \{I'\}}(Q_i)$. Note that the response time of later arrived queries can also be affected. In the case $N = 1$, the response time of each later arrived query Q_j, $j \in (i, \mu]$, can be indirectly reduced by time t. Otherwise, each time N queries can be concurrently evaluated by the system. That is, to complete the evaluations of all later arrived queries (Q_{i+1}, $\cdots$, Q_μ), the system needs to execute about $\frac{\mu - i + 1}{N}$ rounds. Among the these $\mu - i + 1$ queries, the response time of abound $\frac{\mu - i + 1}{N}$ queries can be affected by the performance improvement of Q_i.

Subsequently, we further define the benefit of a set of candidate indexes for a waiting query as follows.

Definition 4.2 (Benefit of a Candidate Index Set)**.** *Given a set of selected candidate indexes IS_I^s, $IS_I^s \subseteq IS_I^c$, the benefit of IS_I^s for Q_i, $i \in [1, \mu]$, wrt IS is defined as $B_{IS}(IS_I^s, Q_i) = \max_{I' \in IS_I^s} B_{IS}(I', Q_i)$. Furthermore, we can define the benefit of IS_I^s for S_w wrt IS as $B_{IS}(IS_I^s, S_w) = \sum_{Q_i \in S_w} B_{IS}(IS_I^s, Q_i)$.*

To identify a set of candidate indexes for materialization, we utilize the "what-if" query plan to use these candidate indexes to estimate the costs of query evaluations, even though these candidate indexes are not physically created. For a candidate index $I' \in IS_I^c$, we use $Cost(I')$ to denote the total cost (IO and CPU cost) of index generation and materialization. Then our goal of index selection is to identify an optimal subset of candidate index $IS_I^s \subseteq IS_I^c$ with the maximum value of $B_{IS}(IS_I^s, S_w) - \sum_{I' \in IS_I^s} Cost(I')$. Since the performance of running queries will be diminished if the system spends too many resources to generate and materialize a large set of candidate indexes, in this thesis, we simply restrict the cardinality of IS_I^s under γ, which is set as $\lceil \frac{N}{8} \rceil$.

The problem of identifying the optimal set of candidate indexes is NP-Complete, since it is reduced from the set-cover problem that is a well-known NP-Complete problem. In this thesis, we use a greedy algorithm shown in Algorithm 4.1 to find an approximate set of selected candidate indexes IS_I^s. In Algorithm 4.1, we iteratively select an candidate index I' with the maximum value of $B_{IS \cup IS_I^s}(I', S_w) - Cost(I')$.

The time complexity of Algorithm 4.1 is $\mathcal{O}(\mu \cdot (|IS| + \gamma |IS_I^c|))$. The system can frequently trigger it to select an approximate set of indexes. For example, the system can trigger it when the number of newly arrived queries is greater than a given threshold. After obtaining the approximate index set IS_I^s, we need to determine whether it is interesting to generate and materialize them. It is known that frequently tuning can result in unwanted oscillations, in which the same indexes are continuously created and dropped. To avoid

Algorithm 4.1: `CandidateIndexSelect` (IS, IS_I^c, S_w, γ)

Input: The existing index set IS, the candidate index set IS_I^c, the set of waiting queries S_w and the maximum number of the selected index set γ
Output: The set of selected candidate indexes IS_I^s

```
1 IS_I^s ← ∅;
2 while |IS_I^s| < γ do
3     Let I' ∈ IS_I^c be the index with the maximum value of
      B_{IS∪IS_I^s}(I', S_w) − Cost(I');
4     if B_{IS∪IS_I^s}(I', S_w) − Cost(I') ≤ 0 then
5         Break;
6     else
7         IS_I^s ← IS_I^s ∪ {I'};
8         IS_I^c ← IS_I^c − {I'};
9 Return IS_I^s;
```

the unwanted oscillations, we utilize the strategy provided in [15] by setting a lower bound $\mathcal{B}$ and only allow to create these indexes if $B_{IS}(IS_I^s, S_w) - \sum_{I' \in IS_I^s} Cost(I') > \mathcal{B}$.

Furthermore, we should first guarantee that the system has enough available disk space to materialize the selected set of candidate indexes. If it has enough space, they can be easily materialized; otherwise, we have to drop some existing indexes that are not very useful. Let Γ denote the lower bound of extra space needed for the materialization of these selected candidate indexes. Therefore, we need to drop such a subset of indexes $IS^d \subset IS$ that the total size of indexes in IS^d is no less than Γ, and the benefit $B_{IS-IS^d}(IS^d, S_w)$ is minimized.

The same as before, the problem is also NP-Complete. We use a greedy algorithm shown in Algorithm 4.2 to free the space of some indexes for materializing those indexes in IS_I^s. In Algorithm 4.2, we iteratively free the space of an index from IS with the minimum benefit until that the freed space is large enough.

Normally, the size of IS^d is roughly equal to that of IS_I^s. Therefore, the time complexity of Algorithm 4.2 is $\mathcal{O}((1+\gamma) \cdot |\mu| \cdot |IS| + |IS_I^s|)$.

Algorithm 4.2: `ReallocateIndexSpace` (IS, S_w, Γ)

Input: The index set IS, the set of waiting queries S_w and the lower bound of space Γ that is needed to be free

```
1 P ← ∅;
2 IS^d ← ∅;
3 while P < Γ do
4 |   Let I' ∈ IS be the index with the minimum benefit B_{IS−IS^d}(IS^d, S_w);
5 |   P ← P+ the size of I';
6 |   IS^d ← IS^d ∪ {I'};
7 |   IS ← IS − {I'};
8 foreach I' ∈ IS^d do
9 |   Free the space of I';
```

4.5 System Implementation

In this section, we describe the implementation of our system. Figure 4.8 shows the diagram of our system architecture. As can be seen in Figure 4.8, our system consists of two parts: the middleware part and the DBMS part. The middleware part is to manage multiple online diversity queries issued by different users, and to pick and send waiting queries into the DBMS part for processing. On the other hand, the DBMS part is to concurrently evaluate multiple diversity queries, followed by returning these diverse result sets to users. In the system, JDBC is used to communicate between the two parts.

The middleware part was implemented in Java. In the middleware part, a query queue is created to maintain the diversity queries issued by different users. As mentioned in previous sections, we improve the opportunity for shared index scan, by supporting query reordering in the query queue. Reordering waiting queries, however, needs some statistic information about query evaluations in the database system. For instance, for a running diversity query, the middleware needs to know which index is being used to evaluate it, and how many index pages are still needed to be accessed. To fetch such kind of information from the database system, a *Notifier* module is designed to periodically notify relevant kernel information of DBMS to the middleware. Based on these notified information, the middleware can get a snapshot of the current state of the database system. Consequently,

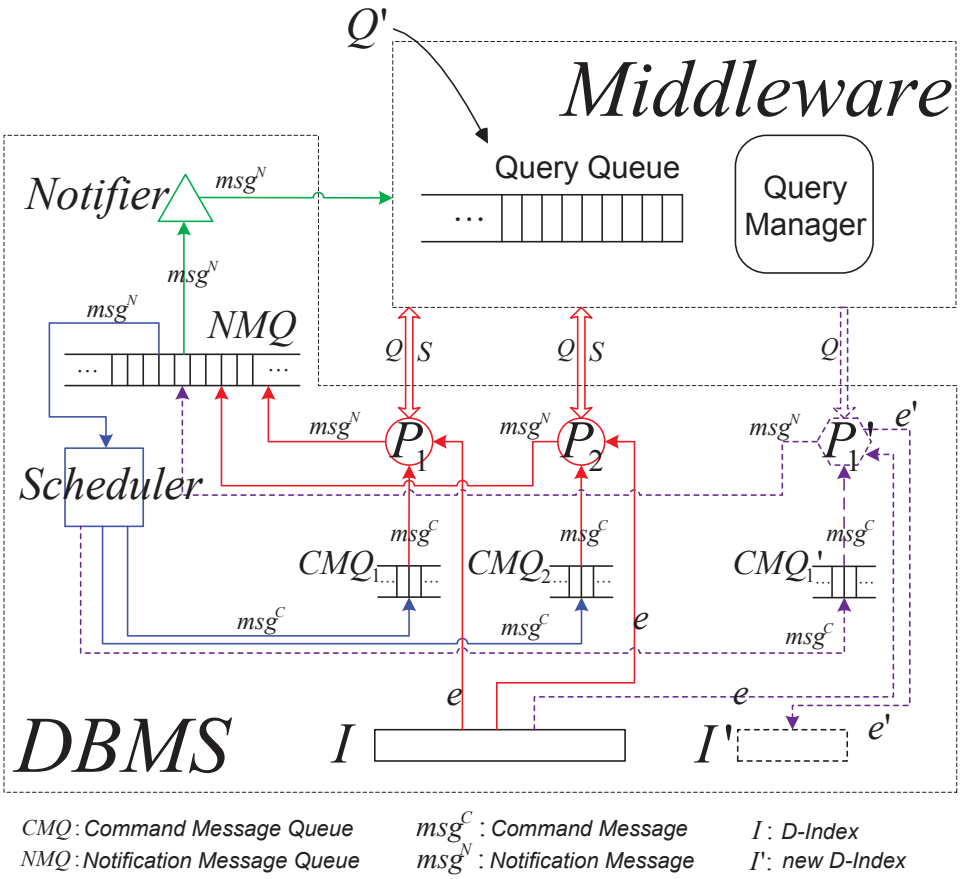


Figure 4.8: The implementation of our system

the middleware can more precisely pick the next query to maximize shared index scans.

We implemented the multiple query optimization component in PostgreSQL 9.0.2, which contains several background worker processes. The PostgreSQL server maintains a process pool that contains N processes for evaluating queries. When receiving a query, the server will find an idle process to evaluate the query. In the DBMS part, the master/slave model is used to communicate among different processes to concurrently evaluate several different diversity queries. More specifically, we have designed a *Scheduler* module (master) to control the concurrent executions of multiple queries. To support the communication between the *Scheduler* (master) and each process (slave) that is to evaluate an individual diversity query or to generate a new D-Index, we have designed two different kinds of communication channels: the *Command Message Queue* (CMQ), and the *Notification Message Queue* (NMQ).

Message	Operation
start command message	command the *Process* to start the query execution
execute command message	command the *Process* to continue the execution
switch command message	command the *Process* to switch to scan a new index

Table 4.2: Command messages send from the *Scheduler*

For each process P, there is a unique CMQ used for communicating between P and the *Scheduler*. In our system, we have defined three different kinds of command messages: (a) start command message, (b) execute command message, and (c) switch command message. As can be seen in Table 4.2, the start command message is to command the *Process* to start a query evaluation on a partial `D+-tree` index, the execute command message is to command the *Process* to continue a query evaluation by conducting a disk page read (multiple disk pages could be read in each time), and the switch command message is to command the Process to switch its query evaluation to scan a different partial `D+-tree` index. Based on these command messages, the *Scheduler* can precisely control the actions of each process.

Message	Operation
initialize notification message	inform the *Scheduler* of a new query to be evaluated
start notification message	inform the *Scheduler* of the execution that has just started
execute notification message	inform the *Scheduler* of a query execution that has just conducted an index page read
switch notification message	inform the *Scheduler* of the execution that switches to scan a different index
end notification message	inform the *Scheduler* of the completion of a query evaluation

Table 4.3: Notification messages send from a *Process*

For a process, when taking actions, it also needs to inform the *Scheduler* of this action. A NMQ is created to maintain all notification messages from different processes to the *Scheduler*. There are five different kinds of notification message: (a) initialize notification message, (b) start notification message, (c) execute notification message, (d) switch notification message and (e) end notification message. As can be seen in Table 4.3, the initialize notification message is to inform the *Scheduler* of a new query that needs to be evaluated, the start notification message is to inform the *Scheduler* of the query eval-

uation that has just started, the execute notification message is to inform the *Scheduler* that a query evaluation that has just conducted an index page read, the switch notification message is to inform the *Scheduler* that a query evaluation has just switched to scan a new partial D$^+$-tree index, and the end notification message is to inform the *Scheduler* of the completion of a current query evaluation.

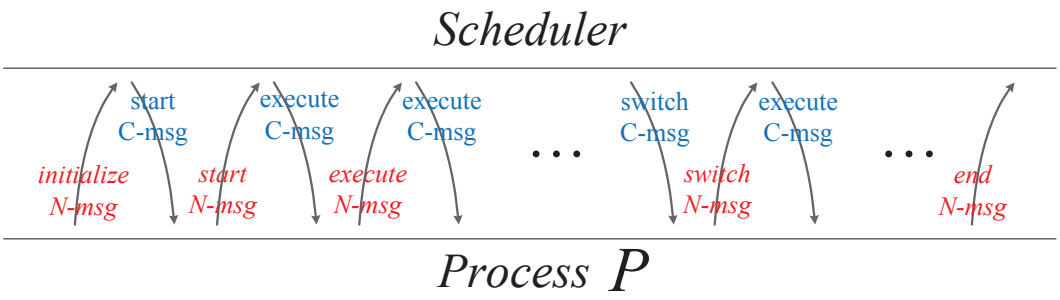


Figure 4.9: Example communication between the *Scheduler* and a *Process*

Figure 4.9 shows the communication between the *Scheduler* and a *Process* P. In Figure 4.9, C-msg and N-msg present the command message and the notification message, respectively. A waiting query Q is picked to be evaluated if there exists an idle *Process* P in the process pool of the DBMS. P is now ready to evaluate Q, and it sends an *Initialize Notification Message* to the *Scheduler*. After receiving this notification message, the *Scheduler* finds an optimal partial D$^+$-tree index I for evaluating for Q, followed by sending a *Start Command Message* to P. After receiving this message that contains the information of index I, P starts to evaluate Q by scanning I (shared scan a particular dis page with other queries or scan from the beginning), followed by informing the *Scheduler* when the first page has been processed. The *Scheduler* commands P to continue the query evaluation by accessing the next index page. When the pages of a disk read have been processed, P informs the *Scheduler*. In this way, P iteratively process the next disk read to evaluate Q. In some cases, the *Scheduler* could decide to switch the query evaluation to scan another index I', and then it sends a *Switch Command Message* to P. After receiving this message that contains the information of the new index I', P switches the query evaluation to scan index I'. P then iteratively scans each disk page until that the

evaluation of Q has been completed. Consequently, when the query evaluation has been completed, P sends a *End Notification Message* to inform the *Scheduler* of the completion of query evaluation.

Furthermore, as mentioned in Section 4.4, our system is capable of online index tuning to further optimize query evaluation. After deciding to create a new index, the middleware can issue a special "query" to inform the database system to generate the new index. The evaluation of the special "query" can also share the index scan with other normal diversity queries as shown in Figure 4.8.

4.6 Performance Study

We conducted an experimental study to evaluate the effectiveness of our proposed framework. Section 4.6.1 compares the performance of shared index scan and switched index techniques by considering two queries, and Section 4.6.2 compares their performance based on a query workload.

Data sets. We used the same four synthetic tables R_1(1G), R_2(5G), R_3(10G) and R_4(15G) from the previous chapter, and the properties of these tables are shown in Table 3.2. Each R_i consists of 10 attributes, denoted as $A, \cdots, J$.

Database Indexes We generated a set of partial $\mathtt{D}^+\mathtt{-tree}$ indexes on each R_i under the space constraint of 20% of the table size. Specifically, for each database R_i, we generated six partial $\mathtt{D}^+\mathtt{-tree}$ indexes as follows: $I_1(A, B, C, D, E, F, G, H, I)$, $I_2(A, B, C, D, E, F, G, H, I, J)$, $I_3(A, B, C, D, E, F, G, H)$, $I_4(A, B, C, D, E, F, G)$, $I_5(A, B, C, D, E, F)$ and $I_6(A, B, C, F, G, H)$.

Algorithms Based on the set of partial $\mathtt{D}^+\mathtt{-tree}$ implemented in PostgresSQL 9.0.2, we compared the performance of the following five evaluation algorithms:

- `Concurrent`: this strategy concurrently processes multiple diversity queries, each of which is individually evaluated by selecting the optimal partial `D+-tree` index to scan;

- `Sequential`: this strategy sequentially processes each diversity query using a single database server;

- `ConcurrentSharedScan`: this strategy concurrently processes multiple diversity queries, by supporting shared index scan technique;

- `ConcurrentSwitchedScan`: concurrently process multiple diversity queries, by supporting both shared index scan and switched index evaluation techniques;

- `ConcurrentTuning`: concurrently process multiple diversity queries, by supporting all the proposed techniques (shared index scan, switched index evaluation, online index tuning).

The first two strategies are baseline algorithms and the remaining three strategies are our proposed techniques.

Parameters. We varied the following two experimental parameters: (1) the number of clients (background worker processes in PostgreSQL) with a default value of 4, (2) the size of reorder window with a default size of 10.

The experiments were conducted on a server with an Intel Xeon 1.80GHz processor, 32GB of memory, two 1TB disks, running CentOS 3.13.0. Both the operating system and PostgreSQL were installed on the one disk, while the database was installed on the other disk. In our experiments, we measured the response time for each query in the workload, and the total execution time of the query workload.

4.6.1 Comparison for two queries

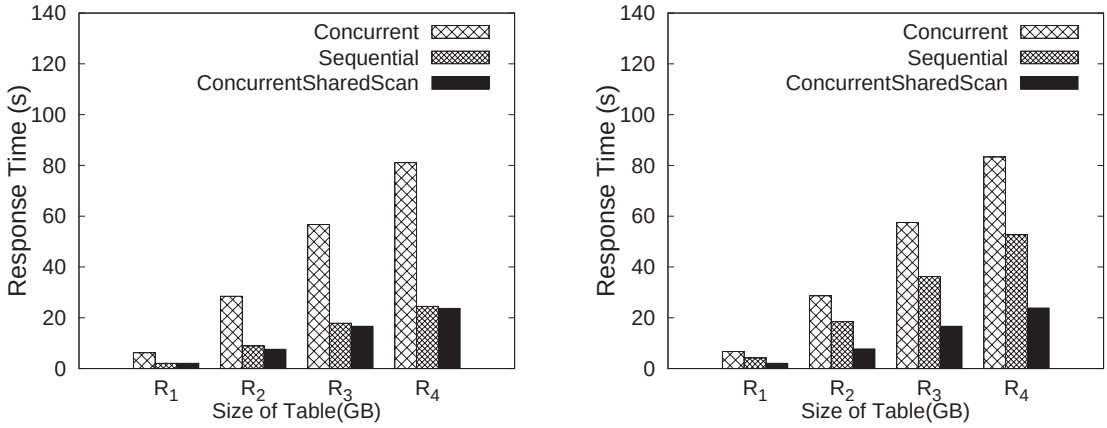
To study the performance of `ConcurrentSharedScan` and `ConcurrentSwitchedScan`, we first consider the two diversity queries shown in Figure 4.10.

Query	Selection Predicate Attribute	d-order	Limit Size
Q_1	C	I, F, A, D, B	10
Q_2	C	J, F, A, D, B	10

Figure 4.10: Two diversity queries

Firstly, we consider the case that the two queries arrive at the same time. `Concurrent` can concurrently evaluate the two queries by only simultaneously scan partial `D+-tree` index I_1 $(A, B, C, D, E, F, G, H, I)$ and partial `D+-tree` index I_2 $(A, B, C, D, E, F, G, H, I, J)$; `Sequential` can evaluate Q_1 by scanning I_1, followed by evaluating Q_2 by shared scanning I_2; `ConcurrentSharedScan` can concurrently evaluate the two queries by only scanning I_2. Figure 4.11 shows the response time of the two queries on different data sizes by applying `Concurrent`, `Sequential` and `ConcurrentShar-edScan` , respectively. Note that the response time of Q_2 by applying `Sequential` includes the waiting time for the evaluation of Q_1. Figure 4.12 compares the total execution time of the two queries. As can be seen in Figure 4.12, `ConcurrentSharedScan` outperforms the other two algorithms. Furthermore, the performance of `Concurrent` is the worse one, due to the random seeks on the two different indexes.

Subsequently, to evaluate the performance of `ConcurrentSwitchedScan`, we assume that the two queries do not arrive at the same time: the system first pick query Q_1, followed by receiving query Q_2 one second later. The two algorithms `Concurrent` and `Sequential` performed the same as in the previous case. Let us now compare the executions of the two queries by using `ConcurrentSharedScan` and `ConcurrentSwitchedScan` . When Query Q_1 arrives, both of the algorithms choose I_1 to evaluate Q_1. After Query Q_2 arrives, `ConcurrentSharedScan` has to delay the execution



(a) Query Q_1 (b) Query Q_2

Figure 4.11: Response time for two diversity queries

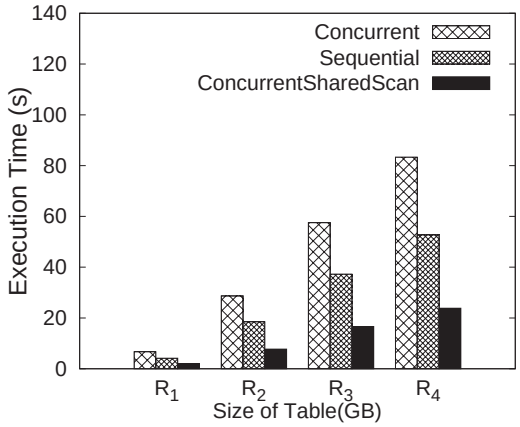
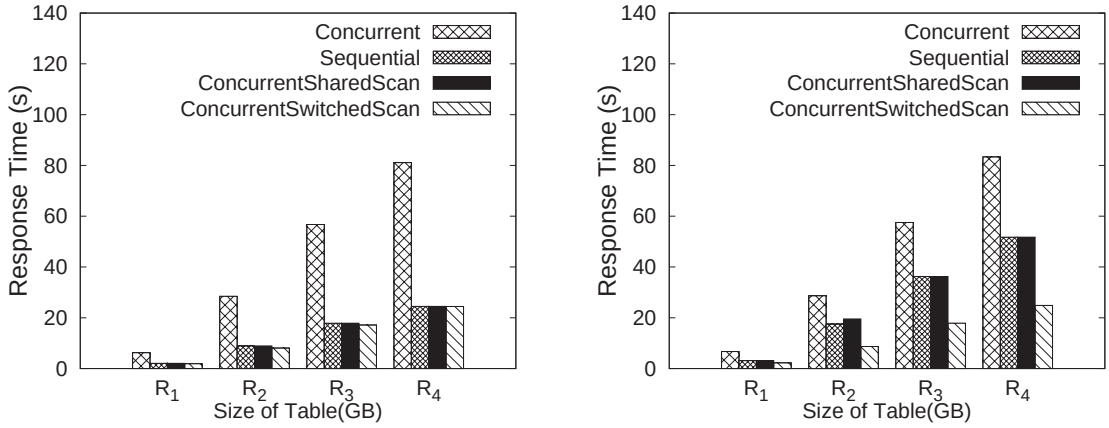


Figure 4.12: Total execution time

of Q_2 since the currently accessed index I_1 is not able to be used to evaluate Q_2. Therefore, the performance of `ConcurrentSharedScan` is similar with `Sequential`. On the other hand, `ConcurrentSwitchedScan` can switch the evaluation of Q_1 to scan I_2. Then `ConcurrentSwitchedScan` can scan I_2 to concurrently evaluate the two queries. Figure 4.13 shows the response time of the two queries on different data sizes by applying the four algorithms. For both of `Sequential` and `ConcurrentSharedScan`, the response time of query Q_2 includes the waiting time of the evaluation of query Q_1 by scanning index I_1. Figure 4.14 shows the total execution time of the two queries. As can be seen in Figure 4.14, `ConcurrentSwitchedScan` outperforms `ConcurrentShar-edScan` by a factor of 2.



(a) Query Q_1 (b) Query Q_2

Figure 4.13: Response time for two diversity queries

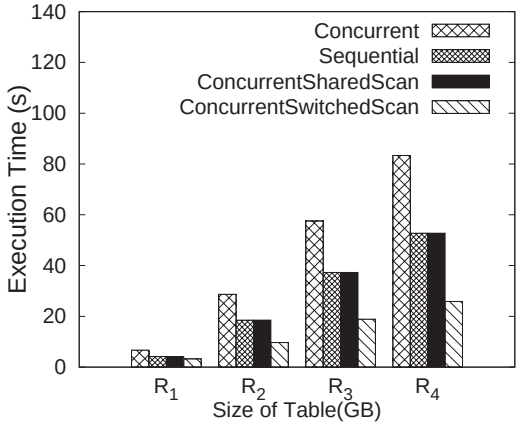


Figure 4.14: Total execution time

4.6.2 Comparison for a query workload

We now compare the performance of the five algorithms for a query workload. To simulate the online environment, we randomly generated a stream of queries at a constant rate, and each query was generated based on the parameters shown in Table 4.4. In our experiment, we randomly generated a workload including 50 diversity queries, which arrives at different time. In the workload, the query arrival rate was one query per second.

Query Parameter	Value
# of predicates	[1-3]
length of d-order	[4-10]
limit size k	10 - 50

Table 4.4: Parameters for Diversity Queries

Effectiveness of online index tuning

In this experiment, we examine the effect of self-tuning the set of partial `D+-tree` indexes. Figure 4.15-4.16 show the performance comparison between `ConcurrentSwit-chedScan` and `ConcurrentTuning`. In `ConcurrentSwitchedScan`, we use the default set of partial `D+-tree` indexes. As can be seen in Figure 4.15, for some query (e.g. Q_3, Q_4, Q_{29} and Q_{35}), its execution time is rather slow due to the large size of the accessed index such as I_1 and I_2. In `ConcurrentTuning`, the system can generate a new index $I_7(A, B, C, D, F, G, I, J)$ when shared scanning index I_2 to concurrently evaluate query Q_4 and Q_7. Due to the space constraint, the system removes index I_3. As can be seen in Figure 4.16, the system performance can be improved by choosing index I_7 rather than I_2 to evaluate some subsequent queries, such as Q_{29} and Q_{35}.

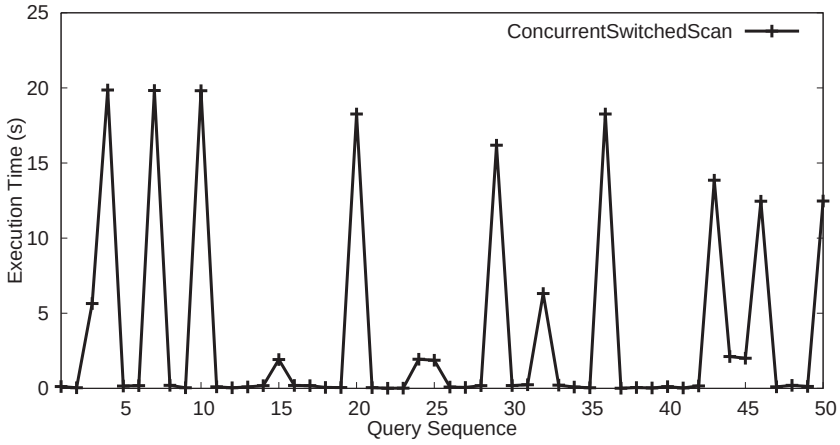


Figure 4.15: The performance of `ConcurrentSharedScan`

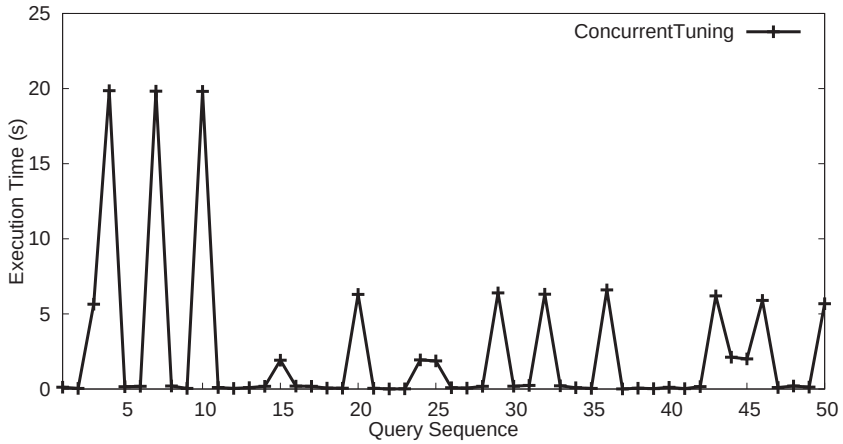


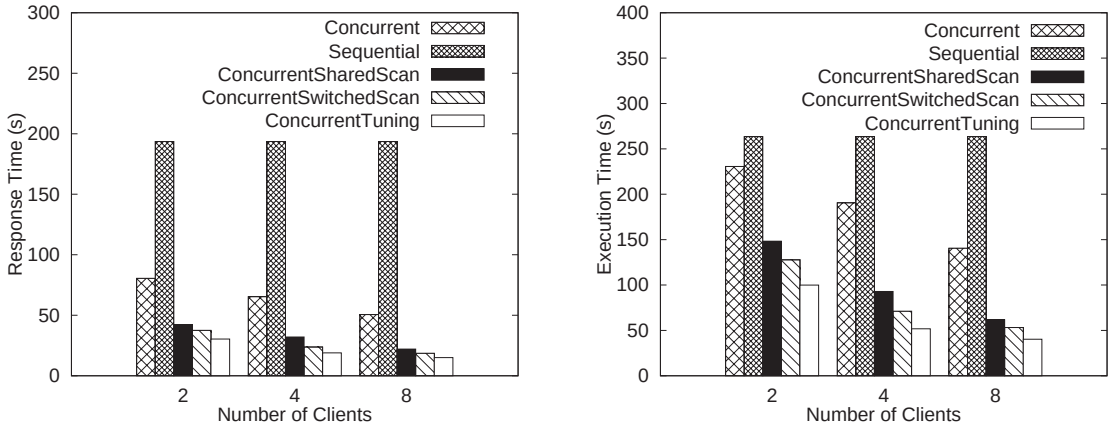
Figure 4.16: The performance of `ConcurrentTuning`

Varying the number of clients

In this experiment, we examine the effect of varying the number of clients. Figure 4.17 shows the performance of the five algorithms when varying the number of clients.

As can be seen in Figure 4.17, our three algorithms consistently outperform the two baseline algorithm. Furthermore, different from the results mentioned in Section 4.6.1, `Concurrent` outperforms `Sequential`. The reason behind is that some index pages are buffered in the system when evaluating queries, and these buffered index page could benefit the concurrent evaluation of subsequent queries.

When increasing the number of clients, the performance is improved for our three algorithms. However, when increasing the number of clients from n_1 to n_2, the improvement factor is smaller than n_2/n_1. There are two main reasons: (1) not all of the running query evaluations are able to shared index scan with each other, and (2) the communication cost among these clients will increase when increasing the number of clients.



(a) Average Response Time (b) Total Execution Time

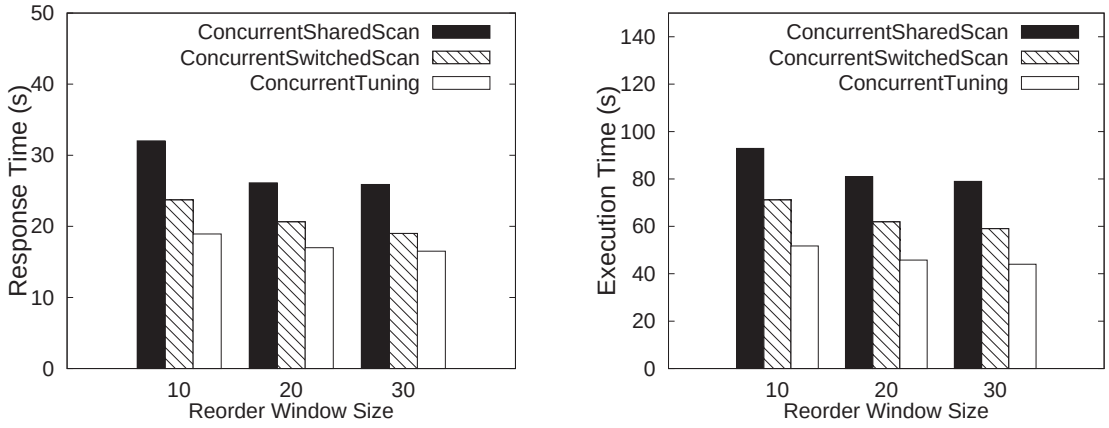
Figure 4.17: Varying the number of clients

Varying the size of reorder window

In this experiment, we examine the effect of varying the size of the reorder window. Figure 4.18 shows the performance of our three algorithms when varying the size of the reorder window. As can be seen in Figure 4.18, when increasing the size from 10 to 20, both of the average response time and the total execution time are improved by around 20%. The reason of this improvement is that increasing the size of the reorder window can improve the opportunity for shared index scans. However, when increasing the size from 20 to 30, the performance is only slightly improved. Even though it can further improve the opportunity of shared index scans, our system will avoid to skip over a large number of queries to first pick a later arrived query. If such a query is picked, the average response time could be increased due to the execution delay of a large number of skipped queries.

4.7 Summary

In this chapter, we have examined the problem of optimizing multiple online diversity queries. To optimize the evaluations of multiple diversity queries, we have proposed a new



(a) Average Response Time (b) Total Execution Time

Figure 4.18: Varying the size of reorder window

framework to allow to reorder an online query, and presented a new technique to support to switch a query evaluation from scanning an index to a different index, and implemented a new online index tuning technique to automatically update the set of indexes. Our comprehensive performance study shows the efficiency of our proposed techniques.

CHAPTER 5

DIVERSIFIED SPATIAL KEYWORD SEARCH

5.1 Overview

In this chapter, we study the problem of diversified spatial keyword search and design two types of spatial diversity queries: DSQ, and N-DSQ. Given a set of keyword concepts, each of which could contain multiple words (e.g. "*Singapore Restaurant*"), a DSQ query can be issued to find top-k groups of diversified objects which collectively cover all keyword concepts and are closely located. For example, reconsider the simple DSQ query with only one keyword concept "*Singapore Restaurant*". In Figure 1.3(a), the top-2 groups G_1 and G_2 are returned, and each one of the two groups contains three restaurants that provide three different kinds of cuisines. Although group G_3 contains one more

restaurant than each of the top-2 groups, we prefer not to return G_3 since all of the four restaurants provide the same cuisine ("*Western Food*"). If a user is not interested in western food, group G_3 will not satisfy user's preference. For another example, reconsider the DSQ query with multiple keyword concepts "*Singapore Restaurant*" and "*Entertainment Facility*". In Figure 1.3(b), the top-2 groups G'_1 and G'_2 are returned, and each one contains both restaurants and entertainment facilities. For the top-k returned groups for a DSQ query, it is possible that some of them could be highly overlapped. For example, the top-2 groups G'_1 and G'_2 (shown in Figure 1.3(b)) for the previous DSQ query are highly overlapped. Therefore, instead of only focusing on the semantic diversity among the objects within each result group, we extend DSQ to a new type of query N-DSQ for returning the top-k spatially diversified groups by taking account of the inter-group distance among different groups. For example, reconsider the N-DSQ query with keyword concepts "*Singapore Restaurant*" and "*Entertainment Facility*". In Figure 1.3(b), the two spatially diversified groups G'_1 and G'_3 are returned.

In this chapter, we show that some existing spatial indexes can be extended to evaluate the new types of queries, but they are very inefficient. To efficiently evaluate the two new types of queries, we propose a novel textual-first spatial index, named IOQ-tree index. In an IOQ-tree index, each partition corresponding to a keyword concept is organized based on a new structure called OQ-tree, where several upper bound scores are maintained in each node. We further introduce two index variants of OQ-tree to organize each partition, named OQ$^+$-tree and OQ*-tree.

For each type of spatial queries, we propose two efficient evaluation methods based on the two proposed index variants. The key idea of these evaluation methods is to reduce the search space by using suitably maintained upper bound scores to filter out as many nodes in OQ-trees as possible. More specifically, the first evaluation method is based on the top-down traversal of the OQ$^+$-tree index, while the second evaluation is based on the sorted list maintained in the OQ*-tree index. We demonstrate the efficiency of our

proposed approaches with a comprehensive experimental evaluation which shows that our approaches outperform the state-of-the-art algorithms by up to one order of magnitude.

For convenience, the notation table of this chapter is provided in Table 5.1, and the rest of this chapter is organized as follows. Section 5.2 presents the formal definitions of our proposed two types of queries. We then present the problem challenges in Section 5.3. A novel index is described in Section 5.4. Based on the proposed index, we present evaluation methods for the two different types of queries in Section 5.5 and Section 5.6, respectively. Subsequently, we present a performance study in Section 5.7. Finally, we conclude this chapter in Section 5.8.

$\mathcal{D}$	The spatial object database	o, o_i, o'_i	A spatial object in the database
$\mathbb{R}$	The whole rectangular region for each OQ-tree	$dis(p_1, p_2)$	The distance between two geo-location points p_1 and p_2
c, c', c_i	A keyword concept	$Q.\psi$	The keyword concepts of Q
ζ_c	The relevant sub-concepts of c	$\zeta_c(o)$	The relevant sub-concepts of c covered by object o
$cov_c(o)$	The coverage score of o wrt c		
r	The query radius	R_Q	The query region
k	The query limit size	$Q(\mathcal{D})$	The query result of Q on $\mathcal{D}$
G, G_i, G'_i	A candidate result group	$\xi_\psi(G)$	The ranking score of G wrt ψ
$\xi_c(G)$	The diversity score of G wrt c	I	The IOQ-tree index
$f_\psi(S)$	The spatial diversity ranking of S wrt ψ	T_c	The OQ-tree that corresponds to c
T	An OQ-tree	N_{root}	The root node of an OQ-tree
N	A node in an OQ-tree	N^c_{root}	The root node of T_c
N_{id}	A node in an OQ-tree with id as node ID	N^c_{id}	A node in T_c with id as node ID
κ	# of maintained scores	$N.\varepsilon_i$	The i-th score in N
$N.\varepsilon^\ell_i$	The hierarchical i-th score in N for descendants at level ℓ	$N.\varepsilon(r)$	The r-score of N
		Γ	The diameter of R_Q

Table 5.1: Notation table for Chapter 5

5.2 Problem Definition

Let $\mathcal{D}$ be a set of spatial objects. Each object $o \in \mathcal{D}$ is associated with a geo-location denoted by $o.\lambda = (o.lat, o.long)$ and a set of keyword concepts denoted by $o.\psi$. For

any two geo-location points p_1, p_2, we use $dis(p_1, p_2)$ to denote the Euclidean distance between the two points.

For each keyword concept c, there are several relevant sub-concepts that cover different aspects of c [5, 85]. Let ζ_c present the set of relevant sub-concepts of c. We use $\upsilon_c(c')$, $\upsilon_c(c') \in [0, 1]$, to denote the weight of a relevant sub-concept c', $c' \in \zeta_c$, such that $\sum_{c' \in \zeta_c} \upsilon_c(c') = 1$. Intuitively, with respect to keyword concept c, the higher the weight $\upsilon_c(c')$ is, the more relevant is the sub-concept c' to c. The sub-concept weights can be determined by analyzing the historical behavior of users [5]. For example, for keyword concept c : "*Singapore Restaurant*", the set of relevant sub-concepts could be $\zeta_c = \{$ "*Chinese Food*", "*Japanese Food*", "*India Food*", "*Western Food*", "*Korean Food*", "*Malay Food*" $\}$. Table 5.2 shows an example for the weights of relevant sub-concepts wrt the keyword concept "*Singapore Restaurant*".

Relevant Sub-concept	Chinese Food	Japanese Food	Indian Food	Western Food	Korean Food	Malay Food
Weight	0.25	0.125	0.15	0.175	0.15	0.2

Table 5.2: The weights of relevant sub-concepts of "*Singapore Restaurant*"

We are now ready to formally define the two types of novel diversified spatial keyword search queries.

5.2.1 DSQ Query

Consider a DSQ query $Q\ (\psi, r, R_Q, k)$, where ψ is the set of query keyword concepts, r denotes the query radius, R_Q is the query rectangular region and k represents the limit size of query.

Let us first define a candidate result group for Q as follows:

Definition 5.1 (A candidate result group)**.** *For a DSQ query $Q\ (\psi, r, R_Q, k)$, a candidate result group G, $G \subseteq \mathcal{D}$, is defined as a set of objects that collectively cover the keyword*

concepts ψ and are located in a circle of radius r within the query region R_Q. That is, there exists a geo-location point p such that $G = \{o \in \mathcal{D} : o.\psi \cap \psi \neq \emptyset \wedge o.\lambda \in R_Q \wedge dis(o.\lambda, p) \leq r\}$ and for each keyword concept c, $c \in \psi$, there always exists at least one object o in G such that $c \in o.\psi$.

For example, consider a DSQ query Q with two keyword concepts "*Singapore Restaurant*" and "*Entertainment Facility*". Group G'_1 shown in Figure 1.3(b) is a candidate result group for Q, and G'_1 contains three restaurants and two entertainment places which are within a circle. However, group G_2 shown in Figure 1.3(a) is not a candidate result group for Q since there does not exist an entertainment place in G_2.

With respect to a keyword concept c, $c \in \psi$, intuitively, we prefer to highly rank a candidate result group G which covers more sub-concepts of c. We use $cov_{c'}(o)$ to denote the coverage score of an object o wrt a specific subtopic c', $c' \in \zeta_c$, and the coverage score is defined as follows:

$$cov_{c'}(o) = \begin{cases} 0 & \text{if } c' \notin o.\psi, \\ P(o|c') & \text{if } c' \in o.\psi \end{cases} \tag{5.1}$$

where $P(o|c')$ is the probability that o is relevant to c' [5]. The probability can be estimated based on some existing models [5, 73, 76], and in this thesis, we compute it using Dirichlet method [76].

Based on the coverage-based diversification model [85], we compute the diversity score of a candidate result group G wrt a keyword concept c, $c \in \psi$, as follows:

$$\xi_c(G) = \sum_{c' \in \zeta_c} v_c(c') \cdot \ln\left(1 + \sum_{o \in G} cov_{c'}(o)\right) \tag{5.2}$$

The logarithm function in Equation 5.2 is to ensure the decrease of gain when adding one more object covering the sub-concept that has already been well covered. Intuitively, the benefit of adding an object covering the sub-concept c' should be smaller if c' has already

been well covered by G and this is desired from end users' viewpoints [29].

The result of Q on database $\mathcal{D}$ is the top-k result groups based on the ranking function as follows:

$$\xi_\psi(G) = \frac{1}{|\psi|} \sum_{c \in \psi} \xi_c(G) \tag{5.3}$$

5.2.2 N-DSQ Query

A N-DSQ query extends DSQ query and takes the spatial diversity into account. Given a N-DSQ $Q(\psi, r, R_Q, k)$, intuitively, the top-k spatially diversified result groups should have high ranking scores defined in Equation. 5.3 and spatially disperse within the query search region.

Given two result groups G and G', we use $Dis(G, G')$ to denote the distance of the two groups. Let us now discuss how to measure the distance $Dis(G, G')$. For a result group G, we say $p(G)$ is a geo-center of G, if all objects in G are located within the circle of radius r with p as circle center. The distance $Dis(G, G')$ can be simply measured by $dis(p(G), p(G'))$. Unfortunately, there could be more than one geo-center for each group. Instead, for group G, we further define the mass center $\sigma(G)$ as $\sigma(G) = (\frac{\sum_{o \in G} o.lat}{|G|}, \frac{\sum_{o \in G} o.long}{|G|})$. Thus, the distance $Dis(G, G')$ is measured by $dis(\sigma(G), \sigma(G'))$ which is the Euclidean distance between the two mass centers $\sigma(G)$ and $\sigma(G')$.

The result of Q is defined to be the set S, $|S| = k$, of top-k spatially diversified result groups based on the the popular *max-sum diversification* function [19, 42]:

$$f_\psi(S) = \frac{\delta}{k} \cdot \sum_{G \in S} \frac{\xi_\psi(G)}{\xi_{max}} + \frac{1-\delta}{k(k-1)} \cdot \sum_{G, G' \in S} \frac{Dis(G, G')}{\Gamma} \tag{5.4}$$

where ξ_{max} is the normalized factor which is the maximum ranking score among all result groups with circle area of radius r, Γ is the normalized factor which is the diameter of

query region, and the factor δ, $\delta \in [0, 1]$, is used to balance between the ranking scores of these result groups and the distances among these result groups. When $\delta = 1$, the N-DSQ query is reduced to a DSQ query. Note that the *max-sum diversification* problem has been proved to be NP-Complete [42], since it can be reduced from the classic MAX-SUM-DISPERSION problem.

5.3 Challenges For Spatial Diversity Query

To motivate the need for a new approach to evaluate the two types of queries proposed in this chapter, we argue that existing spatial indexes are very inefficient to evaluate the new queries. For a spatial keyword query with a small number of query keywords, it is reported [25, 82] that the textual-first spatial indexes outperform the other two kinds of spatial indexes (the spatial-first index and the hybrid spatial index) since only those textual postings lists corresponding to query keyword concepts need to be accessed. Let us now discuss how to evaluate a new type of query (DSQ or N-DSQ) by using `I3-index` [82] and `S2I` [59], which are the state-of-the-art textual-first spatial indices for the top-k spatial keyword search. Furthermore, we show the inefficiency of these evaluations.

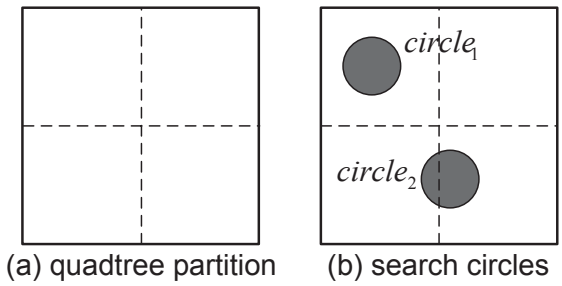


Figure 5.1: Searching over basic quadtree node

The `I3-index`[82] maintains an inverted postings list for each keyword concept, and each postings list is organized as a quadtree, where each node corresponds to a rectangular region, and the region of each internal node is partitioned into four non-overlapped sub-regions of the same size shown as in Figure 5.1(a). Consider a simple DSQ query

$Q(\psi, r, R_Q, k)$, where $|\psi| = 1$. The query evaluation of Q needs to search the quadtree that corresponds to the query keyword concept $c \in \psi$. Note that any candidate result group must be within a circle of radius r. To find out the top-k result groups, the query evaluation has to search these circles, which could be within a single node or span several neighboring nodes. For example, as can be seen in Figure 5.1(b), $circle_1$ is within the left-top node, while $circle_2$ spans the two bottom nodes. For circles within a single node, the query evaluation can enumerate them to find the highly ranked candidate result groups by applying the existing *circle-placement* algorithm [23] with a time complexity of $\mathcal{O}(n^2)$, where n is the number of objects located in the node. Additionally, some upper bound scores can be maintained in each node to improve the query evaluation, by efficiently filtering out some sparse nodes which do not contribute to the final top-k result groups. On the other hand, for those circles spanning several neighboring nodes, the evaluation has to enumerate and search a large number of node combinations where the distance between every two nodes is no larger than $2 \times r$. It is very costly to enumerate too many node combinations. Unfortunately, there does not exist an efficient strategy that can be used to improve the query evaluation by avoiding to search too many node combinations. More generally, for a query (DSQ or N-DSQ) with multiple keyword concepts, the query evaluations are also inefficient due to the large search space.

In `S2I` [59], each inverted postings list is organized as a R-tree. Similarly, for a simple DSQ query with radius r and a single keyword concept c, the query evaluation has to search the R-tree that corresponds to c, and enumerate a large number of node combinations where the distance between the Minimum Bounding Rectangles (MBRs) of every two nodes is no larger than $2 \times r$. More generally, for a query (DSQ or N-DSQ) with multiple keyword concepts, the query evaluation has to spatial join among several different R-trees. Comparing with the spatial join on multiple quadtrees based on a uniform space decomposition mechanism, the spatial join on different R-trees is very costly [82].

5.4 The IOQ-tree Index

An efficient index is required to support the evaluations of the two new types of spatial queries. In this section, we introduce a novel textual-first spatial index, named IOQ-tree (Inverted Overlapping-Quadtree), to manipulate the inverted postings list for each keyword concept.

For each spatial object o, $o \in \mathcal{D}$, we represent it using a set of $|o.\psi|$ tuples. Each tuple t is associated with only one keyword concept $c \in o.\psi$, and is in the following format:

$$t = \langle t.id, t.c, o.id, o.\lambda, \zeta_c(o) \rangle$$

Here, the tuple inherits the object id and the location coordinate from the object o. $\zeta_c(o)$, $\zeta_c(o) = \zeta_c \cap \zeta_c(o)$, represents the set of relevant sub-concepts of keyword concept c covered by the object o.

After partitioning these tuples according to their keyword concepts, all tuples corresponding to the same keyword concept are organized as an OQ-tree (Overlapping-Quadtree), which is a variant of Quadtree. There are two important goals that OQ-tree aims to achieve. Towards an efficient search, the first is to guarantee that each query circle only falls into a small number of index nodes to eliminate the high overhead of enumerating many node combinations. The second is to achieve an efficient query evaluation by pruning the search space. The first goal is achieved by providing a new space decomposition mechanism that is introduced in Section 5.4.1, while the second goal is achieved by maintaining summary information in each node which is introduced in Section 5.4.2. To efficiently support query evaluations, we further introduce two index variants of OQ-tree in Section 5.4.3. Finally, the data operations of OQ-tree are described in Section 5.4.4.

5.4.1 OQ-tree Index Structure

In this section, we introduce a disk-based space-partitioning tree structure OQ-tree for storing all tuples that corresponds to the same keyword concept. Each index node N corresponds to a rectangular region, denoted by $N.R$. All tuples are stored in disk pages, each of which is pointed to a leaf node in an OQ-tree. When the disk page corresponding to a leaf node is full, the node is split into several nodes and all tuples in the disk page are redistributed among these child nodes. Different from the Quadtree where the region of each internal node is split into four non-overlapping sub-regions, the region of each internal node in the OQ-tree is split into at most nine overlapping sub-regions with region number from 1 to 9 as shown in Figure 5.2(b). However, not every internal node is split into nine child nodes that corresponds to the nine split sub-regions, since these overlapping sub-regions would bring duplicate nodes if every internal node were split into nine child nodes.

Consider the example data partition for the keyword concept "*Singapore Restaurant*" as shown in Figure 5.2(a). The partition can be organized as an OQ-tree as shown in Figure 5.2(c), where we assume that each disk page can store at most 3 tuples.

In an OQ-tree, the root node N_{root} (at level 0) corresponds to the whole region space $\mathbb{R} = \{[X_{min}, X_{max}], [Y_{min}, Y_{max}]\}$, and all nodes at level ℓ, $\ell > 0$, correspond to the same region size whose width and length are $X_\ell = 2^{-\ell}(X_{max} - X_{min})$, and $Y_\ell = 2^{-\ell}(Y_{max} - Y_{min})$, respectively. For a node N at level ℓ, $\ell \geq 0$, its region $N.R$ is specified by $\{[X_{min} + \frac{i}{2}X_\ell, X_{min} + \frac{i+1}{2}X_\ell], [Y_{min} + \frac{j}{2}Y_\ell, Y_{min} + \frac{j+1}{2}Y_\ell]\}$, $i, j \in \{0, 1, \cdots, 2^{\ell+1} - 1\}$. To avoid occurring duplicate nodes that correspond to those overlapping regions, each node N in an OQ-tree are classified into four different types depending on the i and j values of $N.R$ as follows:

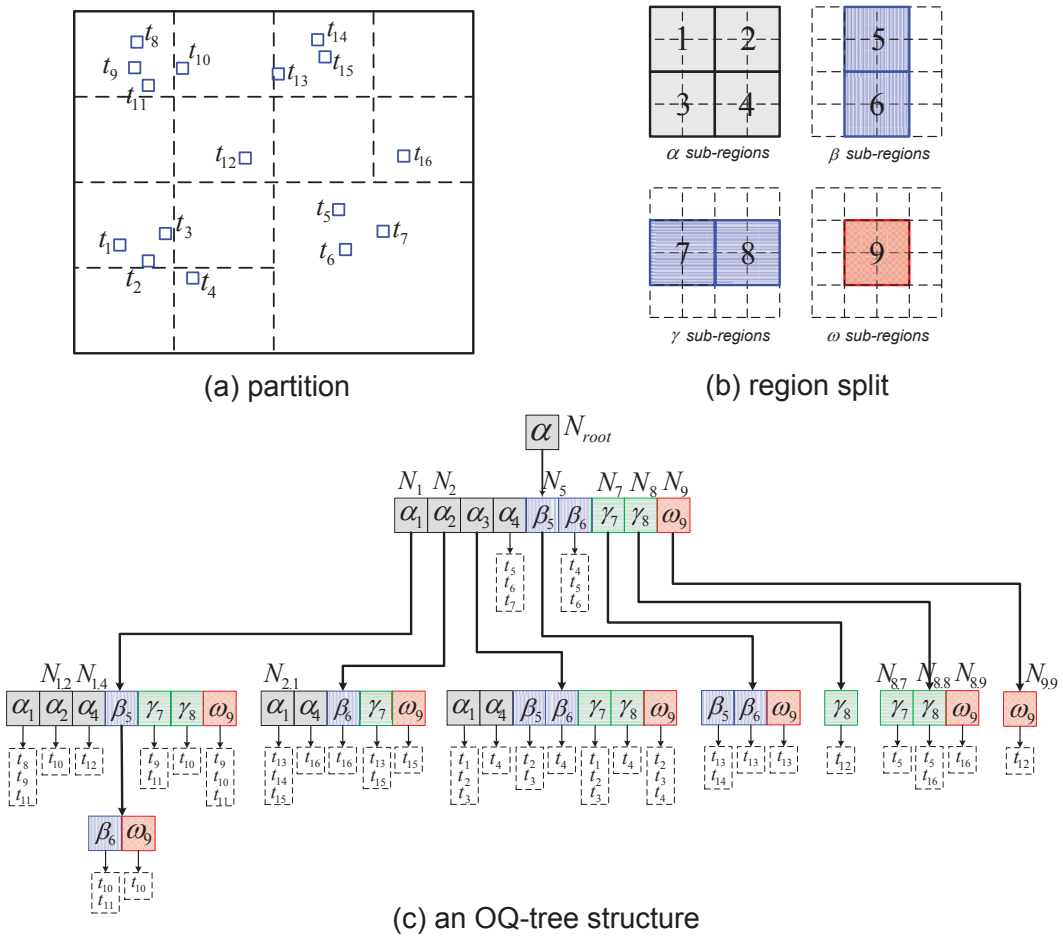


Figure 5.2: Example OQ-tree for the keyword concept "*Singapore Restaurant*"

- N is an α-node if both of i and j are even;
- N is a β-node if i is odd but j is even;
- N is a γ-node if i is even but j is odd;
- N is a ω-node if both of i and j are odd.

The root node N_{root} at level 0 is an α-node. The number of child nodes for an internal node N varies from 1 to 9 depending on the node type. Each child node corresponds to one of the nine sub-regions shown in Figure 5.2(b). More specifically, an internal α-node is split into nine child nodes that correspond to all of the nine sub-regions. Four of them are α-nodes corresponding to sub-regions with region IDs from 1 to 4; two of them are β-nodes corresponding to sub-regions with region IDs 5 and 6; two of them are γ-nodes corresponding to sub-regions with region IDs 7 and 8; and one of them is ω-node

corresponding to the sub-region with region ID 9. For example, the root node N_{root} in the OQ-tree shown in Figure 5.2(c) is split into nine child nodes that contains four α-nodes, two β-nodes, two γ-nodes and one ω-node.

An internal β-node is split into three child nodes that correspond to only three of the nine sub-regions. Two of child nodes are β-nodes corresponding to sub-regions with region IDs 5 and 6; one of them is ω-node corresponding to the sub-region with region ID 9. For example, the β-node N_5 in the OQ-tree is split into three child nodes including two β-nodes and one ω node.

An internal γ-node is split into three child nodes that correspond to only three of the nine sub-regions. Two of them are γ-nodes corresponding to sub-regions with region IDs 7 and 8; one of them is ω-node corresponding to the sub-region with region ID 9. For example, the γ-node N_8 in the OQ-tree is split into three child nodes with two γ-nodes and one ω node.

An internal ω-node only has one child node that corresponds to the sub-region with region ID 9. For example, the ω-node N_9 in the OQ-tree has only one child node $N_{9.9}$. Note that if the sub-region corresponding to a child node N is empty (it does not contain any tuple), then N is not physically created. For example, in the OQ-tree shown in Figure 5.2(c), the α-node N_2 only has five child nodes since the other four child nodes correspond to empty sub-regions.

For convenience, each node at level ℓ, $\ell > 0$, is represented as N_{id}, where id is the ID of this node. For each node at level 1, its ID is set as the region ID of the corresponding region. For example, the first child node of N_{root} in the example OQ-tree is represented as N_1. For a node at level ℓ, $\ell > 0$, its ID is specified by "$z_1.z_2.\cdots.z_\ell$" which is constructed by appending the region ID of its corresponding region "z_ℓ" to its parent node's ID "$z_1.z_2.\cdots.z_{\ell-1}$". For example, the three child nodes of node N_8 shown in Figure 5.2(c) are represented as $N_{8.7}$, $N_{8.8}$ and $N_{8.9}$, respectively.

In an OQ-tree, let $S(N.R)$ denote the set of tuples within the region $N.R$ of node N. Each leaf node N in an OQ-tree is of the form (id, R, P, s) , where id is the node id, R is the corresponding region, P refers to the corresponding disk page that stores all tuples in $S(N.R)$, and s is some summary information (to be explained later) which is used to speed up query processing.

Each internal α-node N is of the form (id, R, *children-pointers*, s) , where *children-pointers* are the pointers pointing to its child nodes. Based on these *children-pointers*, the tuple set $S(N.R)$ can be retrieved by recursively traversing all descendant α leaf nodes of N, since the region $N.R$ is equal to the union of the regions of these descendant α leaf nodes.

Each internal non-α-node N is of the form (id, R, *children-pointers*, *referred-pointers*, s) , where *referred-pointers* are the pointers pointing to the four lowest α nodes whose regions enclose the split sub-regions with region IDs from 1 to 4. of neighboring α-nodes that cover the four partitioned subregions with ID from 1 to 4. Note that some of these pointers could be empty. The intuition of including these *referred-pointers* is to facilitate the retrieval of $S(N.R)$, since $N.R$ is not equal to the union of the regions of its child nodes. For example, consider a α node N_5 in Figure 5.2(c). There are three referred nodes $N_{1.2}$, $N_{1.4}$ and $N_{2.1}$, and there is no referred node that covers the empty sub-region with region ID 4.Thus, the set $S(N.R)$ can be retrieved from $S(N_{1.2}.R) \cup S(N_{1.4}.R) \cup S(N_{2.1}.R)$.

Consider a simple DSQ query Q with only one keyword concept c and query radius r $r \in (0, \frac{L_{\mathbb{R}}}{2}]$, where $L_{\mathbb{R}}$ denotes the breadth of the whole region $\mathbb{R}$. Let T_c denote the OQ-tree corresponding to c. We are able to answer Q by only considering the r-related nodes in T_c defined as follows:

Definition 5.2 (r-related node)**.** *A r-related node in an index T_c is defined to be a node at level ℓ_r, $\ell_r = \lfloor \lg \frac{L_{\mathbb{R}}}{4r} \rfloor$, or a leaf node at a higher level ℓ'_r, $\ell'_r < \ell_r$.*

For example, given a DSQ query with keyword concept c :"*Singapore Restaurant*" and radius $r = \frac{L_{\mathbb{R}}}{16}$, the set of r-related nodes in T_c (shown in Figure 5.2(c)) contains all nodes at level 2 and the two leaf nodes (N_4 and N_6) at level 1.

The following result states the sufficient condition of utilizing r-related nodes to evaluate a DSQ or N-DSQ query, rather than considering combinations of neighboring nodes.

Lemma 5.1. *Consider a radius r, a query region R_Q and an OQ-tree T. For any circle of radius r within R_Q, there exists a r-related node in T that contains that circle.*

For example, consider the DSQ query with radius $r = 0.1 \cdot L_{\mathbb{R}}$, we only need to focus on the nine r-related nodes from N_1 to N_9 as shown in Figure 5.2(a).

5.4.2 Summary Information in Nodes

In this section, we present the summary information maintained in each node. The objective of the summary information is to prune the search space of a query evaluation by avoiding the enumeration of too many r-related nodes.

Consider the corresponding OQ-tree of keyword concept c. In each node N, a naive design is to maintain the upper bound score $\xi_c(S(N.R))$. Comparing with the top-1 group $G \subseteq S(N.R)$, the upper bound score $\xi_c(S(N.R))$ could be very "loose", since the circle of radius r is much smaller than the region of a r-related node. For example, consider a simple DSQ query with keyword concept c : "*Singapore Restaurant*" and radius $r = 0.1 \cdot L_{\mathbb{R}}$, and a r-related node N_5 (shown in Figure 5.4(a)) which contains five tuples (t_{10}, t_{12}, t_{13}, t_{14} and t_{15}). For convenience, the coverage score $cov_{c'}(o)$ is simply computed as 1 if $c' \in o.\psi$. The group G within the circle contains three tuples (t_{13}, t_{14} and t_{15}), and we have $\xi_c(G) = 0.399$, which is much smaller than the naive upper bound $\xi_c(S(N.R)) = 0.556$.

To provide a tighter bound for a given radius r, we maintain a series of κ upper bound aggregation scores that are suitable for different circle radii. Figure 5.3 shows the summary information maintained in each node N of an OQ-tree T_c. For node N, we maintain κ scores, denoted as $N.\varepsilon_1$, $N.\varepsilon_2$, $\cdots$, $N.\varepsilon_\kappa$, and also maintain some statistical information to facilitate the incremental update of these scores.

Scores			
$N.\varepsilon_1$	$N.\varepsilon_2$	$\cdots$	$N.\varepsilon_\kappa$

(a) maintained scores

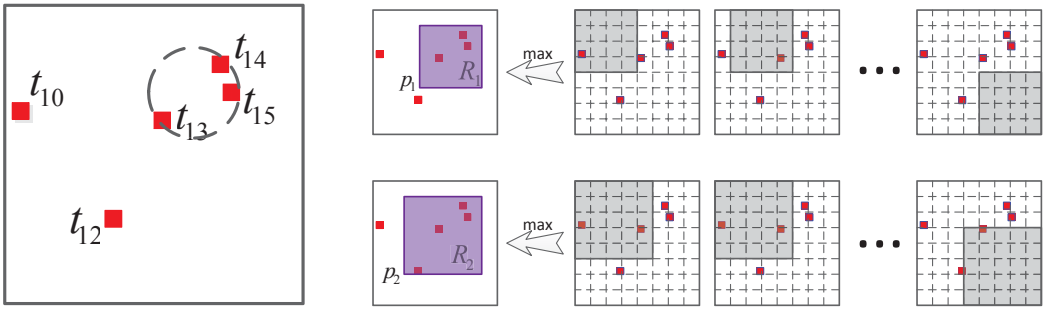
Statistical Information
$\langle\, p_1, \zeta_1^1, \cdots, \zeta_1^{\lvert\zeta_c\rvert} \,\rangle$
$\langle\, p_2, \zeta_2^1, \cdots, \zeta_2^{\lvert\zeta_c\rvert} \,\rangle$
$\cdots$
$\langle\, p_\kappa, \zeta_\kappa^1, \cdots, \zeta_\kappa^{\lvert\zeta_c\rvert} \,\rangle$

(b) maintained statistical information

Figure 5.3: Maintained summary information in a node N

More specifically, after partitioning the region $N.R$ into $2^\tau \times 2^\tau$, $\tau > 2$, grids, we maintain a series of κ, $\kappa = 2^{\tau-2}$, upper bound aggregation scores, where the i-th, $i \in [1, \kappa]$, score $N.\varepsilon_i$ represents the maximum aggregation score of tuples that are located within any one region covering $(\kappa + i + 1) \times (\kappa + i + 1)$ grids. For convenience, let R_i be such a region covering $(\kappa + i + 1) \times (\kappa + i + 1)$ grids where the set of tuples is with the maximum aggregation score. That is, we have $N.\varepsilon_i = \xi_c(S(R_i))$, where $S(R_i)$ is the set of tuples located within R_i. Note that the advantage of the grid partitioning is to efficiently calculate and update these scores based on the quadtree-like structure. For example, assuming that $\tau = 2$, we maintain 2 scores $N_5.\varepsilon_1$ and $N_5.\varepsilon_2$ for node N_5. As can be seen in Figure 5.4(b), after partitioning region $N_5.R$ into 8×8 grids, the first score $N_5.\varepsilon_1$ denotes the maximum aggregation score of tuples within any one region covering 4×4 grids, while the second score $N_5.\varepsilon_2$ represents the maximum aggregation score of tuples within any one region covering 5×5 grids.

To support the incremental update of score $N.\varepsilon_i$, $i \in [1, \kappa]$, we record some statistical information $\langle\, p_i, \zeta_i^1, \cdots, \zeta_i^{|\zeta_c|} \,\rangle$, where p_i represents the left bottom corner point of R_i, and ζ_i^j, $j \in [1, |\zeta_c|]$, denotes the sum of coverage scores for those objects which are located in R_i. That is, $\zeta_i^j = \sum_{o \in S(R_i)} cov_{c'_j}(o)$. For example, reconsider node N_5 and



(a) node N_5 (b) a series of scores for N_5

Figure 5.4: Maintained scores for node N_5

the two maintained scores shown in Figure 5.4. To support the incremental update, the maintained statistical information for the two maintained scores is shown in Table 5.3. Based on the weights of these relevant sub-concepts as shown in Table 5.2, score $N_5.\varepsilon_1$ can be calculated as $0.25 \cdot \ln(1+1) + 0.175 \cdot \ln(1+1) + 0.15 \cdot \ln(1+1) = 0.399$. In the same way, we can also calculate score $N_5.\varepsilon_2 = 0.482$.

	Covered Objects	Left-bottom Corner	Chinese Food	Japanese Food	Indian Food	Western Food	Korean Food	Malay Food
R_1	t_{13}, t_{14}, t_{15}	p_1	1	0	0	1	1	0
R_2	o_{12}, o_{13}, t_{14} t_{15}	p_2	1	1	0	1	1	0

Table 5.3: Example of statistical information in node N_5

After introducing the series of maintained scores, let us now discuss how to find the suitable score for a given radius r. For convenience, we use L_ℓ to denote the breadth of region $N.R$ for a node N at level ℓ. The following result states the sufficient condition of using the i-th score of a r-related node as the upper bound score.

Lemma 5.2. *Consider a DSQ query $Q(\{c\}, r, R_Q, k)$ and a r-related node N at level ℓ in T_c. From the set of tuples $S(N.R)$, let G be the candidate result group with the maximum score. If $r \leq \frac{L_\ell}{8} + \frac{i}{8\kappa}$, we have $N.\varepsilon_i \geq \xi_c(G)$.*

Among the series of maintained scores for a r-related node, let us further define the r-score as follows:

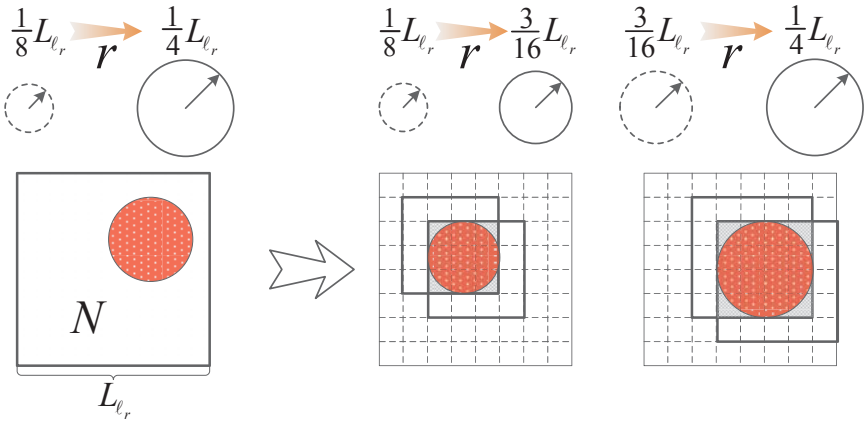


Figure 5.5: Mapping r into the r-score of N

Definition 5.3 (r-score of a r-related node)**.** *For a r-related node N, the r-score $N.\varepsilon(r)$ is defined to be the tightest score for the given radius r. If N is at level ℓ_r, $\ell_r = \lfloor \lg \frac{L_{\mathbb{R}}}{4r} \rfloor$, the r-score $N.\varepsilon(r)$ will be $N.\varepsilon_i$, where $r \in (\frac{L_{\ell_r}}{8} + \frac{i-1}{8\kappa}, \frac{L_{\ell_r}}{8} + \frac{i}{8\kappa}]$. Otherwise, N is a leaf node at level ℓ'_r, $\ell'_r < \ell_r$, and the r-score $N.\varepsilon(r)$ will be $N.\varepsilon_1$.*

Consider a r-related node N at level ℓ_r associated with the two maintained scores $N.\varepsilon_1$ and $N.\varepsilon_2$ as shown in Figure 5.5. If $r \in (\frac{1}{8}L_{\ell_r}, \frac{3}{16}L_{\ell_r}]$, the r-score $N.\varepsilon(r)$ will be $N.\varepsilon_1$. If $r \in (\frac{3}{16}L_{\ell_r}, \frac{1}{4}L_{\ell_r}]$, we have $N.\varepsilon(r) = N.\varepsilon_2$.

For example, reconsider the previous DSQ query with radius $r = 0.1 \cdot L_{\mathbb{R}}$ and node N_5. Since $r \in (\frac{1}{8}L_{\ell_r}, \frac{3}{16}L_{\ell_r}]$, the r-score of N_5 is equal to $N_5.\varepsilon_1 = 0.399$. Comparing with the naive score $\xi_c(N_5)$, the r-score is much tighter.

5.4.3 OQ-tree Vairants

Based on the OQ-tree, the query evaluation needs to enumerate and evaluate the r-related nodes with high r-scores. Therefore, to efficiently support the query evaluation, let us now present two different variants of OQ-tree: the OQ$^+$-tree index and the OQ*-tree index.

Besides the series of maintained scores mentioned in Section 5.4.2, the OQ$^+$-tree index maintains some additional information in each node to improve the search space pruning

of the query evaluation. In an OQ$^+$-tree index, each node N additionally maintains a hierarchical series of upper bound scores for each low level of the subtree rooted at N. The OQ$^+$-tree index is able to efficiently support the query evaluation in a top-down manner. When searching a node N with multiple r-related descendant nodes, the query evaluation can directly utilize these maintained hierarchical scores in N to determine whether there exist r-related descendant nodes with higher r-scores. If there does not exist, the query evaluation can efficiently filter out the enumerations of all r-related descendants of N.

On the other hand, instead of maintaining the additional information in each node, the OQ*-tree index additionally maintains several sorted node reference lists for each level of the OQ-tree. Based on these sorted lists, the OQ*-tree index can efficiently support the query evaluation by directly searching those r-related nodes in descending order of their r-scores.

The OQ$^+$-tree Index

We now introduce the first variant index (OQ$^+$-tree index). In an OQ$^+$-tree, node N maintains both the basic summary information (shown in Figure 5.3) as well as the additional hierarchical scores (shown in Figure 5.6(b)). Consider a node N at level ℓ with descendant nodes at level ℓ', $\ell' \in (\ell, L]$, where L is the lowest level of the subtree rooted at N. For each level ℓ' of its subtree, node N maintains a hierarchical series of scores, denoted as $\{N.\varepsilon_1^{\ell'}, \cdots, N.\varepsilon_\kappa^{\ell'}\}$, as shown in Figure 5.6(b). For convenience, each maintained score $N.\varepsilon_i$, $i \in [1, \kappa]$, can also be denoted as $N.\varepsilon_i^{\ell}$. For example, reconsider node N_5 in Figure 5.4(a). The score $N_5.\varepsilon_1$ can also be denoted as $N_5.\varepsilon_1^1$, since N_5 is at level 1.

To efficiently support the hierarchical filtering, each maintained hierarchical score $N.\varepsilon_i^{\ell'}$, $\ell' \in (\ell, L]$, is the maximum value among the i-th score maintained in any descendant node at level ℓ' and the 1-st score maintained in any descendant leaf node at a high level ℓ'', $\ell'' \in (\ell, \ell')$.

Scores			
$N.\varepsilon_1$	$N.\varepsilon_2$	$\cdots$	$N.\varepsilon_\kappa$

(a) scores

Subtree level	Hierarchical scores			
ℓ	$N.\varepsilon_1^\ell$	$N.\varepsilon_2^\ell$	$\cdots$	$N.\varepsilon_\kappa^\ell$
$\ell+1$	$N.\varepsilon_1^{\ell+1}$	$N.\varepsilon_2^{\ell+1}$	$\cdots$	$N.\varepsilon_\kappa^{\ell+1}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
L	$N.\varepsilon_1^L$	$N.\varepsilon_2^L$	$\cdots$	$N.\varepsilon_\kappa^L$

(b) hierarchical scores

Figure 5.6: Maintained scores in a node N at level ℓ

For example, reconsider node N_5 with three child nodes $N_{5.5}$, $N_{5.6}$ and $N_{5.9}$ as shown in Figure 5.2(c). For child node $N_{5.5}$, we can estimate the two upper bound scores $N_{5.5}.\varepsilon_1$ and $N_{5.5}.\varepsilon_2$ as 0.173 and 0.295, respectively. For child node $N_{5.6}$, we can estimate the two scores $N_{5.6}.\varepsilon_1$ and $N_{5.6}.\varepsilon_2$ as 0.087 and 0.087, respectively. For another child node $N_{5.9}$, the two scores $N_{5.9}.\varepsilon_1$ and $N_{5.9}.\varepsilon_2$ can be estimated as 0.173 and 0.173, respectively. The hierarchical score $N_5.\varepsilon_1^2$ can be set as $\max(N_{5.5}.\varepsilon_1^2, N_{5.6}.\varepsilon_1^2, N_{5.9}.\varepsilon_1^2) = \max(0.173, 0.087, 0.173) = 0.173$, and $N_5.\varepsilon_2^2$ can be set as $\max(N_{5.5}.\varepsilon_2^2, N_{5.6}.\varepsilon_2^2, N_{5.9}.\varepsilon_2^2) = \max(0.295, 0.087, 0.173) = 0.295$.

More generally, we define the r-score of a node in OQ$^+$-tree with some r-related descendant nodes to facilitate the hierarchical filtering of the query evaluation.

Definition 5.4 (r-score of a node)**.** *Consider a node N at level ℓ and a given radius $r \in (0, \frac{L_\ell}{4}]$. It is known that node N has some r-related descendant nodes, and the r-score of node N, denoted as $N.\varepsilon(r)$, is generally defined as the maximum score among the r-scores of these r-related descendant nodes of N. If $r \in (\frac{L_{\ell'}}{8} + \frac{i-1}{8\kappa}, \frac{L_{\ell'}}{8} + \frac{i}{8\kappa}]$, $\ell' \in [\ell, L]$ $i \in [1, \kappa]$, the r-score of N $N.\varepsilon(r)$ will be $N.\varepsilon_i^{\ell'}$. If $r < \frac{L_L}{8}$, we have $N.\varepsilon(r)$ is $N.\varepsilon_1^L$.*

For example, reconsider node N_5 (shown in Figure 5.2(c)) and a simple DSQ query with keyword concept c : "*Singapore Restaurant*" and radius $r = \frac{L_\mathbb{R}}{16}$. The r-score of N_5 equals to $N_5.\varepsilon_2^2$.

The OQ*-tree Index

Instead of hierarchically filtering those r-related nodes with low r-scores, we introduce the second variant index (OQ*-tree index) to support the direct access of r-related nodes in descending order of their r-scores.

Besides the tree structure of OQ-tree, the OQ*-tree index additionally maintains several sorted node reference lists for each level of the OQ-tree. More specifically, for nodes at level ℓ, the OQ*-tree maintains $\kappa+1$ sorted lists, denoted as $list_1^\ell, \cdots, list_\kappa^\ell, llist_1^\ell$. Each entry of these lists is with the same format $\langle ref(N), N.\varepsilon\rangle$, where $ref(N)$ is the node reference of node N. The sorted list $list_i^\ell$, $i \in [1,\kappa]$, maintains all entries $\langle ref(N), N.\varepsilon_i\rangle$ in descending order of $N.\varepsilon_i$, where N is any node at level ℓ. Differently, the list $llist_1^\ell$ only focuses on those leaf nodes at level ℓ, and maintains all entries $\langle ref(N), N.\varepsilon_1\rangle$ in descending order of $N.\varepsilon_1$, where N is any leaf node at level ℓ. Note that $llist_1^\ell$ could be empty if there is no leaf node at level ℓ.

Based on Definition 5.2, all r-related nodes are within either list of $list_i^{\ell_r}$, $llist_1^{\ell_r-1}$, $\cdots$, $llist_1^1$, where $\ell_r = \lfloor \lg \frac{L_\mathbb{R}}{4r} \rfloor$ and $r \in (\frac{L_{\ell_r}}{8} + \frac{i-1}{8\kappa}, \frac{L_{\ell_r}}{8} + \frac{i}{8\kappa}]$. By run-time merging these lists, we can generate a list of all r-related nodes in descending order of their r-scores. In the OQ*-tree corresponding to a keyword concept c, we denote the sorted list as $list_c(r)$.

5.4.4 Data Operation

Now we introduce how to build and maintain the index. More specifically, we explain three basic data operations on an IOQ-tree index, including data insertion, deletion and update. Furthermore, we also discuss the data operations on the two index variants.

Algorithm 5.1: `Insert` (N, t)

```
   Input: Node N of the OQ-tree subtree and the tuple t
1  if (N is a leaf node) and (N.P is full) then
2      N.childs ← childs by splitting N;
3      foreach tuple t' in N.P do
4          foreach child node N' ∈ N.childs, where t' is located do
5              if N'.P = null then
6                  N'.P ← allocate a new disk page;
7              Insert(N', t');
8      delete N.P;
9  if N is an internal node then
10     foreach child node N' ∈ N.childs, where t will be located do
11         Insert(N', t);
12 else
13     store t in N.P;
14 UpdateSeriesScores (N, t);
```

Algorithm 5.2: `UpdateSeriesScores` (N, t)

```
   Input: Node N at level ℓ and the tuple t
1  for i ← 1 to κ do
2      if t is located within N.R_i then
3          update the sum of the coverage score of relevant sub-concepts by using
           tuple t;
4          calculate N.ε_i^ℓ;
5      else
6          identify the optimal region covering (κ + i + 1) × (κ + i + 1) partitioned
           grids;
7          calculate N.ε_i^ℓ;
```

Data Insertion

We first present the insertion operation on an OQ-tree, following by the discussion on the two index variants (OQ$^+$-tree and OQ*-tree). Algorithm 5.1 shows the psecudocode of recursively inserting a tuple t into the subtree of the OQ-tree rooted at node N. Therefore, we can insert t into the OQ-tree by evoking `Insert`(N_{root}, t).

In the function `Insert`, we first check whether it is necessary to split node N (line 1). If N is a leaf node, we have to split N if the page is full (line 2-8). All tuples in the full page will be reinserted into these new split child nodes of N. Subsequently, we then inserts the

Algorithm 5.3: `UpdateHierarchicalScores` (N)

Input: Node N at level ℓ

1 $L \leftarrow$ the lowest level of the subtree rooted at N;
2 **for** $\ell' \leftarrow \ell + 1$ ***to*** L **do**
3 **foreach** *child node* $N' \in N.childs$*, where* t *is located* **do**
4 $L' \leftarrow$ the lowest level of the subtree rooted at N';
5 **for** $i \leftarrow 1$ ***to*** κ **do**
6 **if** $\ell' \leq L'$ **then** $N.\varepsilon_i^{\ell'} \leftarrow \max(N.\varepsilon_i^{\ell'}, N'.\varepsilon_i^{\ell'})$;
7 **else** $N.\varepsilon_i^{\ell'} \leftarrow \max(N.\varepsilon_i^{\ell'}, N'.\varepsilon_1^{L'})$;

tuple t into the current node N (line 9-13). If N is an internal node, we recursively insert tuple t into each of those child nodes, whose corresponding regions cover t (line 9-11). Otherwise, we store tuple t in the disk page $N.P$ if N is a leaf node (line 13). Finally, we update the hierarchical series of scores with the inserted tuple t by evoking the function `UpdateSeriesScores` shown in Algorithm 5.2.

In the function `UpdateSeriesScores` as shown in Algorithm 5.2, we update the series of scores for node N at level ℓ. For each score $N.\varepsilon_i^{\ell}$, $i \in [1, \kappa]$, we will check whether tuple t is located in the recorded optimal region covering $(\kappa+i+1) \times (\kappa+i+1)$ grids. If it is, we can directly update the sum of coverage scores of relevant subconcepts, followed by calculating the score based on the Equation 5.2 (line 2-4). Otherwise, we identify the optimal region covering $(\kappa+i+1) \times (\kappa+i+1)$ grids, and calculate the score for the set of tuples covered by the optimal region (line 6-7).

The time complexity of inserting a tuple into an OQ-tree by applying the function `Insert` is $\mathcal{O}(3H)$, where H is the height of the OQ-tree.

Subsequently, we discuss the insertion operation on the OQ$^+$-tree index. A tuple can be inserted into an OQ$^+$-tree index, by applying an extension insertion function of `Insert` (shown in Algorithm 5.1). To update the hierarchical scores maintained in node N, we can extend the function `Insert` by evoking the function `UpdateHierarchicalScores` (shown in Algorithm 5.3) at the end. In the function `UpdateHierarchicalScores`,

each hierarchical score can be updated based on those hierarchical scores maintained in the child nodes of N. The time complexity of the insertion on an OQ$^+$-tree is also $\mathcal{O}(3H)$.

Finally, we now present the insertion operation on the OQ*-tree index. We first apply the function `Insert` (shown in Algorithm 5.1) to update these maintained scores in node N. After that, for each updated score $N.\varepsilon_i$, $i \in [1, \kappa]$, we need to update the reference entry $\langle ref(N), N.\varepsilon_i \rangle$ in the sorted list $list_i^\ell$. Therefore, the time complexity of the insertion on an OQ*-tree will be $\mathcal{O}(3H + 3(\kappa + 1)H)$.

Data Deletion and Update

After discussing the insertion operation, the deletion is much easier to be understood. Consider a tuple t which needs to be deleted. We can traverse the corresponding OQ-tree downward started from the root. After identifying the disk page where t is stored, we can directly remove t from the page, and the page will be dropped if it becomes empty. Furthermore, we will update the series of scores maintained in each node N where t is located. For the score $N.\varepsilon_i^\ell$ where ℓ is the level of N and $i \in [1, \kappa]$, we do not need to verify if tuple t is not located in the recorded optimal region covering $(\kappa+i+1)\times(\kappa+i+1)$ partitioned grids. Otherwise, we have to identify the new optimal region, and update the score.

In an OQ$^+$-tree index, the update of the maintained scores in N will propagate upwards the update of the hierarchical series of scores for each ancestor node of N. In an OQ*-tree index, the update of each maintained score will also affect the sorted lists as mentioned before.

Overall, the time complexity of the deletion operation is the same with the insertion operation.

An update operation is treated as a deletion followed by an insertion because its location information or keyword information could be changed and the tuple belongs to another node.

5.5 Evaluation of DSQ queries

In this section, we present our approach for evaluating a DSQ query $Q(\psi, r, R_Q, k)$ with an IOQ-tree index I. Let us assume that $\psi = \{c_1, \cdots, c_n\}$, and for keyword concept $c_i \in \psi$, we use T_{c_1} to denote the OQ-tree corresponding to c_i. To evaluate query Q, the query evaluation needs to traverse these corresponding OQ-trees $T_{c_1}, \cdots, T_{c_n}$ in parallel. To distinguish nodes in different OQ-trees, we use $N_{id}^{c_i}$, $c_i \in \psi$, to represent a node in T_{c_i}.

As mentioned in Section 5.4, all of these OQ-trees are based on the uniform space decomposition mechanism. For a node N^{c_i} in an OQ-tree T_{c_i}, there always exists such a node N^{c_j} in another OQ-tree T_{c_j}, $c_i \neq c_j$, such that $N^{c_i}.R$ is the same with $N^{c_j}.R$ or $N^{c_j}.R$ contains $N^{c_i}.R$ if N^{c_j} is a leaf node. To facilitate the parallel traversal, let us first introduce a parallel state η with the format $\langle R, \{N^{c_1}, \cdots, N^{c_n}\}\rangle$, where R is a partitioned region and $N^{c_i}.R$, $c_i \in \psi$, represents the lowest node in T_{c_i} covering region R. For two parallel states $\eta : \langle R, \{N^{c_1}, \cdots, N^{c_n}\}\rangle$ and $\eta_u : \langle R_u, \{N_u^{c_1}, \cdots, N_u^{c_n}\}\rangle$, we say that η_u is a descendant parallel state of η if $R \supset R_u$. Additionally, we say η is a r-related parallel state if each node N^{c_i}, $c_i \in \psi$, is a r-related node in T_{c_i}.

Based on Lemma 5.1, we have the following sufficient condition for evaluating a spatial query by only considering each r-related parallel states.

Lemma 5.3. *Consider a DSQ query $Q(\psi, r, R_Q, k)$ and $\psi = \{c_1, \cdots, c_n\}$. For any candidate result group G, there exists a r-related parallel state $\eta : \langle R, \{N^{c_1}, \cdots, N^{c_n}\}\rangle$ such that $G \subseteq \cup_{c_i \in \psi} S(N^{c_i}.R)$.*

The key idea of the query evaluation is to maintain and refine the top-k result groups $\{G_1, \cdots, G_k\}$ by enumerating these r-related parallel states. Rather than simply enumerating all r-related parallel states, we would like to reduce the search space of query evaluation by filtering out as many r-related parallel states as possible. Based on the two OQ-tree variants, we propose two efficient evaluation methods: OQ$^+$-tree and OQ*-tree.

For a parallel state $\eta : \langle R, \{N^{c_1}, \cdots, N^{c_n}\}\rangle$, we maintain an upper bound score $F(\eta)$ for any result group G, $G \subseteq \cup_{c_i \in \psi} S(N^{c_i}.R)$. A simple upper bound score $F(\eta)$ can be estimated as $\frac{1}{n}\sum_{c_i \in \psi} F^{\eta}(N^{c_i})$, where $F^{\eta}(N^{c_i})$ is set as $N^{c_i}.\varepsilon(r)$. Additionally, in the later section, we show that $F(\eta)$ can be further restricted by refining $F^{\eta}(N^{c_i})$ after retrieving tuples that are within $N^{c_i}.R$.

5.5.1 OQ$^+$-tree Evaluation

Let us first introduce the OQ$^+$-tree evaluation, which traverses these corresponding OQ$^+$-tree indexes T_{c_1}, $\cdots$, T_{c_n} in a top-down, best-first manner. For each r-related parallel state $\eta : \langle R, \{N^{c_1}, \cdots, N^{c_n}\}\rangle$, the query evaluation retrieves tuples within these r-related nodes $\{N^{c_1}, \cdots, N^{c_n}\}$ if $F(\eta) \geq \xi_{\psi}(G_k)$, and then finds highly ranked candidate result groups from these retrieved tuples to refine the current maintained top-k result groups $\{G_1, \cdots, G_k\}$. On the other hand, for each parallel state $\eta : \langle R, \{N^{c_1}, \cdots, N^{c_n}\}\rangle$ with some descendant r-related parallel states, we search down these subtrees $T_{c_1}(N^{c_1})$, $\cdots$, $T_{c_n}(N^{c_n})\}$ in parallel, in the case that $F(\eta) \geq \xi_{\psi}(G_k)$.

Algorithm 5.4 shows the pseudo-code of the OQ$^+$-tree evaluation method. To efficiently support the best-first parallel traversal, a priority queue PQ is designed to maintain the set of parallel states, whose upper bound scores are greater than $\xi_{\psi}(G_k)$. In the priority queue PQ, the top one will always be with the highest upper bound score. After initializing the priority queue PQ and each maintained group G_i, $i \in [1, k]$, as $\emptyset$, we start the query evaluation by pushing the initial parallel state $\langle \mathbb{R}, \{N^{c_1}_{root}, \cdots, N^{c_n}_{root}\}\rangle$, where $\mathbb{R}$ is the uniform

Algorithm 5.4: `OQ`$^+$`-tree-Eval` (Q, I)

Input: the DSQ query $Q(\psi, r, R_Q, k)$ with the keyword concepts $\psi = \{c_1, \cdots, c_n\}$, and the IOQ-tree index I
Output: the top-k result groups $\{G_1, \cdots, G_k\}$

```
1  PQ ← ∅, {G_1, ··· , G_k} ← {∅, ··· , ∅};
2  push ⟨ℝ, {N^{c_1}_root, ··· , N^{c_n}_root}⟩ into PQ;
3  while PQ is not empty do
4      pop the top state η : ⟨R, {N^{c_1}, ··· , N^{c_n}}⟩ from PQ;
5      if F(η) < ξ_ψ(G_k) then break ;
6      if η is not a r-related parallel state then
7          S_η ← GenerateChildStates(η, R_Q) ;
8          foreach child state η_u ∈ S_η do
9              push η_u into PQ if F(η_u) ≥ ξ(G_k);
10     else
11         if tuples for all nodes of η have been retrieved then
12             S ← ⋃_{i∈[1,n]} S_i^η(R, R_Q);
13             foreach group G that can be generated from S do
14                 if ξ_ψ(G) > ξ_ψ(G_k) then
15                     refine {G_1, ··· , G_k} with G;
16         else
17             retrieve tuples of N_i that is the non-retrieved node of η at the highest
               level;
18             let S_i^η(R, R_Q) be the retrieved tuples of S(N^{c_i}.R) which are located
               within R and R_Q;
19             restrict F(η) by replacing F^η(N^{c_i}) with ξ_ψ(S_i^η(R, R_Q));
20             if S_i^η(R, R_Q) ≠ ∅ then
21                 re-push η into PQ;
22 return {G_1, G_2, ··· , G_k};
```

whole region, and $N^{c_i}_{root}$, $c_i \in \psi$, denotes the root node of T_{c_i} (line 1-2). Subsequently, we then iteratively pop and process the top parallel state $\eta : \langle R, \{N^{c_1}, \cdots, N^{c_i}, \cdots, N^{c_n}\}\rangle$ of PQ.

In the case that the top state η is not a r-related parallel state, a set of child parallel states can be generated by evoking the function `GenerateChildStates`(Algorithm 5.5). We then push those generated child states into PQ for further processing, if their upper bound scores are greater than $\xi_\psi(G_k)$ (line 8-9).

On the other hand, let us focus on the case that the top state η is r-related parallel state. For each node N^{c_i} in η, all tuples in $S(N^{c_i}.R)$ need to be retrieved. Based on these retrieved

Algorithm 5.5: `GenerateChildStates` (η, R_Q)

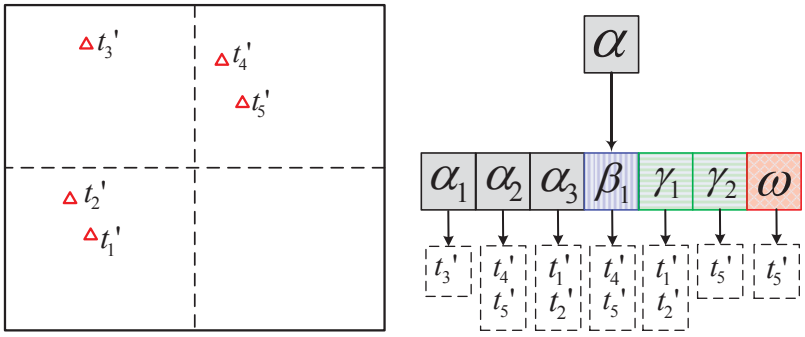
Input: the parallel state $\eta : \langle R, \{N^{c_1}, \cdots, N^{c_n}\}\rangle$ and the query region R_Q
Output: the set of child states S_η

```
1  let N^{c_i}, i ∈ [1, n], be the node at the lowest level among N^{c_1}, ..., N^{c_n} ;
2  split N^{c_i}.R into several sub-regions based on the type of node N^{c_i};
3  S_η ← ∅ ;
4  foreach splitted sub-region R_u do
5      if R_u ∩ R_Q = ∅ then continue ;
6      for node N^{c_i} in η do
7          if N^{c_i} is a leaf node then
8              N_u^{c_i} ← N^{c_i} ;
9          else
10             N_u^{c_i} ← the child node of N^{c_i} covering R_u ;
11     if every node N_u^{c_i} is not empty then
12         Generate the child parallel state η_u : ⟨R_u, {N_u^{c_1}, ..., N_u^{c_n}}⟩ ;
13         S_η ← S_η ∪ {η_u} ;
14 Return S_η ;
```

tuples, we can apply the circle-placement algorithm [23] to identify those candidate result groups whose ranking scores are greater than $\xi_c(G_k)$, followed by refining the current top-k result groups $\{G_1, \cdots, G_k\}$. One of the straight-forward methods is to directly retrieve tuples for all nodes of η at a time. As mentioned, nodes in η could be at different levels. For a node N^{c_i}, $N^{c_i}.R \supset R$ at a high level, we use $S_i^\eta(R, R_Q)$ to represent the set of tuples of N^{c_i} that are located within R and R_Q. Sometimes the set $S_i^\eta(R, R_Q)$ could be empty, and there will not exist a candidate result group in η based on Definition 5.1. The IO cost of retrieving tuples for nodes in η will be wasted. Instead, we prefer to first retrieve tuples of the non-retrieved node N^{c_i} at the highest level (line 17). After retrieving tuples in $S(N^{c_i}.R)$, we then restrict $F(\eta)$ by replacing $F^\eta(N^{c_i})$ with $\xi_\psi(S_i^\eta(R, R_Q))$ (line 19). If the restricted upper bound is no greater than $\xi_\psi(G_k)$, all the tuples in other non-retrieved nodes of η will not be retrieved. Otherwise, before immediately retrieving tuples in the non-retrieved node of η at the next highest level, we re-push η into PQ to guarantee that each time only the state with the highest upper bound score will be processed, since the upper bound score $F(\eta)$ after restriction could be lower than the upper bound score of the top state of the current priority queue excluding η (line 20-21). After restriction,

we can directly process to refine $\{G_1, \cdots, G_k\}$ if the tuples for all nodes in η have been retrieved (line 12-15). Finally, the query evaluation is terminated when there exist no more unprocessed parallel states whose upper bound scores are greater than $\xi_\psi(G_k)$.

The function `GenerateChildStates`(η, R_Q) (shown in Algorithm 5.5) generates the child parallel states of state $\eta : \langle R, \{N^{c_1}, \cdots, N^{c_n}\}\rangle$. Among these nodes in η, let N^{c_i} be the one at the lowest level. Based on the type of node N^{c_i}, we split the region $N^{c_i}.R$ into several different sub-regions. For each sub-region R_u, we try to generate a child parallel state $\eta_u : \langle R_u, \{N_u^{c_1}, \cdots, N_u^{c_n}\}\rangle$, where $N_u^{c_i}$ is the lowest node in T_{c_i} covering R_u.



(a) partition (b) an OQ-tree

Figure 5.7: Example OQ-tree for the keyword concept "*Entertainment Facility*"

Relevant Sub-concept	Movie Theater	KTV Club	Bar
Weights	0.3	0.3	0.4

Table 5.4: The weights of relevant sub-concepts of "*Entertainment Facility*"

Let us take an example to illustrate the OQ$^+$-tree evaluation method.

Example 5.1: Reconsider a DSQ query Q with the query keyword concepts c_1 : "*Singapore Restaurant*" and c_2 : "*Entertainment Facility*", a limit size 2, and the query radius $r = 0.1 \cdot L_{\mathbb{R}}$. Table 5.4 shows the weights of relevant sub-concepts of the keyword concept "*Entertainment Facility*". For the two keyword concepts, the corresponding partitions are shown in Figure 5.2(a) and Figure 5.7(a), respectively. Assuming the maximum tuple capacity of each disk page is 3, the corresponding OQ$^+$-trees T_{c_1} and T_{c_2} are shown in Figure 5.2(c) and Figure 5.7(b), respectively. Table 5.5 and Table 5.6

region	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
r-related node	$N_1^{c_1}$	$N_2^{c_1}$	$N_3^{c_1}$	$N_4^{c_1}$	$N_5^{c_1}$	$N_6^{c_1}$	$N_7^{c_1}$	$N_8^{c_1}$	$N_9^{c_1}$
r-score	0.282	0.399	0.537	0.433	0.399	0.295	0.433	0.504	0.295

Table 5.5: r-related nodes in T_{c_1}

region	R_1	R_2	R_3	R_5	R_7	R_8	R_9
r-related node	$N_1^{c_2}$	$N_2^{c_2}$	$N_3^{c_2}$	$N_5^{c_2}$	$N_7^{c_2}$	$N_8^{c_2}$	$N_9^{c_2}$
r-score	0.208	0.482	0.416	0.482	0.482	0.277	0.277

Table 5.6: r-related nodes in T_{c_2}

show the r-related nodes associated with their r-scores in T_{c_1} and T_{c_2}, respectively. The OQ$^+$-tree evaluation traverses down the two OQ$^+$-trees T_{c_1} and T_{c_2} in parallel. From the root state $\eta : \langle \mathbb{R}, \{N_{root}^{c_1}, N_{root}^{c_2}\}\rangle$, the evaluation generates 7 r-related parallel states $\eta_1 : \langle R_1, \{N_1^{c_1}, N_1^{c_2}\}\rangle$, $\eta_2 : \langle R_2, \{N_2^{c_1}, N_2^{c_2}\}\rangle$, $\eta_3 : \langle R_3, \{N_3^{c_1}, N_3^{c_2}\}\rangle$, $\eta_5 : \langle R_5, \{N_5^{c_1}, N_5^{c_2}\}\rangle$, $\eta_7 : \langle R_7, \{N_7^{c_1}, N_7^{c_2}\}\rangle$, $\eta_8 : \langle R_8, \{N_8^{c_1}, N_8^{c_2}\}\rangle$, and $\eta_9 : \langle R_9, \{N_9^{c_1}, N_9^{c_2}\}\rangle$. All of these states are pushed into the maintained priority queue.

The OQ$^+$-tree evaluation first pops and processes the top state η_3 with the upper bound $F(\eta_3)$ as 0.4765, and then obtains two candidate result groups G'_1 ($\xi_\psi(G'_1) = 0.458$), and G'_2 ($\xi_\psi(G'_2) = 0.312$) shown in Figure 1.3(b). After searching state η_7 with the upper bound score $F(\eta_7)$ as 0.4575, state η_5 with the upper bound score $F(\eta_5)$ as 0.440, state η_2 with the upper bound score $F(\eta_2)$ as 0.440 and state η_8 with the upper bound score $F(\eta_8)$ as 0.3905, the evaluation will be terminated since the upper bound scores of all other states are smaller than $\xi_\psi(G'_2)$. □

5.5.2 OQ*-tree Evaluation

Having discussed the OQ$^+$-tree evaluation based on the top-down traversals of these corresponding OQ$^+$-trees $T_{c_1}, \cdots, T_{c_n}$, let us now introduce the OQ*-tree evaluation to directly search the r-related parallel states constructed by accessing r-related nodes from the sorted lists $list_{c_1}(r), \cdots, list_{c_n}(r)$. In the process of constructing parallel states, some parallel states with unseen nodes can be generated. To distinguish them from those parallel states without unseen nodes, we say that a parallel state $\eta : \langle R, \{N^{c_1}, \cdots, N^{c_n}\}\rangle$ is

complete if every N^{c_i}, $c_i \in \psi$, is an accessed node. Otherwise, we say that η is a partial parallel state.

Consider a constructed r-related parallel state $\eta : \langle R, \{N^{c_1}, \cdots, N^{c_n}\}\rangle$. In the case that η is complete, the upper bound score $F(\eta)$ is initialized by setting $F^{\eta}(N^{c_n})$ as $N^{c_i}.\varepsilon(r)$. Otherwise, for an unseen node N^{c_i}, $F(\eta)$ is estimated by substituting $F^{\eta}(N^{c_i})$ with the r-score of the last accessed r-related node in $list_{c_i}(r)$. During query evaluation, we use a binary heap $BHeap$ to maintain all constructed (partial or complete) r-related parallel states in descending order of their upper bound scores.

The main algorithm of the OQ*-tree evaluation is shown in Algorithm 5.6. The evaluation is started by initializing the binary heap $BHeap$ and each maintained group G_i, $i \in [1, k]$, as $\emptyset$ (line 1). We then iteratively access and process the next r-related node N^{c_i} from $list_{c_i}(r)$ in parallel (line 2-17). Based on these newly accessed r-related nodes, we are able to construct some new parallel states, followed by pushing them into $BHeap$. Some existing partial parallel states could also be updated by joining with these newly accessed r-related nodes, and the upper bound score of each partial state in $BHeap$ is also updated. Consider a new r-related node N^{c_i} that has just been accessed from $list_{c_i}(r)$. For a partial state η without an accessed node in $list_{c_i}(r)$, the upper bound score $F(\eta)$ is refined by setting $F^{\eta}(N^{c_i})$ as $N^{c_i}.\varepsilon(r)$. After updating the upper bound scores for states in $BHeap$, the evaluation then iteratively pops and processes the top state η only if η is complete. For a top state $\eta : \langle R, \{N^{c_1}, \cdots, N^{c_n}\}\rangle$ that is complete, we first retrieve tuples for the node of η at the highest level, rather than retrieving all tuples in all of these nodes at a time. After retrieving tuples in the node at the highest level, we restrict the upper bound scores of η and other states in $BHeap$ by evoking the function `RestrictUpperBound` (Algorithm 5.7). If tuples for all nodes in the top state have been retrieved, we directly refine $\{G_1, \cdots, G_k\}$ (line 12). The evaluation is terminated if there exist no more unprocessed (partial or complete) r-related parallel states whose upper bound scores are greater than $\xi_{\psi}(G_k)$.

Algorithm 5.6: `OQ*-tree-Eval` (Q, I)

Input: the DSQ query $Q(\psi, r, R_Q, k)$ with the keyword concept set $\psi = \{c_1, \cdots, c_n\}$, and the IOQ-tree index I
Output: the top-k result groups $\{G_1, \cdots, G_k\}$

1 $BHeap \leftarrow \emptyset$, $\{G_1, \cdots, G_k\} \leftarrow \{\emptyset, \cdots, \emptyset\}$;
2 **while** *there exists at least one unaccessed r-related node in either inverted list $list_{c_i}(r)$, $c_i \in \psi$* **do**
3 access the next r-related node N^{c_i} ($N^{c_i}.R \cap R_Q \neq \emptyset$) from each list $list_{c_i}(r)$ in parallel;
4 construct r-related parallel states with newly accessed nodes, and push them into $BHeap$;
5 join spatial parallel states in $BHeap$ with these newly accessed nodes;
6 **foreach** *newly accessed node N^{c_i}* **do**
7 update the upper bound score for each partial state without an accessed node in $list_{c_i}(r)$;
8 **while** *the top state is a complete* **do**
9 pop the top state $\eta : \langle R, \{N^{c_1}, \cdots, N^{c_n}\}\rangle$ from $BHeap$;
10 **if** $F(\eta) \leq \xi_\psi(G_k)$ **then** break ;
11 **if** *tuples for all nodes of η have been retrieved* **then**
12 refine $\{G_1, \cdots, G_k\}$ following steps 12-15 in Algorithm 5.4;
13 **else**
14 `RestrictUpperBound`$(\eta, BHeap, R_Q)$;
15 $\eta \leftarrow$ the top state in $BHeap$;
16 **if** $F(\eta) \leq \xi_\psi(G_k)$ **then**
17 break;
18 return $\{G_1, \cdots, G_k\}$;

The function `RestrictUpperBound`$(\eta, BHeap, R_Q)$ (shown in Algorithm 5.7) restricts $F(\eta)$ by retrieving tuples from the non-retrieved node N^{c_i} at the highest level in η, and also restricts the upper bound scores for all parallel states in $BHeap$ containing N^{c_i}. For the parallel state $\eta_u : \langle R_u, \{\cdots, N^{c_i}, \cdots\}\rangle$, let $S_i^\eta(R_u, R_Q)$ denote the set of retrieved tuples from $S(N^{c_i}.R)$ which are located within R_Q. If $S_i^\eta(R_u, R_Q)$ is empty, we can directly discard η_u from $BHeap$ to avoid to spend extra IOs to retrieve tuples from other non-retrieved nodes in η_u. Otherwise, $F(\eta_u)$ is restricted by setting $F^{\eta_u}(N^{c_i})$ as $\xi_\psi(S_i^\eta(R_u, R_Q))$. Overall, the upper bound score restriction can benefit the OQ*-tree evaluation by improving the filter rating of these parallel states.

Algorithm 5.7: `RestrictUpperBound` $(\eta, BHeap, \psi, R_Q)$

Input: the complete r-related parallel state $\eta : \langle R, \{N^{c_1}, \cdots, N^{c_n}\}\rangle$, the binary heap $BHeap$, the query keyword concepts ψ and the query region R_Q

1 let N^{c_i} be the non-accessed node at the highest level among $\{N^{c_1}, \cdots, N^{c_n}\}$;
2 retrieve tuples which are located in N^{c_i};
3 let $S_i^{\eta}(R, R_Q)$ be the retrieved tuples $S(N^{c_i}.R)$ which are located within R and R_Q;
4 restrict $F(\eta)$ by replacing $F^{\eta}(N^{c_i})$ with $\xi_{\psi}(S_i^{\eta}(R, R_Q))$;
5 **if** $S_i^{\eta}(R, R_Q) \neq \emptyset$ **then**
6 | re-push η into $BHeap$;
7 **foreach** *each state* $\eta_u : \langle R_u, \{\cdots, N^{c_i}, \cdots\}\rangle$ **do**
8 | let $S_i^{\eta}(R_u, R_Q)$ denote the retrieved tuples which are located within the intersection area of R_u and R_Q;
9 | **if** $S_i^{\eta}(R_u, R_Q) = \emptyset$ **then**
10 | | remove η_u from $BHeap$;
11 | **else**
12 | | refine $F(\eta_u)$ by setting $F^{\eta_u}(C_i)$ as $\xi_{\psi}(S_i^{\eta}(R', R_Q))$;

Let us now take an example to illustrate the OQ*-tree evaluation method.

Example 5.2: Reconsider the DSQ query in Example 5.1. The OQ*-tree evaluation accesses the r-related nodes of T_{c_1} in the order of $N_3^{c_1}$, $N_8^{c_1}$, $N_4^{c_1}$, $N_7^{c_1}$, $N_5^{c_1}$, $N_2^{c_1}$, $N_6^{c_1}$, $N_9^{c_1}$, $N_1^{c_1}$, while the evaluation scan the r-related nodes of T_{c_2} in the order of $N_2^{c_2}$, $N_5^{c_2}$, $N_7^{c_2}$, $N_3^{c_2}$, $N_8^{c_2}$, $N_9^{c_2}$, $N_1^{c_2}$. Figure 5.8 shows the sequence of updating $BHeap$ by OQ*-tree evaluation.

- (a) The evaluation accesses $N_3^{c_1}$ and $N_2^{c_2}$, followed by generating two partial states $\eta_3 : \langle R_3, \{N_3^{c_1}, -\}\rangle$ and $\eta_2 : \langle R_2, \{-, N_2^{c_2}\}\rangle$ with the upper bound score 0.5095 as shown in Figure 5.8(a).

- (b) The evaluation scans $N_8^{c_1}$ and $N_5^{c_2}$, followed by generating two partial states $\eta_8 : \langle R_8, \{N_8^{c_1}, -\}\rangle$ and $\eta_5 : \langle R_5, \{-, N_5^{c_2}\}\rangle$ with the upper bound score 0.492 as shown in Figure 5.8(b). Furthermore, $F(\eta_2)$ is refined as 0.492, due to the decrease of r-score of last seen node in $list_{c_2}(r)$.

- (c) The evaluation scans $N_4^{c_1}$ and $N_7^{c_2}$, followed by generating two partial states $\eta_4 : \langle R_4, \{N_4^{c_1}, -\}\rangle$ and $\eta_7 : \langle R_7, \{-, N_7^{c_2}\}\rangle$ and refining the upper bounds for η_2

States	Scores
$\langle R_3, \{N_3^{c_1}, -\}\rangle$	0.5095
$\langle R_2, \{-, N_2^{c_2}\}\rangle$	0.5095

(a) access $N_3^{c_1}$ and $N_2^{c_2}$

States	Scores
$\langle R_3, \{N_3^{c_1}, -\}\rangle$	0.5095
$\langle R_2, \{-, N_2^{c_2}\}\rangle$	0.492
$\langle R_8, \{N_8^{c_1}, -\}\rangle$	0.492
$\langle R_5, \{-, N_5^{c_2}\}\rangle$	0.492

(b) access $N_8^{c_1}$ and $N_5^{c_2}$

States	Scores
$\langle R_3, \{N_3^{c_1}, -\}\rangle$	0.5095
$\langle R_8, \{N_8^{c_1}, -\}\rangle$	0.492
$\langle R_7, \{-, N_7^{c_2}\}\rangle$	0.4575
$\langle R_2, \{-, N_2^{c_2}\}\rangle$	0.4575
$\langle R_4, \{N_4^{c_1}, -\}\rangle$	0.4575
$\langle R_5, \{-, N_5^{c_2}\}\rangle$	0.4575

(c) access $N_4^{c_1}$ and $N_7^{c_2}$

States	Scores
$\langle R_3, \{N_3^{c_1}, N_3^{c_2}\}\rangle$	0.4765
$\langle R_8, \{N_8^{c_1}, -\}\rangle$	0.46
$\langle R_7, \{N_7^{c_1}, N_7^{c_2}\}\rangle$	0.4575
$\langle R_2, \{-, N_2^{c_2}\}\rangle$	0.4575
$\langle R_5, \{-, N_5^{c_2}\}\rangle$	0.4575
$\langle R_4, \{N_4^{c_1}, -\}\rangle$	0.4245

(d) access $N_7^{c_1}$ and $N_3^{c_2}$

States	Scores
$\langle R_7, \{N_7^{c_1}, N_7^{c_2}\}\rangle$	0.4575
$\langle R_5, \{N_5^{c_1}, N_5^{c_2}\}\rangle$	0.4405
$\langle R_2, \{-, N_2^{c_2}\}\rangle$	0.4405
$\langle R_8, \{N_8^{c_1}, N_8^{c_2}\}\rangle$	0.3905
$\langle R_4, \{N_4^{c_1}, -\}\rangle$	0.355

(e) access $N_2^{c_1}$ and $N_8^{c_2}$

States	Scores
$\langle R_2, \{N_2^{c_1}, N_2^{c_2}\}\rangle$	0.4405
$\langle R_8, \{N_8^{c_1}, N_8^{c_2}\}\rangle$	0.3905
$\langle R_4, \{N_4^{c_1}, -\}\rangle$	0.355
$\langle R_9, \{-, N_9^{c_2}\}\rangle$	0.3205

(f) access $N_5^{c_1}$ and $N_9^{c_2}$

States	Scores
$\langle R_4, \{N_4^{c_1}, -\}\rangle$	0.3205
$\langle R_9, \{-, N_9^{c_2}\}\rangle$	0.286
$\langle R_6, \{N_6^{c_1}, -\}\rangle$	0.2515
$\langle R_1, \{-, N_1^{c_2}\}\rangle$	0.2515

(g) access $N_6^{c_1}$ and $N_1^{c_2}$

Figure 5.8: Sequence of updating $BHeap$ by OQ*-tree evaluation in Example 5.2

and η_5 as shown in Figure 5.8(c).

- (d) The evaluation accesses $N_7^{c_1}$ and $N_3^{c_2}$, followed by generating two complete states η_3 and η_7, and refining the upper bounds for other states. As can be seen in Figure 5.8(d), the top state η_3 is a complete state, and we can obtain the current top-2 result groups G'_1 ($\xi_\psi(G'_1) = 0.458$) and G'_2 ($\xi_\psi(G'_2) = 0.315$) (shown in Figure 1.3(b)) based on the retrieved tuples of $N_3^{c_1}$ and $N_3^{c_2}$.

- (e) The evaluation scans $N_5^{c_1}$ and $N_8^{c_2}$, and obtain two complete states η_5 and η_8. The top-2 states η_7 and η_5 can be processed.

- (f) The evaluation scans $N_2^{c_1}$ and $N_9^{c_2}$, and obtain the complete state η_2. The top-2 states η_2 and η_8 are processed.

Algorithm 5.8: `OQ-tree-SD-Eval` (Q, I, N)

Input: the N-DSQ query $Q(\psi, r, R_Q, k)$, and the IOQ-tree I, and the size of candidate groups N
Output: the spatially diversified set of k result groups

```
1 U ← the top-N highly ranked result groups generated by evoking
  OQ+-tree-Eval or OQ*-tree-Eval;
2 S ← ∅;
3 while |S| < k do
4 |   find G ∈ U to maximize f_G(S);
5 |   S ← S ∪ {G};
6 |   U ← U − {G};
7 return S;
```

- (g) The evaluation scans $N_6^{c_1}$ and $N_1^{c_2}$, followed by generating two partial states η_6 and η_9. Even though the top partial η_4 whose upper bound score is greater than $\xi_\psi(G'_2)$, there does not exist an candidate result group within R_4, since there exist no more r-related nodes in $list_{c_2}(r)$. Additionally, the upper bound scores of other states are smaller than $\xi_\psi(G'_2)$. Therefore, the evaluation is terminated.

□

5.6 Evaluation of N-DSQqueries

In this section, we discuss the query evaluation for a N-DSQ query $Q(\psi, r, R_Q, k)$. In the query evaluation, we apply the well known two-phase diversification model proposed in [19]: (a) retrieving the top-N high ranked result groups as the candidate set and (b) finding out the k spatially diversified result groups from the candidate set. More specifically, the candidate collection of result groups can be generated by applying the top-N evaluation methods mentioned in previous section. To address the NP-Complete problem of finding out the k spatial diversified result group from the candidate set, we use the greedy algorithm proposed in [13] with an approximation rate of 2.

The query evaluation will iteratively selecting one result group from the the candidate set. Let us first define the marginal gain of selecting a new candidate result group. For any given subset $S \subseteq U$ and an element $G \in U - S$, let $f_{\psi}^{G}(S)$ be the marginal gain of selecting G from the set $U - S$ calculated as

$$f_{\psi}^{G}(S) = \delta \cdot \frac{\xi(G)}{\xi_{max}} + \frac{1-\delta}{|S|+1} \cdot \sum_{G' \in S} \frac{dis(G, G')}{\Gamma} \tag{5.5}$$

The main algorithm of the query evaluation is shown in Algorithm 5.8. After generating the top-N result groups, the query evaluation iteratively picks the result group G with the maximum marginal gain $f_G(S)$ until that $|S| = k$.

5.7 Experiments

We conducted an experimental study to evaluate the efficiency of our proposed techniques. All of the indexes were implemented in Java and experiments were conducted on a server with an Intel Xeon 1.80GHz processor, 32GB of memory, running Ubuntu 14.04. In our experiments, each execution time reported refers to the total running time for a query. Each query is run 5 times, and the reported running time is the average of 3 values excluding the minimum and maximum values.

Algorithms. For DSQ queries, we compared OQ^{+}-tree and OQ*-tree against the baseline I^{3+}, which is an extended evaluation method by using `I3-index` (mentioned in Section 5.3). For N-DSQ queries, we compared OQ^{+}-tree-SD and OQ*-tree-SD against the baseline I^{3+}-SD. The three methods OQ^{+}-tree-SD, OQ*-tree-SD and I^{3+}-SD generate the candidate set by applying OQ^{+}-tree, OQ*-tree and I^{3+}, respectively. In our experiments, we did not compare our proposed algorithms against the baseline algorithm using R-tree, since the spatial join on different R-trees is very costly.

Dataset. In this experiment, we used two real spatial datasets *Foursquare* with 3,396,580 spatial Points of Interest (PoIs) and *Tweets* with 122,472,892 geo-tweets.

Parameter	Parameter Value (Default Value)
number of keyword concepts (DSQ or N-DSQ)	1-4 (2)
query limit size k	1-100 (10)
query radius r	0.001°-0.004° (0.002°)
query region R_Q	Global, US, UK, Germany (Global)

Table 5.7: Query Parameters

Queries. Table 5.7 shows the parameters of spatial queries used in our experiments. For a DSQ or N-DSQ query, we varied the number of query keyword concepts from 1 to 4, with the default value of 2, and varied the limit size from 1 to 100, with the default value of 10. The query radius r was varied from 0.001° to 0.004°, with the default value of 0.002°. Note that the geo-distance is around 111 meter when r equals to $0.001°$. To study the effect of the query region R_Q, we investigated the performance on different query regions: the global geo-location region $\{[-180°, 180°][-90°, 90°]\}$, the US geo-location region $\{[-125°, -70°][30°, 48°]\}$, the UK geo-location region $\{[-7°, 2°][50°, 58°]\}$ and the Germany geo-location region $\{[6°, 15°][47°, 55°]\}$.

The real dataset *Foursquare* was evaluated using the five queries (Q_1 to Q_5) shown in Table 5.8, while the real dataset *Tweets* was evaluated using the five queries (Q'_1 to Q'_5) shown in Table 5.9.

Query	Query Type	Keyword Concepts
Q_1	DSQ	"*Restaurant*"
Q_2	DSQ	"*Restaurant*", "*Entertainment*"
Q_3	DSQ	"*Restaurant*", "*Entertainment*", "*Hotel*"
Q_4	DSQ	"*Restaurant*", "*Entertainment*", "*Hotel*", "*Shopping*"
Q_5	N-DSQ	"*Restaurant*", "*Entertainment*"

Table 5.8: Queries on *Foursquare*

For each real dataset, we built an IOQ$^+$-tree that is a variant of IOQ-tree where each

Query	Query Type	Keyword Concepts
Q'_1	DSQ	"*Sport*"
Q'_2	DSQ	"*Sport*", "*Health Eating*"
Q'_3	DSQ	"*Sport*", "*Health Eating*", "*Healthcare*"
Q'_4	DSQ	"*Sport*", "*Health Eating*", "*Healthcare*", "*Exercise*"
Q'_5	N-DSQ	"*Sport*", "*Health Eating*"

Table 5.9: Queries on *Tweets*

partition is organized as an OQ$^+$-tree, an IOQ*-tree that is a variant of IOQ-tree where each partition is organized as an OQ*-tree, and an I^3 [82]. For each keyword concept, we used the LDA model [11] to generate the set of relevant sub-concepts associated with their weights. In our experiments, the number of relevant sub-concepts is set as 16. In the IOQ$^+$-tree and IOQ*-tree, we varied the number of maintained scores κ from 4 to 8, with a default value of 8. Figure 5.9 shows the size of implemented indexes for the two real datasets. Our implementation shows that IOQ$^+$-tree is about 2.4 times larger than I^3, while IOQ*-tree is about 7.6 times larger than I^3. As an example, when $\kappa = 8$, the size of IOQ$^+$-tree in *Foursquare* is 4.4GB while the size of I^3 is only 1.3GB. Furthermore, when κ increases from 4 to 8, the size of IOQ$^+$-tree increases by a factor of 1.6.

Index	Index Size	
	$\kappa = 4$	$\kappa = 5$
I^3	1.3GB	1.3GB
IOQ$^+$-tree	2.7GB	4.4GB
IOQ*-tree	5.1GB	9.9GB

(a) Foursquare

Index	Index Size	
	$\kappa = 4$	$\kappa = 5$
I^3	48.7GB	48.7GB
IOQ$^+$-tree	101.8GB	173.6GB
IOQ*-tree	196.8GB	381.9GB

(b) Tweets

Figure 5.9: Index sizes on the two real datasets

5.7.1 Simple DSQ queries with only one keyword concept

In this section, we investigate the performance study of simple DSQ queries containing only one keyword concept, by varying several different parameters.

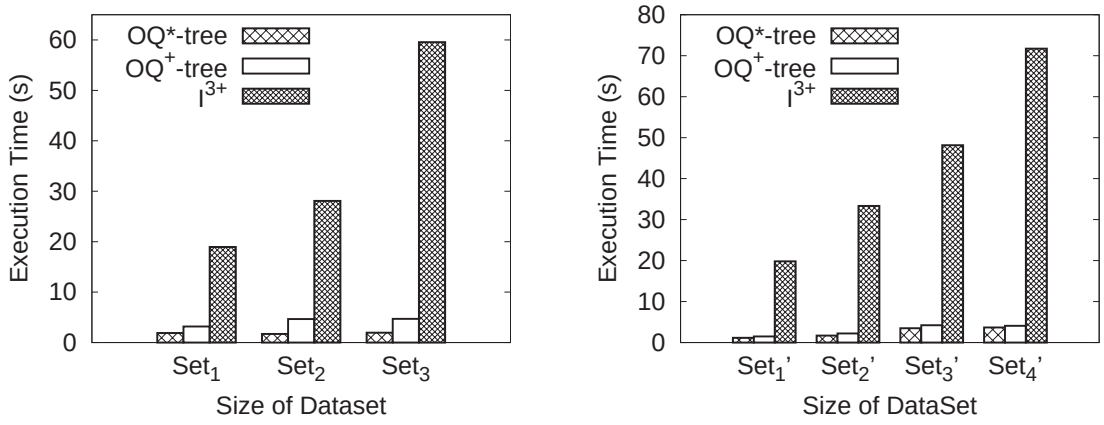
Effect of data size

Figure 5.10 compares the performance for different data sizes on Q_1 and Q'_1. For Q_1 on the *Foursquare* dataset, there are a total of 796,482 PoIs (denoted by Set_3) that cover the keyword concept "*Restaurant*". To study the effect of data size, we generated two small data sets Set_1 and Set_2, by randomly picking 300,000 and 500,000 PoIs from Set_3, respectively. As can be seen in Figure 5.10(a), the results show that OQ*-tree gives the best performance and it outperforms I^{3+} by an increasing factor of 10.1, 16.4 and 30.4 as the data size increases. Observe that while OQ$^+$-tree and OQ*-tree perform similarly for the different data sizes, I^{3+}'s performance worsens with increasing data size. Furthermore, the results also show that OQ*-tree outperforms OQ$^+$-tree by up to a factor of 2.7. The number of accessed r-related nodes by OQ*-tree increases from 23 to 29 and 40 as the dataset is varied from Set_1 to Set_2 and Set_3, while the number of accessed nodes by OQ$^+$-tree increases from 67 to 102 and 141.

For Q'_1 on the *Tweets* dataset, there are a total of 931,827 geo-tweets (denoted by Set'_4) that cover the keyword concept "*Sport*". Additionally, we generated three small sets Set'_1, Set'_2 and Set'_3 by randomly picking 300,000, 500,000, 700,000 geo-tweets from Set'_4, respectively. In Figure 5.10(b), the results for Q'_1 show similar performance trends with the previously mentioned results for Q_1.

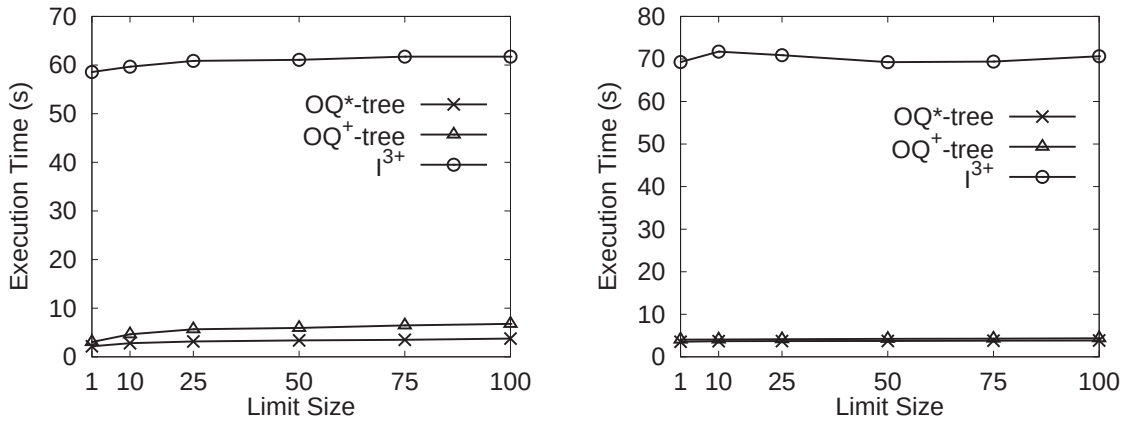
Effect of query limit, k

Figure 5.11 compares the performance for different values of the query limit k on Q_1 and Q'_1. Here again, the results for Q_1 show that OQ*-tree gives the best performance which outperforms I^{3+} by up to a factor of 27. The number of r-related nodes accessed by OQ*-tree increases from 34 to 139 as k increases from 1 to 100, while the number of nodes and node-combinations accessed by I^{3+} increases from 149,776 to 198,832. The results for Q'_1 show similar performance trends.



(a) Foursquare (b) Tweets

Figure 5.10: Effect of data size



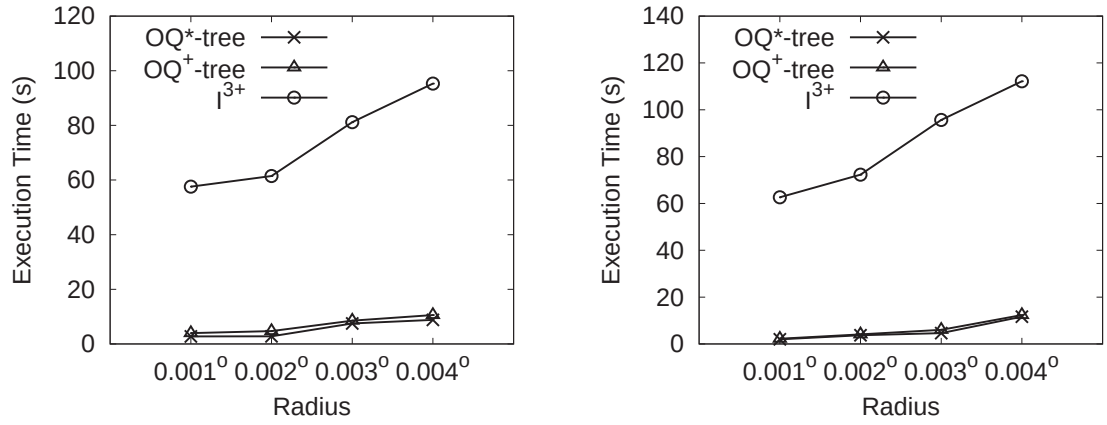
(a) Foursquare (b) Tweets

Figure 5.11: Effect of query limit, k

Effect of query radius, r

Figure 5.12 compares the performance for different query radius r on Q_1 and Q_1'. The results for Q_1 show that OQ*-tree gives the best performance which outperforms I^{3+} by up to a factor of 21. Observe that the performance of each method worsens with increasing query radius r. The performance is dominated by the circle-placement evaluation on the increasing number of tuples within each accessed r-related node as the radius r increases. The CPU execution time of OQ*-tree increases from 1.9 second to 2.2, 6.7 and 7.9 as radius r increases from 0.001° to 0.002°, 0.003° and 0.004°. The results for Q_1' show

similar performance trends.



(a) Foursquare (b) Tweets

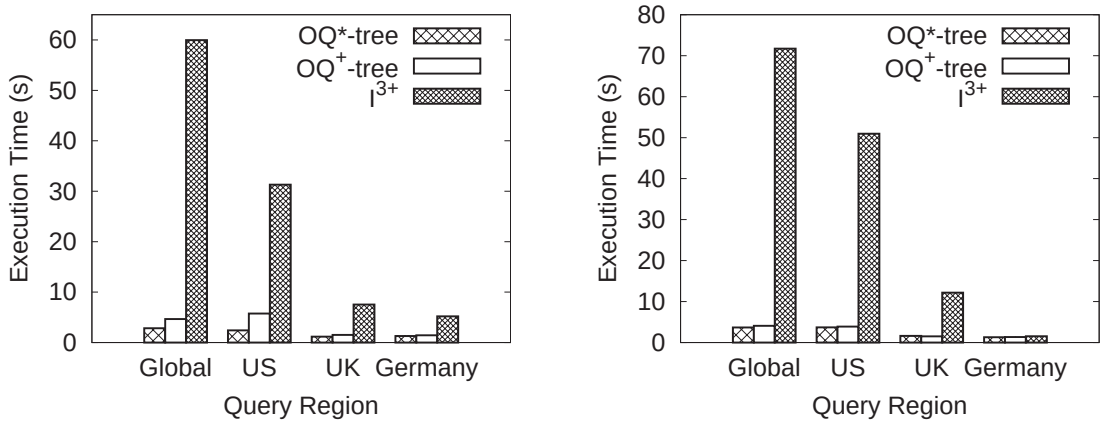
Figure 5.12: Effect of query radius, r

Effect of size of query region, R_Q

Figure 5.13 compares the performance for different query search region R_Q on Q_1 and Q'_1 when varying the query search region R_Q from the global geo-location region to the US geo-location region, the UK geo-location region and the Germany geo-location region. The results for Q_1 show that OQ*-tree gives the best performance which is improved by the percentages of 15.1%, 58.7% and 54.5% as the global region is restricted to the geo-location region of US, UK and Germany, respectively. The number of r-related nodes accessed by OQ*-tree reduces from 70 to 30, 27 and 23 as R_Q is restricted from the global region to US, UK and Germany, respectively. Observe that the performance of other two methods (OQ$^+$-tree and I^{3+}) is also improved as R_Q is restricted. The results for Q'_1 show similar performance trends.

Effect of the number of maintained scores, κ

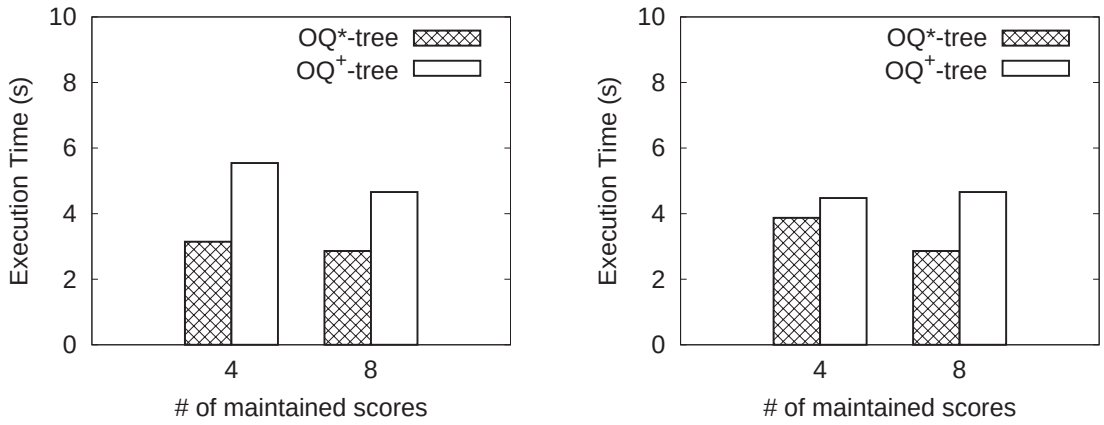
Figure 5.14 compares the performance for different values of the number of maintained scores κ on Q_1 and Q'_1. The results for Q_1 show that the performance of the two proposed



(a) Foursquare (b) Tweets

Figure 5.13: Effect of query region R_Q

evaluation methods (OQ$^+$-tree and OQ*-tree) is improved as κ increases from 4 to 8. The number of r-related nodes accessed by OQ*-tree reduces from 147 to 70 as κ increases from 4 to 8. Furthermore, the results for Q'_1 show similar performance trends.



(a) Foursquare (b) Tweets

Figure 5.14: Vary κ

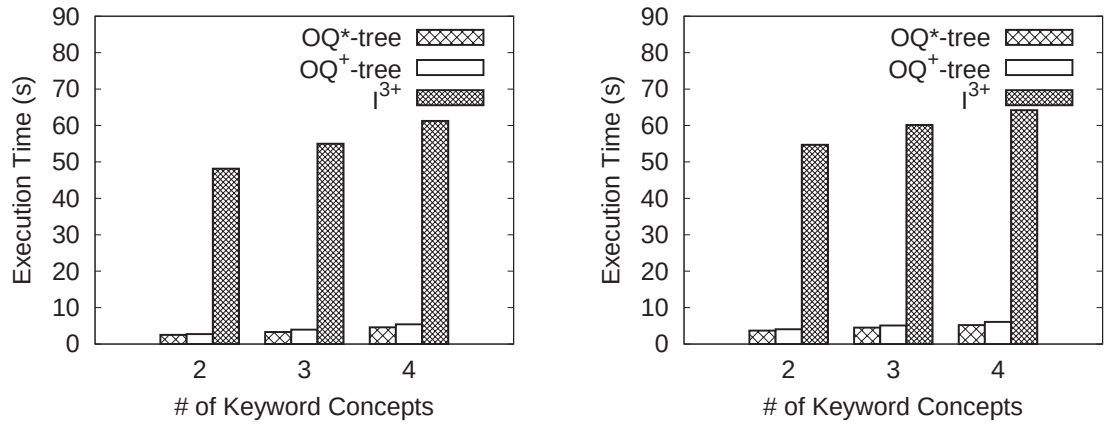
5.7.2 DSQ queries with multiple keyword concepts

In this section, we investigate the performance study of DSQ queries with multiple keyword concepts by varying different parameters.

Effect of the number of query keyword concepts, $|\psi|$

Figure 5.15 compares the performance as the number of query keyword concepts is varied. In the dataset *Foursquare*, we used queries Q_2, Q_3 and Q_4, which have 2, 3 and 4 keyword concepts, respectively. In the dataset *Tweets*, we used queries Q'_2, Q'_3 and Q'_4, which have 2, 3 and 4 keyword concepts, respectively.

The results for Q_2, Q_3 and Q_4 show that OQ*-tree gives the best performance and it outperforms I^{3+} by a factor of 19.1, 16.8 and 13.3, respectively. For the three methods, their performance worsens when $|\psi|$ increases from 2 to 4, since they need to spatially join among more OQ-trees (Quadtrees). The results for the three queries (Q'_2, Q'_3 and Q'_4) in *Tweets* show similar performance trends.

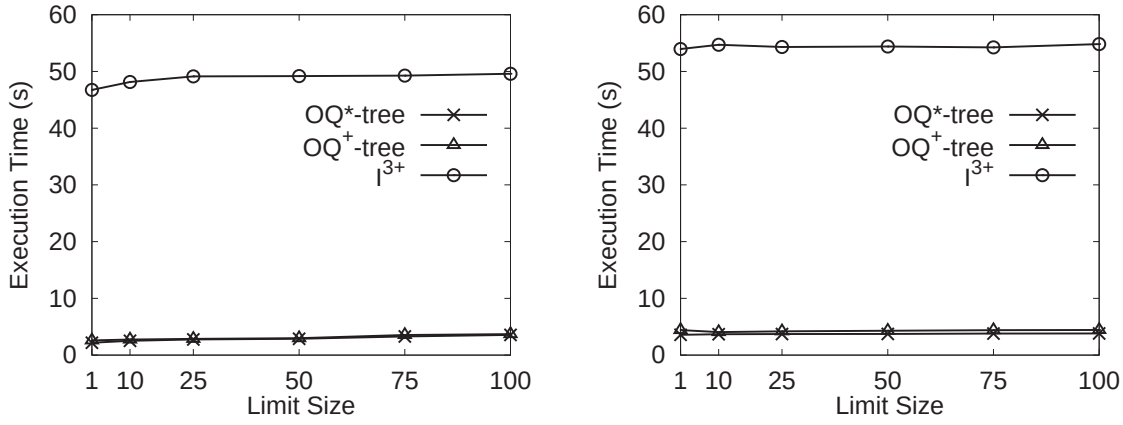


(a) Foursquare (b) Tweets

Figure 5.15: Effect of the number of query keyword concepts $|\psi|$

Effect of query limit, k

Figure 5.16 compares the performance for different values of the query limit k on Q_2 and Q'_2. The results for Q_2 show that OQ$^+$-tree and OQ*-tree perform similarly and they outperform I^{3+} by up to a factor of 21.5. The results for Q'_2 show similar performance trends.

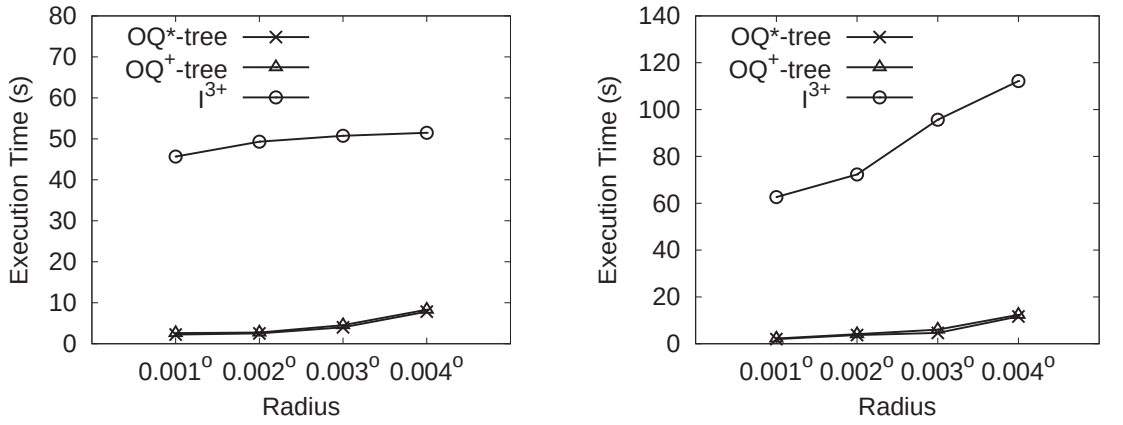


(a) Foursquare (b) Tweets

Figure 5.16: Effect of query limit k

Effect of query radius, r

Figure 5.17 compares the performance for different query radius r on Q_2 and Q_2'. The results for Q_2 show that OQ*-tree gives the best performance which outperforms I^{3+} by up to a factor of 21. Observe that the performance of each method worsens with increasing query radius r. Similar with our previous study of the effect of r for DSQ queries with one keyword concept, the performance of each method for Q_2 worsens with increasing query radius r. The results for Q_2' show similar performance trends.

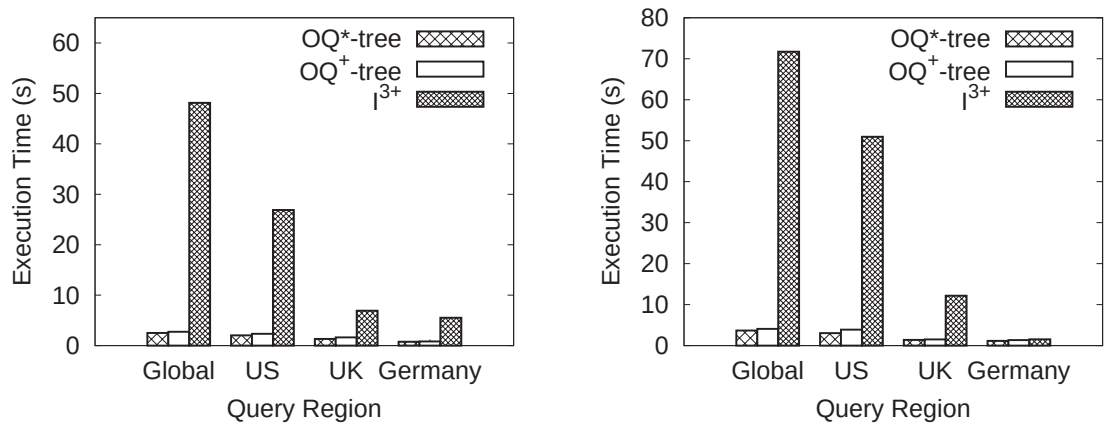


(a) Foursquare (b) Tweets

Figure 5.17: Effect of query radius r

Effect of size of query region, R_Q

Figure 5.13 compares the performance for different query region R_Q on Q_2 and Q_2' when R_Q is varied from the global geo-location region to the geo-location regions of US, UK and Germany. The results for Q_2 show that the performance of OQ$^+$-tree and OQ*-tree is improved as R_Q is restricted and they outperform I^{3+} by up to a factor of 19.1. The results for Q_2' show similar performance trends.



(a) Foursquare (b) Tweets

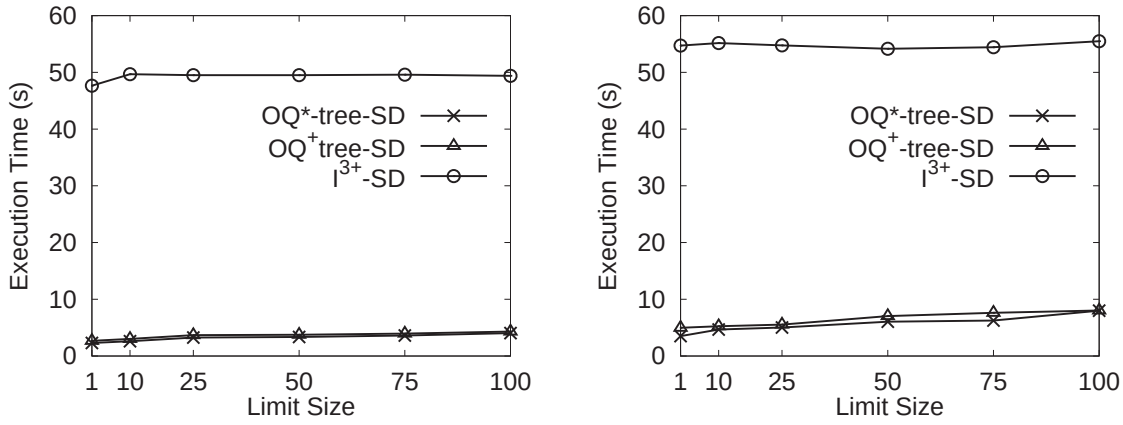
Figure 5.18: Effect of query region R_Q

5.7.3 Comparison on Evaluations for N-DSQ queries

For N-DSQ queries, the two-phase diversification model [19] is applied for each compared method. More specifically, each method first collect a candidate set of top-N result groups, followed by spatially diversifying k groups from the candidate collection.

In our experiments, we set $N = 5 \times k$. Figure 5.19 compares the performance for different values of query limit on Q_5 and Q_5'. The results for Q_5 show that OQ$^+$-tree-SD and OQ*-tree-SD perform similarly and they outperform I^{3+}-SD by up to a factor of 20.7, and the results for Q_5' show similar performance trends. Observe that the performance trends of Q_5 and Q_5' are similar to that of Q_2 and Q_2' shown in Figure 5.16, since the query execution

is dominated by generating the candidate set for a relative small k. Therefore, we omit the study of the effect of other parameters due to the performance domination of generating the top-N result groups by using either method (OQ$^+$-tree, OQ*-tree and I^{3+}).



(a) Foursquare (b) Tweets

Figure 5.19: Effect of limit size k

5.8 Summary

In this chapter, we have examined the problem of diversified spatial keyword search. We have designed two novel spatial diversity queries (DSQ and N-DSQ), and proposed a novel textual-first index, IOQ-tree, to evaluate each type of spatial queries. Based on the spatial index, two evaluation methods have been proposed to efficiently evaluate each type of spatial queries. Our comprehensive performance study comparing against the state-of-the-art technique [82] showed that both of our proposed algorithms (OQ$^+$-tree and OQ*-tree) outperforms existing techniques on average by a factor of 20. For spatial diversity queries with only one keyword concept, the OQ*-tree algorithm outperforms OQ$^+$-tree by up to a factor of 2.7. On the other hand, for spatial diversity queries with multiple keyword concepts, the two algorithms have the similar performance. Therefore, for these queries, we recommend the OQ$^+$-tree algorithm due to the low maintainance cost of the OQ$^+$-tree indexes.

CHAPTER 6

CONCLUSION

In this thesis, we have studied three problems related to the efficient computation of diverse query results, namely, indexing for dynamic diversity queries, evaluation of multiple diversity queries, and diversified spatial keyword search. In this chapter, we summarize our works and highlight some interesting works that are worthy of further exploration.

6.1 Contributions

Our first contribution is the study of efficient evaluation techniques for the computation of diverse query results with respect to a sequence of attributes known as the d-order. We observe that it is very inefficient to evaluate dynamic diversity queries (DDQs) with dynamic d-orders by extending existing techniques [70] designed for static diversity queries

(SDQs) with a predefined d-order. We further propose a novel approach for evaluating diversity queries that is based on the concept of computing a core cover of a query. Based on this concept, we design a new index method, D-Index, and introduce two index variants, namely, `D-tree` and `D+-tree`. Our experimental results on PostgreSQL demonstrate that our proposed D-Index technique consistently outperforms [70] for both SDQs as well as DDQs.

Our second contribution is the study of optimization of multiple online DDQs. We first propose a new framework to maximize the shared index scans among multiple online queries by reordering the execution of these online queries. We then present a novel technique of adaptive query evaluation to dynamically adapt the query plans by switching query evaluation to scan another inactive index. Moreover, we introduce an online index tuning technique to automatically adapt the set of physical indexes by exploiting the waiting queries. Our experimental results on PostgreSQL demonstrate the efficiency of our proposed techniques.

Our third contribution is the study of diversified spatial keyword search. We first propose two novel spatial diversity keyword queries: DSQ and N-DSQ. We observe that existing spatial indexes [82] are inefficient to evaluate such spatial queries, and we introduce a new textual-first spatial index, termed IOQ-tree, where each inverted posting list corresponding to a keyword concept is organized based on a novel space-partitioning Quadtree-like structure termed OQ-tree with two variants (OQ^+-tree and OQ^*-tree). Based on the two variants of IOQ-tree, we propose two efficient evaluation methods for each type of spatial queries. Our experimental results on two real datasets (*Foursquare* and *Tweets*) demonstrate that our proposed techniques outperforms the state-of-the-art technique [82] by up to one order of magnitude.

6.2 Future works

In this section, we discuss some interesting future directions related to the problems examined in this thesis.

6.2.1 d-order Recommendation

In Chapter 3, we studied the problem of diversifying DDQs, based on the assumption that the d-order of a DDQ can well represent the user preference. However, in some real applications, a user might not be familiar with his own preference, and it will be quite challenging for him to issue a proper d-order. An interesting direction for future work is to improve the database usability by recommending a set of frequently used d-orders.

6.2.2 Adaptive Query Evaluation Generalization

The technique of adaptive query evaluation studied in Chapter 4 can be used to dynamically adapt the query plans for DDQs by switching a current query evaluation to scan an inactive partial $\texttt{D}^+\texttt{-tree}$ index. As mentioned, a partial $\texttt{D}^+\texttt{-tree}$ index is a B^+-tree index, which is frequently used to evaluate the conventional SQL queries in DBMSs. An interesting direction for future work is to generalize the technique of adaptive query evaluation to optimize multiple queries evaluations on B^+-tree indexes.

6.2.3 Efficient Spatial Diversification Model

For a N-DSQ query, the query evaluation studied in Chapter 5 uses the two-phase model to first generating a candidate set of top-N result groups, followed by spatially diversifying

the candidate set. One disadvantage of this approach is how to determine the value of N. In some real applications, the top-N groups could be highly spatially overlapped. In such scenario, we need to set N to be a large value, and the query performance could be much worse for the large N. This motivated an interesting direction to directly diversify result groups rather than using the two-phase model.

BIBLIOGRAPHY

[1] ”foursquare offical website”. https://foursquare.com/.

[2] The statistics for amazon. http://www.123cha.com/alexa/amazon.com.

[3] The statistics for taobao. http://www.123cha.com/alexa/taobao.com.

[4] Result diversity. *IEEE Data Eng. Bull.*, 32(4), 2009.

[5] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong. Diversifying search results. In *WSDM*, pages 5–14, 2009.

[6] S. Agrawal, S. Chaudhuri, L. Kollar, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database tuning advisor for microsoft sql server 2005. In *SIGMOD*, pages 930–932, 2005.

[7] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *PVLDB*, pages 496–505, 2000.

[8] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: Workload as a sequence. In *SIGMOD*, pages 683–694, 2006.

[9] A. Angel and N. Koudas. Efficient diversity-aware search. In *SIGMOD*, pages 781–792, 2011.

[10] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *CIDR*, pages 238–249, 2005.

[11] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, 2003.

[12] K. S. Bøgh, A. Skovsgaard, and C. S. Jensen. Groupfinder: A new approach to top-k point-of-interest group retrieval. In *PVLDB*, pages 1226–1229, 2013.

[13] A. Borodin, H. C. Lee, and Y. Ye. Max-sum diversification, monotone submodular functions and dynamic updates. *CoRR*, abs/1203.6397, 2012.

[14] K. Bradley and B. Smyth. Improving recommendation diversity. In *AICS*, pages 75–84, 2001.

[15] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, pages 826–835, 2007.

[16] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. In *PVLDB*, pages 277–288, 2009.

[17] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD*, pages 373–384, 2011.

[18] G. Capannini, F. M. Nardini, R. Perego, and F. Silvestri. Efficient diversification of web search results. In *PVLDB*, pages 451–459, 2011.

[19] J. Carbonell and J. Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *SIGIR*, pages 335–336, 1998.

[20] A. Cary, O. Wolfson, and N. Rishe. Efficient and scalable method for processing top-k spatial boolean queries. In *SSDBM*, pages 87–95, 2010.

[21] I. Catallo, E. Ciceri, P. Fraternali, D. Martinenghi, and M. Tagliasacchi. Top-k diversity queries over bounded regions. *ACM Trans. Database Syst.*, 38(2):10:1–10:44, 2013.

[22] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *PVLDB*, pages 146–155, 1997.

[23] B. M. Chazelle and D. T. Lee. On a circle placement problem. *Computing*, 36(1-2):1–16, 1986.

[24] H. Chen and D. R. Karger. Less is more: Probabilistic models for retrieving fewer relevant documents. In *SIGIR*, pages 429–436, 2006.

[25] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: an experimental evaluation. In *PVLDB*, pages 217–228, 2013.

[26] Z. Chen and T. Li. Addressing diverse user preferences in sql-query-result navigation. In *SIGMOD*, pages 641–652, 2007.

[27] S. Cheng, A. Arvanitis, M. Chrobak, and V. Hristidis. Multi-query diversification in microblogging posts. In *EDBT*, pages 133–144, 2014.

[28] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel. Text vs. space: Efficient geo-search query processing. In *CIKM*, pages 423–432, 2011.

[29] C. L. Clarke, M. Kolla, and O. Vechtomova. An effectiveness measure for ambiguous and underspecified queries. In *ICTIR*, pages 188–199, 2009.

[30] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. In *PVLDB*, pages 337–348, 2009.

[31] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. In *PVLDB*, pages 337–348, 2009.

[32] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic sql tuning in oracle 10g. In *PVLDB*, pages 1098–1109, 2004.

[33] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.

[34] E. Demidova, P. Fankhauser, X. Zhou, and W. Nejdl. Divq: Diversification for keyword search over structured databases. In *SIGIR*, pages 331–338, 2010.

[35] Z. Drezner. Note—on a modified one-center model. *Management Science*, 27(7):848–851, 1981.

[36] M. Drosou and E. Pitoura. Diversity over continuous data. *IEEE Data Eng. Bull.*, 32(4):49–56, 2009.

[37] M. Drosou and E. Pitoura. Search result diversification. *SIGMOD Rec.*, 39(1):41–47, 2010.

[38] M. Drosou and E. Pitoura. Disc diversity: Result diversification based on dissimilarity and coverage. In *PVLDB*, pages 13–24, 2012.

[39] B. Eravci and H. Ferhatosmanoglu. Diversity based relevance feedback for time series search. In *PVLDB*, pages 109–120, 2013.

[40] W. Fan, X. Wang, and Y. Wu. Diversified top-k graph pattern matching. In *PVLDB*, pages 1510–1521, 2013.

[41] S. Finkelstein. Common expression analysis in database applications. In *SIGMOD*, pages 235–245, 1982.

[42] S. Gollapudi and A. Sharma. An axiomatic approach for result diversification. In *WWW*, pages 381–390. ACM, 2009.

[43] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (sk) queries in geographic information retrieval (gir) systems. In *SSDBM*, page 16, 2007.

[44] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. *SIGMOD Rec.*, 25(2):205–216, 1996.

[45] X. Huang, H. Cheng, R.-H. Li, L. Qin, and J. X. Yu. Top-k structural diversity search in large networks. In *PVLDB*, pages 1618–1629, 2013.

[46] A. Jain, P. Sarda, and J. R. Haritsa. Providing diversity in k-nearest neighbor query results. In *PAKDD*, pages 404–413, 2004.

[47] D. S. Johnson. Approximation algorithms for combinatorial problems. In *STOC*, pages 38–49, 1973.

[48] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. *SIGMOD Rec.*, 27(2):106–117, 1998.

[49] C. A. Lang, B. Bhattacharjee, T. Malkemus, and K. Wong. Increasing buffer-locality for multiple index based scans through intelligent placement and index scan speed control. In *PVLDB*, pages 1298–1309, 2007.

[50] L. Li and C.-Y. Chan. Efficient indexing for diverse query results. In *PVLDB*, pages 745–756, 2013.

[51] C. Long, R. C.-W. Wong, K. Wang, and A. W.-C. Fu. Collective spatial keyword queries: A distance owner-driven approach. In *SIGMOD*, pages 689–700, 2013.

[52] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *SIGMOD*, pages 659–670, 2004.

[53] E. Minack, W. Siberski, and W. Nejdl. Incremental diversification for very large sets: A streaming-based approach. In *SIGIR*, pages 585–594, 2011.

[54] M. L. Paramita, J. Tang, and M. Sanderson. Generic and spatial approaches to image search results diversification. In *ECIR*, pages 603–610, 2009.

[55] J. Park and A. Segev. Using common subexpressions to optimize multiple queries. In *ICDE*, pages 311–319, 1988.

[56] O. A. Prokopyev, N. Kong, and D. L. Martinez-Torres. The equitable dispersion problem. *EJOR*, 197(1):59–67, 2009.

[57] F. Radlinski and S. Dumais. Improving personalized web search using result diversification. In *SIGIR*, pages 691–692, 2006.

[58] F. Radlinski, R. Kleinberg, and T. Joachims. Learning diverse rankings with multi-armed bandits. In *ICML*, pages 784–791, 2008.

[59] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Norvag. Efficient processing of top-k spatial keyword queries. In *SSTD*, pages 205–222, 2011.

[60] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Rec.*, 29(2):249–260, 2000.

[61] R. L. Santos, C. Macdonald, and I. Ounis. Exploiting query reformulations for web search result diversification. In *WWW*, pages 881–890, 2010.

[62] A. D. Sarma, S. Gollapudi, and S. Ieong. Bypass rates: reducing query abandonment using negative inferences. In *KDD*, pages 177–185, 2008.

[63] K.-U. Sattler, I. Geist, and E. Schallehn. Quiet: Continuous query-driven index tuning. In *PVLDB*, pages 1129–1132, 2003.

[64] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. Colt: continuous on-line tuning. In *SIGMOD*, pages 793–795, 2006.

[65] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13:23–52, 1988.

[66] S. N. Subramanian and S. Venkataraman. Cost-based optimization of decision support queries using transient views. In *SIGMOD*, pages 319–330, 1998.

[67] J. Tang and M. Sanderson. Evaluation and user preference study on spatial diversity. In *ECIR*, pages 179–190, 2010.

[68] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson. Spatio-textual indexing for geographical search on the web. In *SSTD*, pages 218–235, 2005.

[69] M. J. van Kreveld, I. Reinbacher, A. Arampatzis, and R. van Zwol. Multi-dimensional scattered ranking methods for geographic information retrieval. *GeoInformatica*, 9(1):61–84, 2005.

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

[71] M. R. Vieira, H. L. Razente, M. C. N. Barioni, M. Hadjieleftheriou, D. Srivastava, C. Traina-Jr., and V. J. Tsotras. On query result diversification. In *ICDE*, pages 1163–1174, 2011.

[72] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen. Joint top-k spatial keyword query processing. *IEEE Trans. Knowl. Data Eng.*, 24(10):1889–1903, 2012.

[73] D. Yin, Z. Xue, X. Qi, and B. D. Davison. Diversifying search results with popular subtopics. In *TREC*, 2009.

[74] C. Yu, L. Lakshmanan, and S. A. Yahia. It takes variety to make a world: diversification in recommender systems. In *EDBT*, pages 368–378, 2009.

[75] Y. yu Chen. Efficient query processing in geographic web search engines. In *SIGMOD*, pages 277–288, 2006.

[76] C. Zhai and J. Lafferty. A study of smoothing methods for language models applied to information retrieval. *ACM Trans. Inf. Syst.*, 22(2):179–214, 2004.

[77] C. Zhai and J. Lafferty. A risk minimization framework for information retrieval. *Inf. Process. Manage.*, 42(1):31 – 55, 2006.

[78] C. X. Zhai, W. W. Cohen, and J. Lafferty. Beyond independent relevance: methods and evaluation metrics for subtopic retrieval. In *SIGIR*, pages 10–17, 2003.

[79] C. Zhang, Y. Zhang, W. Zhang, X. Lin, M. A. Cheema, and X. Wang. Diversified spatial keyword search on road networks. In *EDBT*, pages 367–378, 2014.

[80] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In *ICDE*, pages 688–699, 2009.

[81] D. Zhang, B. C. Ooi, and A. K. H. Tung. Locating mapped resources in web 2.0. In *ICDE*, pages 521–532, 2010.

[82] D. Zhang, K.-L. Tan, and A. K. H. Tung. Scalable top-k spatial keyword search. In *EDBT*, pages 359–370, 2013.

[83] Y. Zhang, J. Callan, and T. Minka. Novelty and redundancy detection in adaptive filtering. In *SIGIR*, pages 81–88, 2002.

[84] Y. Zhao, P. M. Deshpande, J. F. Naughton, and A. Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. *SIGMOD Rec.*, 27(2):271–282, 1998.

[85] W. Zheng, X. Wang, H. Fang, and H. Cheng. Coverage-based search result diversification. *Inf. Retr.*, 15(5):433–457, 2012.

[86] J. Zhou, P.-A. Larson, J. C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, pages 533–544, 2007.

[87] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In *CIKM*, pages 155–162, 2005.

[88] C.-N. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen. Improving recommendation lists through topic diversification. In *WWW*, pages 22–32, 2005.

[89] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. Db2 design advisor: Integrated automatic physical database design. In *PVLDB*, pages 1087–1097, 2004.

APPENDIX A

LEMMA PROOFS

A.1 Proof of Lemma 3.1

Proof. We now prove the first part of Lemma 3.1. Consider subtree T' rooted at a node V'. (1) If V' is a leaf node, T' is obviously a diverse tree, since there is only one tuple covered by $ptup_\delta(V')$ in a diverse result set. (2) Otherwise, V' is an internal node, (a) if the number of child nodes of V' is equal to $size(T')$, T' is also a diverse tree, since for each two tuples t, t' covered by $ptup_\delta(V')$ in a diverse result set, we have that $SIM_{\delta,ptup_\delta(V')}(t, t') = 0$; (b) we focus on the last case that V' is balanced and not expandable. Since V' is balanced, the only way to reduce $\sum_{t,t' \in S(ptup_\delta(V'))} SIM_{\delta,ptup_\delta(V')}(t, t')$ is that there are more different child nodes for V', but it is impossible since V' is not expandable. Therefore, T' is also a diverse tree.

Subsequently, we consider the case that T' is T. Since T is a b-diverse tree, we have that each subtree rooted at each node in T is a b-diverse tree. Therefore, each subtree is a diverse tree, and we can conclude that T is a diverse result trie, since T is a diverse tree wrt each δ-prefix tuple. □

A.2 Proof of Theorem 3.1

Proof. Let's prove it by contradiction. Consider a core cover $cover(T)$ for Q, there exists such a node V in T that V is expandable and T_V is not k-optimal. Then we need to consider two cases: (1) V is expandable and T_V is not k-sufficient; (2) V is expandable and T_V is k-sufficient but not b-diverse.

We first discuss about Case 1. For simplify, we assume that V is the highest node in T that V is expandable and T_V is not k-sufficient. If V is the root of T, we can easily improve the diversity of T by expanding V, so $cover(T)$ is not a core cover for Q. Otherwise, let T' be the largest subtree rooted at a sibling node V' of V, and we have that T' is k-sufficient and $size(T') - size(T) > 1$. Since V is expandable, and we use V_e to denote an expand child node of V. Let S_1 be a result set cover by $cover(T)$, and we construct a new result set S_2 by replacing a tuple covered by $ptup_\delta(V')$ in S_1 with a tuple covered by $ptup_\delta(V_e)$. Let V_p be the parent node of V and V', and we have that S_2 is more diverse wrt $ptup_\delta(V_p)$. Therefore, $cover(T)$ is not a core cover for Q.

On the other hand, we discuss about Case 2. In such case, we have that the size of T_V is larger than the number of child nodes of V. Let T' be the largest child subtree rooted at V', and then we have that $size(T') > 1$. Since V is expandable, and we use V_e to denote an expand child node of V. Let S_1 be a result set cover by $cover(T)$, and we construct a new result set S_2 by replacing a tuple covered by $ptup_\delta(V')$ in S_1 with a tuple covered by

$ptup_\delta(V_e)$. Subsequently, we have that S_2 is more diverse wrt $ptup_\delta(V)$. Therefore, we can conclude a contradictory result that $cover(T)$ is not a core cover for Q. □

A.3 Proof of Lemma 3.3

Proof. Lemma 3.3 can be easily proved based on the core cover definition. □

A.4 Proof of Lemma 4.1

Proof. let us now prove Lemma 4.1. For an accessed entry $e(a_1, a_2, \cdots, a_n)$ on index I with index key $(A_1, A_2, \cdots, A_n)$, the evaluation of query Q with d-order $\delta = (D_1, \cdots, D_m)$ will optimize the current diverse result set by extracting the prefix $(d_1, \cdots, d_m) = \pi_\delta e$. For another index I' with index key $(A'_1, A'_2, \cdots, A'_n)$, we can find such an entry $e' \in M_{I \to I'}(e)$ that $(d_1, \cdots, d_m) = \pi_\delta e$ based on the definition of D-Index. Therefore, we have the evaluation on entry e is equivalent to that on entry e'. □

A.5 Proof of Lemma 4.2

Proof. We can easily prove Lemma 4.2 since for any entry e on partition $I - E$ of index I, there will exist an equivalent entry e' on the partition $I' - M_{I \to I'}(E)$ of index I'. □

A.6 Proof of Lemma 5.1

Proof. Let us prove Lemma 5.1 by contradiction. We assume that there does not exist a r-related node that covers a circle of radius r. That is, there will exist such a circle of

radius r that spans two neighboring r-related nodes at level ℓ_r as shown in Figure A.1.

For convenience, we use L_ℓ to denote the breadth of the corresponding rectangular region of node N at level ℓ. Thus, we have that $2r > \frac{1}{2}L_{\ell_r}$. However, we can deduce the contradiction that $r > \frac{1}{4}L_{\ell_r} \geq \frac{L_\mathbb{R}}{4 \cdot 2^{\ell_r}} = \frac{L_\mathbb{R}}{4 \cdot 2^{\lfloor \lg \frac{L_\mathbb{R}}{4r} \rfloor}} \geq \frac{L_\mathbb{R}}{4 \cdot 2^{\lg \frac{L_\mathbb{R}}{4r}}} = r$. □

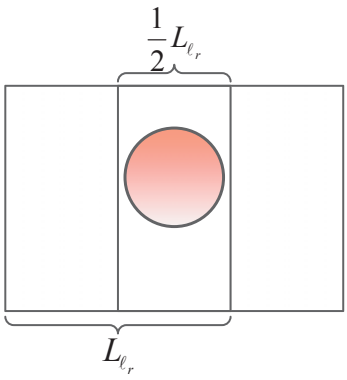


Figure A.1: Overlap of two neighboring r-related nodes

A.7 Proof of Lemma 5.2

Proof. Based on Definition 5.2, a r-related node is a node at level ℓ_r, $\ell_r = \lfloor \lg \frac{L_\mathbb{R}}{4r} \rfloor$, or a leaf node at a higher level ℓ'_r, $\ell'_r < \ell_r$. Let us now prove Lemma 5.2 by considering the two kinds of r-related nodes. For convenience, we use L_ℓ to denote the breadth of the corresponding rectangular region of node N at level ℓ.

Let us first consider a r-related node N at level ℓ_r, when $r \in (\frac{L_{\ell_r}}{8}, \frac{L_{\ell_r}}{4}]$. Then, the breadth of each partitioned grid is $\frac{L_{\ell_r}}{4\kappa}$. For any circle of radius r, $r \in (\frac{L_{\ell_r}}{8} + \frac{i-1}{8\kappa}, \frac{L_{\ell_r}}{8} + \frac{i}{8\kappa}]$, in $N.R$, we need to prove that there always exists such a region R_i covering $(\kappa+i+1) \times (\kappa+i+1)$ grids that encloses the circle. Let us prove it by contradiction. We assume that there does not exists such a circle. That is, there exists such a circle that spans two neighboring covered regions. Then we have that $2r > (\kappa + i)\frac{L_{\ell_r}}{4\kappa}$ since the maximum overlap of two neighboring covered regions contains $(\kappa + i) \times (\kappa + i)$ grids. However, we can deduce

the contradiction that $r > \frac{L_{\ell_r}}{8} + \frac{i}{8\kappa} = (\kappa + i)\frac{L_{\ell_r}}{4\kappa} \geq r$. Therefore, the i-th score $N.\varepsilon_i$ is the tightest score if $r \in (\frac{L_{\ell_r}}{8} + \frac{i-1}{8\kappa}, \frac{L_{\ell_r}}{8} + \frac{i}{8\kappa}]$.

Subsequently, let us consider a r-related leaf node N' at a high level ℓ'_r, $\ell'_r < \ell_r$, the first score $N'.\varepsilon_1$ is the tightest score since any circle of radius r can be enclosed by a region covering $(\kappa + 2) \times (\kappa + 2)$ grids. □

A.8 Proof of Lemma 5.3

Proof. Let us prove Lemma 5.3 in the two following cases: (1) $|c| = 1$ and (2) $|c| > 1$.

We first discuss about Case 1. In such case, this lemma will be reduced to Lemma 5.1.

Then we focus on Case 2. Now we attempt to prove it by generating a parallel state for any candidate result group. Based on Definition 5.1, any candidate result group will be within a circle of radius r. Consider a candidate result group G within a circle of radius r, denoted as $Circle_G$. Based on Lemma 5.1, there exists at least one r-related node N^{c_i} in T_{c_i} that encloses $Circle_G$. Now we present the generation of the parallel state, by identifying r-related nodes from these corresponding OQ-trees $T_{c_1}, \cdots, T_{c_n}$.

For the first keyword concept $c_1 \in \psi$, let N^{c_1} be the r-related node whose corresponding region encloses $Circle_G$. We set R as $N^{c_1}.R$, and set the current lowest level ℓ as the level of N^{c_1}.

For the second keyword concept $c_2 \in \psi$, there could exist several r-related nodes that enclose $Circle_G$. If there exists a r-related node that encloses the region R, we set N^{c_2} as such node. Otherwise, all of these r-related nodes are at lower levels. We set N^{c_2} as anyone r-related node, and update R as N^{c_2}.

In the same way, we then incrementally identify N^{c_i} for other keyword concepts, and the parallel state $\langle R, \{N^{c_1}, \cdots, N^{c_n}\}\rangle$ can be generated. □