6-2014

Recommendation Support for Multi-Attribute Databases

Jilian ZHANG
Singapore Management University, jilian.z.2007@phdis.smu.edu.sg

Recommendation Support for Multi-Attribute Databases

JILIAN ZHANG

SINGAPORE MANAGEMENT UNIVERSITY

2014

Recommendation Support for Multi-Attribute Databases

by

Jilian Zhang

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Information Systems

Dissertation Committee:

Singapore Management University
2014

Recommendation Support for Multi-Attribute Databases

by

Jilian Zhang

Abstract

This dissertation studies the subject of providing recommendation support for multi-attribute databases. Recommendation is an important and very useful information evaluation mechanism that explores a database of huge volume, and retrieves from it the interesting data items (tuples) for users based on their preferences. As a powerful tool for information filtering, recommendation systems find applications in product promotion, search-engine result ranking, multiple-criteria decision making, etc. As basis for the recommendations, we consider relational databases that contain items with multiple attributes, among which we focus on the numerical ones that measure various quantitative features. We call this type of relational databases *multi-attribute databases*.

We consider three different yet highly related recommendation tasks and center this dissertation on the database techniques needed to support those tasks. In the first task, users represent their preferences on some attributes as a hyper-rectangle, i.e., query window, and the recommender system returns those items whose attribute values overlap with or are covered by the query window. We call this task *preference-overlap recommendation*. The second task is *top-k recommendation*. Here, users express their preferences in the form of weights on attributes, and the system employs some (usually monotone) scoring function to find the k items with the highest scores on these attributes. The third task relates to *group recommendation*, where the objective is to recommend items, activities, services, etc, to a group of users with diverse preferences.

To enhance the first recommendation task, we identify an interesting co-occurrence relationship between data items. Specifically, we propose the concept of *direct neighbor* (DN). Given a query object q, an item p is a DN of q if there is some query window that exclusively retrieves q and p. Users can derive valuable information from DNs, because they represent competing alternatives to q. We extend the DN notion to two variants, namely k-DN and All-DN. We devise novel and I/O optimal algorithms for DN, k-DN, and All-DN computation. Due to its semantics of co-existence, the DN query finds various applications such as competitor analysis, alternative recommendation, and spatial index optimization.

For the second task, we render support for top-k recommendation by introducing the notion of *global immutable region* (GIR). The GIR with respect to a top-k query q is the maximal locus in query vector space, where the top-k result retrieved by any query vector q' in the locus remains the same as q. GIR is a sensitivity indicator that tells the users how robust the recommendation is and what the alternatives are besides the recommended results. We also introduce two variants, namely order-insensitive GIR and GIR for non-linear scoring functions. To derive GIRs efficiently, we propose novel algorithms for pruning the majority of database items from consideration.

In the third task, we consider support for group formation, which is an important problem in group recommendation. We formulate the group formation problem as a bucketization problem, or equivalently a balanced multi-way number partitioning (BMNP) problem in Artificial Intelligence. BMNP is NP-hard in nature. We propose three heuristic algorithms to find optimal grouping solutions, which can achieve a theoretical performance gain of two-thirds over the existing state-of-the-art algorithm.

Our work in this dissertation provides support to the above three recommendation tasks, by giving to the users additional and valuable information, along with the result returned by conventional recommender systems. Our techniques find various applications such as multiple-criteria decision making, competitor analysis, product promotion, and sensitivity analysis, etc.

Contents

List of Figures

List of Tables

Publications related to the Dissertation

Listed in reverse chronological order:

1. Jilian Zhang, Kyriakos Mouratidis and HweeHwa Pang. Global Immutable Region Computation. Accepted to *ACM SIGMOD International Conference on Management of Data*, 2014. (Chapter 4)

2. Jilian Zhang, Kyriakos Mouratidis and HweeHwa Pang. Direct Neighbor Query. Accepted for publication in *Information Systems* (IS), 2014. (Chapter 3)

3. HweeHwa Pang, Jilian Zhang and Kyriakos Mouratidis. Enhancing Access Privacy of Encrypted B^{+}-Trees. *IEEE Transactions on Knowledge and Data Engineering* (TKDE), 2013.

4. Jilian Zhang, HweeHwa Pang and Kyriakos Mouratidis. Heuristic Algorithms for Multi-way Number Partitioning. *Proceedings of the International Joint Conference on Artificial Intelligence* (IJCAI), 2011. (Chapter 5)

Acknowledgements

The completion of this dissertation would not be possible without the assistance given to me by several people to whom I feel very grateful. I would like to acknowledge them here.

First of all, I would like to express my gratitude to my supervisor Professor HweeHwa Pang, who has provided the opportunity for me to pursuit a Ph.D in School of Information Systems, Singapore Management University (SMU). I feel very fortunate to be under the supervision of Professor HweeHwa Pang, who not only academically guided me in every detail, but also enlightened me on how to work as a professional researcher. It is not easy to measure the benefits I have received, but I fully comprehend that over the years they have enabled me to achieve what I have today. I am very grateful to Professor HweeHwa Pang.

I owe a lot to my co-supervisor Associate Professor Kyriakos Mouratidis, for the tremendous help and very detailed academic guidance he has given me. I feel lucky to have had the chance to work closely with Associate Professor Kyriakos Mouratidis, whose talent and professional spirit have always inspired me so much. I am grateful to Associate Professor Kyriakos Mouratidis .

I am thankful to Professor Ee-Peng Lim, Professor Steven Miller and Professor Stephen E. Fienberg for establishing the Living Analytic Research Center (LARC), which provided me with a great opportunity, and a scholarship as well, to visit Carnegie Mellon University (CMU) from August 2011 to June 2012. I also want to thank Professor Steven E. Fienberg and Professor Ramayya Krishnan, for supervising me during my visit to CMU.

Meanwhile, I want to thank Professor HweeHwa Pang, Associate Professor Kyriakos Mouratidis, Professor HoongChuin Lau, Professor Robert H. Deng, Associate Professor Xuhua Ding and Assistant Professor Jialie Shen, for giving me the opportunity to take their insightful courses or to work together on research projects and papers.

I would like to thank my dissertation committee members for reviewing my dissertation and providing comments and feedback: Professor HweeHwa Pang, Associate Professor Kyriakos Mouratidis, Professor HoongChuin Lau, and Professor Nikos Mamoulis from The University of Hong Kong (HKU).

I also want to thank the following staff from School of Information Systems and from LARC, for the administrative support during my Ph.D study in SMU and my visit to CMU: Ong Chew Hong, Seow Pei Huan, Chua Kian Peng, Alenzia Wong Poh Luan, Angela Kwek Renfeng, Fong Soon Keat, Nancy Beatty and Ashley Ferenczy.

Dedicated to Mum, Dad, My Wife, and My Son.

Chapter 1

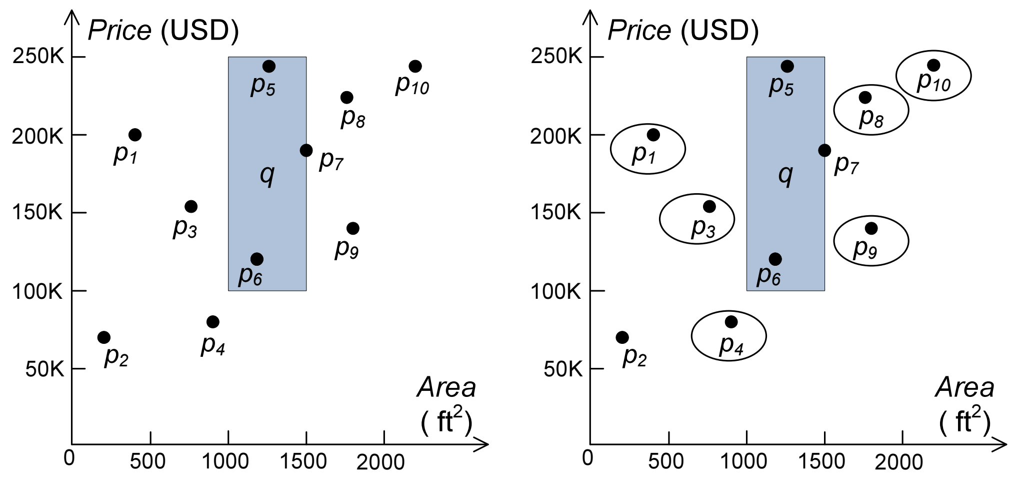
Introduction

With the progress and innovation on web and mobile technologies, users now have easy access to services such as online shopping, financial investment, restaurant searching, and job hunting, provided by major companies like Amazon, Yahoo!, Yelp, Monster.com, etc. These merchants manage huge amounts of information on their products, restaurants, and services into *relational databases* by recording textual attributes such as reviews, as well as numerical attributes such as price, commission rate, salary, etc. Here, a recorded data item is also known as *tuple*, and the database is called a *multi-attribute database.*

On the database, the companies employ intelligent *recommender systems* to identify the most interesting products, stocks, restaurants, hotels, or jobs for the users, according to their *preferences.* This process is called *recommendation*, and the items returned to the users are collectively referred to as *recommendation result* or simply *recommendations.*

Recommender systems are important and powerful information filtering tools, providing the users with a fast and reliable way to explore possibly huge databases, so as to efficiently identify the exact information they need or help significantly narrow down the amount of information to investigate.

Despite the current success of recommender systems, their usability can be enhanced and extended. For instance, the users may want to know more



(a) Recommend tuples covered by window q (b) Find alternatives (the circled points)

Figure 1.1: Example for preference-overlap recommendation

about the impact of their preference queries, about the items recommended to them, or about the alternatives not shown in the recommendation list. In other words, recommendations do not have to be limited to only showing to the users the recommended result items.

We consider three recommendation tasks, namely preference-overlap recommendation, top-k recommendation, and group recommendation, and elaborate on how we can provide support to enhance them.

1.1 Preference-Overlap Recommendation

Consider an online property agent who maintains a database storing information of the houses that are under its management, including attributes such as area, price, distance to downtown, etc. A potential customer Alice, in searching the database, may express her preferences quantitatively by specifying ranges on these attributes, for example, ‘area between 1000 square feet and 1500 square feet’, and ‘price between \$100K and \$250K’.

Upon receiving the preference parameters from Alice, the agent searches the database, with the help of data index structures, like the R-tree [36], and recommends to Alice those houses with area and price attributes falling simultaneously in the ranges of [1000,1500] for area and [100K,250K] for price.

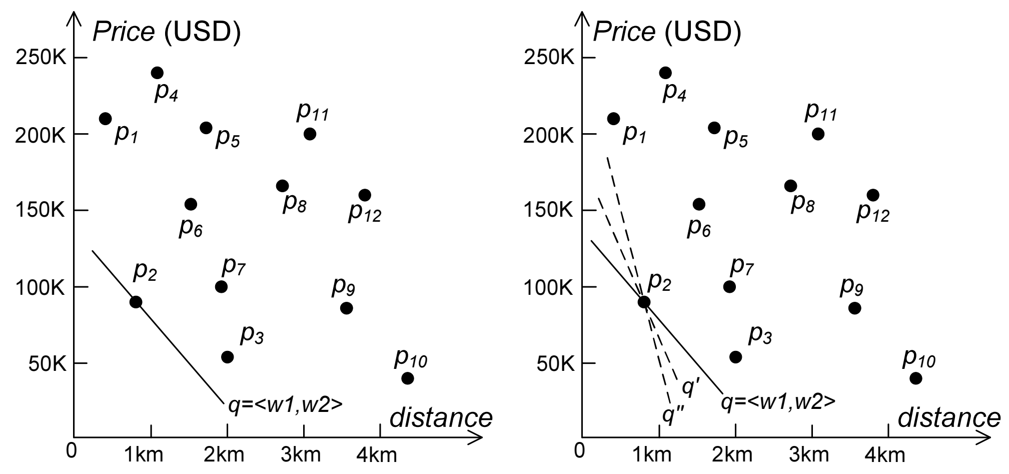
Geometrically, these preference ranges correspond to a rectangular *query window* in the 2-dimensional 'area-price' space (see Figure 1.1(a)). Similarly, a high-dimensional preference query window, in the form of a *hyper-rectangle*, can be defined by taking into account more than 2 attributes at the same time [61].

We call this recommendation task *preference-overlap recommendation*, because here the problem is to scrutinize the tuples in the database, so as to find those with attribute values that directly cover or that overlap the user's preference ranges. For example, in Figure 1.1(a), p_5, p_6 and p_7 fall inside the query window, thus they are reported as the result. Preference-overlap recommendation is a simple yet very common recommendation problem, and closely relates to the multidimensional range query [11] and the nearest neighbor query [64, 39]. We review details of this recommendation task in Chapter 2.

Support for Preference-overlap Recommendation

We consider how to support preference-overlap recommendation. Let us continue the example of Alice in searching the property agent's database for interesting houses. The recommendation list consists of houses directly matching Alice's preference ranges. However, there may be houses outside of the recommendation list that are very similar (in terms of area and price) to a recommended one that could be retrieved exclusively with the latter by some preference ranges. In Figure 1.1(b), for example, the circled points are all competing alternatives. These alternative houses are direct competitors to the recommended ones, and may also be of interest to Alice.

To capture this additional information, we introduce the concept of *direct neighbor* (DN). Specified an object q, a data object p in the database is a DN of q if there exists some query window that exclusively retrieves, i.e., covers or overlaps, q and p. We propose to compute the DNs for each result tuple q in the recommendation list, which form a collection of possible alternatives that



(a) Recommend top-1 result p_2 to the user (b) Find all alternative vectors, e.g., q', q'', such that p_2 remains as the top-1 result

Figure 1.2: Example for top-k recommendation

would be valuable for the users' decision making.

1.2 Top-k Recommendation

Unlike preference-overlap recommendation that considers coverage and overlap relationships between the user's preference ranges and the tuples, top-k recommendation ranks the tuples based on their *score*, computed with a *scoring function* on the attribute values of the tuples.

Continue our earlier example on property search. Alice prefers houses near to downtown, but she has a relatively tight budget. Thus, she may want to find a trade-off between price and location, by specifying her query with preference vector $q =< w_1, w_2 >$, where w_1 and w_2 are numerical *weights* on the distance and price attributes, respectively. Alice may indicate (1) a smaller w_1, meaning that she likes houses nearer to the downtown, or (2) a smaller w_2, meaning that she prefers cheaper houses. Meanwhile, she may only want to see a few, say the top 20, of the most relevant houses. Based on these preferences, the agent could compute the score of all the houses in the database, by taking the aggregate $w_1 * distance + w_2 * price$, and return to Alice as recommendation result the 20 houses with the smallest scores.

Figure 1.2(a) illustrates an example of top-1 recommendation, where the scoring function $w_1 * distance + w_2 * price$ corresponds to a straight line called *sweeping line*, and preference vector q is a normal vector to the line. The sweeping line sweeps the data space from the origin to the top-right corner, and the first point encountered, i.e., p_2, is the top-1 result.

Due to its rich semantics, top-k recommendation has received much attention from industry and academia, and has been employed in applications such as multiple-criteria decision making, search-engine result ranking, product and service recommendation, financial investing, etc [40]. We detail the related work on top-k recommendation in Chapter 2.

Support for Top-k Recommendation

Here, we show how to provide support to top-k recommendation. We continue with the example of Alice using preference vector $q =< w_1, w_2 >$ to look for the top-20 houses. Suppose that in addition to the 20 houses recommended, Alice wants to know whether there are competing houses not in the top-20 list, yet having the potential to overtake some houses in the top-20 list. Alice may also wonder whether this top-20 list will remain the same if she has some flexibility in her budget. For example, as shown in Figure 1.2(b), Alice may want to find all the alternative preference vectors, e.g., q' and q'', such that p_2 remains as the top-1 result with respect to these vectors. These questions are equivalent to the problem of how the changes in preferences will affect the recommendation result.

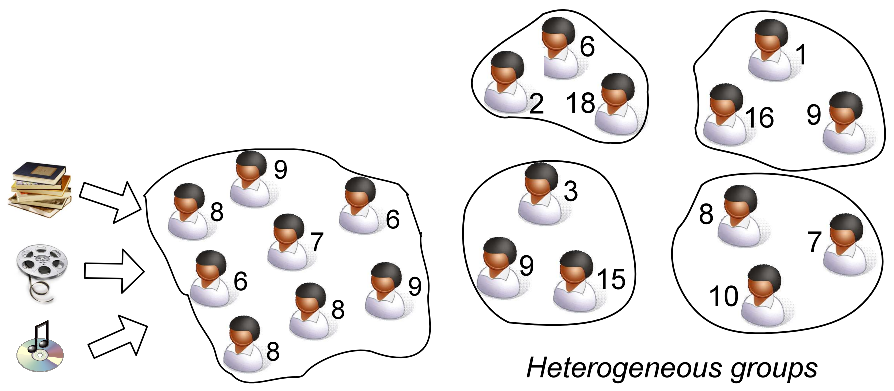
To answer the problem, we propose the concept of *global immutable region* (GIR) for top-k recommendation. Given a top-k query $q =< w_1, w_2 >$ by the user, the GIR of q is the maximal locus consisting of all preference queries q', such that the top-k result with respect to q' is the same as q. For any query outside the GIR, the top-k list will change. The GIR tells Alice the extent to which fluctuations are allowed on w_1 and w_2, such that the top-20 list remains unchanged. GIR can be used as a sensitivity measure [66] to evaluate how robust the recommendation result is when perturbation occurs in the user's original preferences. Furthermore, as we will see later, the boundary of the GIR tells Alice what the competing alternative houses are. The additional information carried by GIR could help Alice to make a wise purchase decision.

Top-k result sensitivity has been considered recently in [70] and [53]. [70] proposes a sensitivity measure called STB for top-k queries, which is a maximal ball containing query vectors that preserve the top-k result. STB is a subset of our GIR, meaning that the information provided by STB is less comprehensive. [53] introduces a sensitivity measure which we call *local immutable region* (LIR). LIR is local in the sense that it only gives an adjustable range on one query weight, while keeping the remaining weights fixed. This feature of LIR hinders its usefulness and does not permit concurrent adjustments in multiple query weights.

1.3 Group Recommendation

The group recommendation task is to recommend relevant items to a group of users, as shown in Figure 1.3(a), according to their preferences, skill set, knowledge level, etc [3, 54]. If a group comprises users of diversified features, it is a *heterogeneous group* [71]. There are two problems involved in group recommendation - how to form groups if the group formation plan is not available in advance, and how to find relevant items for groups. We focus on the group formation problem.

Consider a school organizing a cohort of students for a study trip, say, to visit a science laboratory, or to observe the operations in a factory. The students may have different levels of interest, background knowledge, etc. Moreover, the school needs to divide the students into groups of certain size so as to abide by regulations, such as maximal capacity constraint per group, imposed



(a) Recommend items to a group (b) Form heterogeneous groups

Figure 1.3: Example for group recommendation and group formation

by the laboratory or factory. The school also wishes to mingle the students so as to promote *peer learning* [17] among the students within each group. If the school forms the groups by strategically mixing students of high level of background knowledge with students of low level knowledge in heterogeneous groups, then the students in the same group may greatly benefit from each other through interactions and peer learning, as revealed by a study on peer learning effectiveness [41].

The objective of heterogeneous group formation problem is to design a viable partitioning strategy, so as to satisfy certain tangible constraints, such as capacity, intra-group diversity, and intangible ones, such as promoting peer learning. We consider a sub-problem of the heterogeneous group formation problem, by omitting the intra-group diversity constraint while forming the groups. This sub-problem is closely related to balanced number partitioning, which is an important problem in Artificial Intelligence and has many applications. Unless specified otherwise, in the following we also refer to this sub-problem as heterogeneous group formation problem. A review of the group formation problem and group recommendation is given in Chapter 2.

Support for Group Recommendation

Consider our earlier example of a school organizing students for a study trip. To provide a heterogeneous group formation plan, we have to take into account the capacity constraint on the maximal number of students allowed in each group to enter the laboratory or factory. This means that we need to populate each group with the (roughly) same number of students, which is the maximal capacity, so as to minimize associated costs, such as total visit time, transportation fee, etc.

On the other hand, hoping to achieve a good peer learning outcome, we want to evenly distribute students with various quantitative features among the groups, such that a collective measure, say aggregation, on one of the students' features, e.g., knowledge level, is as similar as possible across all groups. Note that aggregation is one of the most common ways to compute a collective measurement for groups in the group recommendation problem [3].

Figure 1.3(b) shows an example of heterogeneous groups formation, where each group contains the same number of users, and the numbers at the side of each user represents her levels of interest, background knowledge, etc. Although users within a group have diversified numbers, the sums of numbers across groups are roughly the same.

We treat this heterogeneous group formation problem as a *bucketization problem*, which is equivalent to the *balanced multi-way number partitioning problem* (BMNP) in Artificial Intelligence [34]. The objective of BMNP is to bucketize a set of numbers into groups of the same capacity, such that each group is fully packed with the numbers and the sum of numbers in a group is as close as possible to every other group.

1.4 Contributions

We focus on three different recommendation tasks, namely preference-overlap recommendation, top-k recommendation, and group recommendation. We consider how to support these tasks, by finding additional and useful information that is missing in conventional recommender systems. Our contributions in this dissertation are summarized as follows.

1. We introduce the concept of *direct neighbor* (DN) for preference-overlap recommendation, and propose a novel query type called direct neighbor query that retrieves the DNs with respect to a user-specified query object. DN relationship has not been investigated before, and it is useful in various applications such as multiple-criteria decision making, marketing, competitor analysis, etc. We propose two novel algorithms, SDN and CNS, to efficiently compute DNs in large databases. We also study interesting extensions of DN query, namely the k-DN query and the All-DN query.

2. We propose *global immutable region* (GIR) for top-k recommendation. Compared to the LIR proposed in [53], GIR is more useful in that it simultaneously captures all the permissible settings of query weights that represent the user's preferences. To efficiently compute GIR, we propose three novel algorithms, namely SP, CP, and FP, based on skyline processing and convex hull computation. FP is superior to SP and CP, due to the fact that FP prunes the majority of the data tuples from consideration. We also expand the problem space by proposing two interesting extensions of GIR, i.e., order-insensitive GIR and GIR for non-linear scoring functions, and show how to compute these GIR variants. GIR can be used in applications such as sensitivity analysis, stock marketing monitoring, top-k result caching, search-engine result ranking, etc.

3. We investigate the bucketization problem for heterogeneous group formation. We relate the problem to the balanced multi-way number partitioning (BMNP) and propose three novel algorithms, LRM, Meld, and Hybrid. Compared to existing work [51] on BMNP, our LRM algorithm achieves the best performance on datasets with uniform distribution, while Meld is the best on datasets with skewed distribution. When we

have no prior knowledge of the data distribution, our Hybrid algorithm can be applied, which combines the strengths of LRM and Meld algorithms.

1.5 Organization of the Dissertation

The work in this dissertation is an aggregation of several research papers we have published. We organize the papers in this dissertation as follows. In Chapter 2 we review existing work on recommender systems and the three recommendation tasks, namely preference-overlap recommendation, top-k recommendation, and group recommendation. In Chapter 3 we study the problem of providing support to preference-overlap recommendation, and propose to compute direct neighbors to capture additional useful information. In Chapter 4 we investigate the global immutable region for top-k recommendation, and highlight its usefulness as a sensitivity measure. In Chapter 5 we consider the bucketization problem for heterogeneous group formation. Finally, we conclude this dissertation and discuss directions for promising future work in Chapter 6.

Chapter 2

Related Work

In this chapter, we review related concepts and prior work to this dissertation. For now we only survey general related work on the three recommendation tasks, namely preference-overlap recommendation, top-k recommendation, and group recommendation. Detailed discussions on more specific prior work are given in their respective chapters.

We first review the classical concept of recommender systems and some existing models. We then describe prior work on preference-overlap recommendation, covering the topics of range query processing and nearest neighbor query processing. We also review related work on top-k recommendation. We end this chapter by reviewing prior work on group recommendation.

2.1 Recommender Systems

Recommendation is one of the most frequent activities in our daily life: recommend a book to a close friend, recommend a restaurant to a family, recommend financial services to an investor, etc. Various recommender systems have been designed and deployed. As an information filtering mechanism, a recommender system finds, from a possibly very large database, a subset of the most relevant data items to the users according to their preferences.

Since the first research work on collaborative filtering in mid-1990s [62], recommender systems have attracted a lot of attention from industry and academia. Many recommendation techniques have been proposed since then, which can be broadly divided into content-based recommendation techniques and collaborative recommendation approaches [2].

Content-based recommendation techniques recommend to a user items similar to the ones that the user has shown preference for in the past. Content-based recommendation techniques take into account numerical, sometimes even textual information of the items [2]. Clustering and decision trees are two commonly used techniques for content-based recommendation [38].

Collaborative recommendation approaches provide a user with items that were preferred in the past by other people sharing similar taste with the user [59]. This type of recommendation systems is promising and has been adopted by Amazon, Yelp, IMDB, etc, because they utilize information which is neglected by the content-based recommendation systems. Collaborative filtering (CF) [2] is one of the most commonly used collaborative recommendation approaches.

The three recommendation tasks we considered in this dissertation share some similarities with the above two categories of recommendation systems. The first task, preference-overlap recommendation, belongs to content-based techniques, because it considers items that are 'similar' to the preference ranges specified by the users. The second task, top-k recommendation, relates to content-based recommendation systems, because it retrieves for a user the k most similar items from the database according to her preferences and the 'content', i.e., attributes, of the items. The third task, group recommendation, has connections with both categories of recommendation systems, because it takes into account other users' quantitative features, such as preferences, knowledge level, etc. We review related work on the three in the sections that follow.

2.2 Preference-Overlap Recommendation

Preference-overlap recommendation is the simplest type of recommendation, where the problem is to indicate data items, i.e., tuples, whose attribute values overlap with or are fully covered by users' preference ranges on these attributes. Preference-overlap recommendation is closely related to range query and nearest neighbor query – the former retrieves tuples falling in a user-specified hyper-rectangle on some attributes of interest, while the latter retrieves data items that are closest to the one specified by the user. These two are fundamental query types in database systems, and have received extensive attention [61].

Range query originated from the *multikey searching problem* or *multidimensional searching problem* in the 1970s and 1980s, which finds applications in many fields such as database management, statistics, physics, and design automation [11, 56]. Consider a database D in d-dimensional space, and a range query $q = ([l_1, u_1], [l_2, u_2], \ldots, [l_d, u_d])$ where l_i and u_i are the lower and upper bounds on the i-th attribute of D, respectively. Geometrically, this multi-dimensional range q corresponds to a hyper-rectangle. The objective of range query processing is to report from D those tuples $r = (x_1, x_2, \ldots, x_d)$, such that $l_i \leq x_i \leq u_i$ for $i = 1, 2, \ldots, d$; that is, r falls within hyper-rectangle q.

To efficiently answer range queries, various data structures have been proposed, such as B$^+$-tree [25], KDB-tree [63], R-tree [36], etc. B$^+$-tree is one of the most fundamental index structures in database systems, and is used for efficient 1-dimensional range search for tuples whose search keys are covered by the user's query range. For high-dimensional range search, KDB-tree is a multidimensional index structure that iteratively partitions the data space into multiple hyper-rectangles in a hierarchical manner. R-tree is another popular multidimensional index structure, which groups nearby objects together and represents them with *minimum bounding rectangles* (MBRs). These MBRs are treated as objects and grouped in a similar manner in the next higher level of the tree.

Range query processing on these multidimensional indices is straightforward. Given a query range q, starting at the root we recursively check the nodes down the tree in the following manner. At some internal node of the tree, if an entry of the node overlaps with q, we recursively visit the subtree below this entry. Otherwise we remove the entry from consideration. Whenever we encounter a leaf node, the exact information of its data entries is fetched from the disk file and checked against q. Those data entries that fall inside q or overlap with q are reported in the query result.

A *nearest neighbor* query (NN) retrieves from a database the object that lies closest to a user-specified source point q. For NN processing over datasets indexed by a spatial access method, the *depth-first* and *best-first* paradigms have been considered in [64] and [39], respectively. The latter is shown to be superior in I/O cost. Assume that the dataset is indexed by an R-tree. Starting from the root of the R-tree, encountered index entries e are pushed into a min-heap with $mindist(e, q)$ as the key, i.e., the minimum distance between the source point q and the MBR of e. Iteratively, the top entry of the heap is popped and its corresponding R-tree node is accessed from disk; then, its child entries are en-heaped. The process is repeated until the first data entry (object) is popped and reported as the NN. If more nearest neighbors are required, the process continues and the next data entry popped is the second NN, and so on. The method is *incremental* in that it can keep reporting the next NN without needing to specify in advance how many neighbors are required in total. Also, it is I/O optimal, i.e., it fetches from disk the minimum possible number of R-tree nodes.

2.3 Top-k Recommendation

Top-k recommendation, or equivalently top-k query processing, is a powerful information exploration and filtering mechanism that provides users with the k most important tuples from a potentially huge database. In the past decade, various top-k recommendation models have been proposed and successfully used in applications such as product recommendation, search-engine result ranking, multimedia search, etc [40, 32, 5, 75, 42, 52]. Among the top-k query models, in this dissertation we consider the top-k selection query (for simplicity we omit 'selection' in the following).

In a top-k query model, users express their preferences in the form of weights on the tuple attributes. Each tuple is implicitly associated with a score. The score value is given by an (aggregate) scoring function that incorporates the user-specified weights and the tuple's attribute values [40]. Consider a database D of N tuples $r = (x_1, x_2, \ldots, x_d)$ in d-dimensional space and a top-k query with weight vector $q = (w_1, w_2, \ldots, w_d)$, where w_j is the user's preference on the j-th attribute. The scoring function $f : D \mapsto \mathbb{R}$ maps each tuple in D to a real value called *score*. The most common type of scoring functions is linear, i.e., functions of the form $f(r) = \sum_{i=1}^{i=d} w_i x_i$. The system ranks all the tuples in D according to their scores, and recommends the top-k tuples to the user.

Many efficient techniques have been proposed for top-k query processing. Among them, BRS [73] is a top-k algorithm for low-dimensional data indexed by a spatial access method (e.g., an R-tree [36]). Designed for the broad class of monotone scoring functions, BRS applies the *branch-and-bound* methodology. Specifically, it uses a max-heap to organize the entries of visited R-tree nodes so as to access them in decreasing order of their *maxscore*. The *maxscore* of an R-tree node is the largest among the scores of its MBB corners (MBB stands for the node's minimum bounding box) and serves as an upper bound for the score of any record under the node. When a leaf node is accessed, the score of the records inside are computed and the interim top-k result is updated accordingly. BRS terminates when the record with the k-th largest score in the interim result has a score no smaller than the *maxscore* of the last R-tree entry popped from the search heap. BRS is I/O optimal, meaning that it reads the minimum possible number of pages (R-tree nodes) from the disk.

The *threshold algorithm* (TA) [32] is another popular method for top-k query evaluation for queries with monotone scoring function f. Given a database D, TA maintains d lists, where each list L_i corresponds to D sorted on the i-th attribute in descending order. During query evaluation, TA probes the lists in a round-robin fashion, accessing tuples from the top (i.e., with the highest attribute value) to the bottom (i.e., with the smallest attribute value) of the list. For each tuple r encountered in a list, its complete information is fetched via *random access* by looking through the lists or the disk file holding D, so as to compute the score $f(r) = \sum_{i=1}^{i=d} w_i x_i$, where w_i is the i-th component of user query vector q. The k tuples with the highest scores encountered so far are kept in a temporary set $\mathcal{R}$, in descending order on their scores. Let t_i be the next attribute value to access in list L_i, $i \in [1, d]$, and $t^* = (t_1, t_2, \ldots, t_d)$ be a fictitious tuple. TA terminates when the score $f(t^*)$ is smaller than that of the k-th tuple in $\mathcal{R}$, where $\mathcal{R}$ is the top-k result set to report upon termination. Here t^* plays the role of threshold for TA to stop.

Top-k query processing is a broad research area that addresses different topics, depending on the type of data, constraints on the result quality, and the context of applications. For example, there are studies on top-k joins [75], top-k monitoring for dynamic databases [52], distributed top-k computation [5], and top-k evaluation on uncertain data streams [42], to name a few. We omit the details of these techniques, however, because our work in Chapter 4 of this dissertation centers on exact top-k query processing on static data with monotone scoring functions. We employ BRS [73] in our top-k computation module, for its effectiveness and support for spatial index methods such as the

R-tree [36].

A related problem is the reverse top-k query [79, 78], which involves a database D and a collection of user preference functions represented as query vectors. A reverse top-k query returns those query vectors from the collection that include a given record $p \in D$ in their top-k result.

2.4 Group Recommendation and Group Formation

Different from the conventional recommendation model that serves a single user, group recommendation considers a group of similar users and makes recommendations for the group as a whole [3, 54]. Group recommendation is useful in applications such as suggesting a movie for friends to watch together, or choosing a travel destination for a family to visit, etc [71]. The problem of group recommendation is non-trivial, because sometimes the users within a group may be so different that their preferences, knowledge level, skill set, etc, conflict with each other. This type of groups are also called *heterogeneous groups* [71]. There are two major problems in group recommendation, that is, how to form the groups, and how to choose the relevant items for the groups.

Amer-Yahia et al. [3] define the semantics of group recommendation, and formulate the disagreement among group members. Ntoutsi et al. [54] propose to cluster the users based on their preferences, such that each group contains similar users. A group is recommended an item on which the aggregated preference score over all the users in the group is maximal, compared to the other items.

While the majority of existing work addresses the formation of groups with similar users, we focus on forming heterogeneous groups in this dissertation. As explained in the Introduction, in some applications heterogeneous groups can benefit group members, such as in promoting peer learning [17, 41]. To form heterogeneous groups, our objective is to partition into groups a set of users with different quantitative features, e.g., preferences and knowledge level, such that each group contains the same number of users and the aggregate quantitative features of the group members are as close as possible among the groups. Our heterogeneous group formation problem is essentially a bucketization problem, or equivalently a balanced multi-way number partitioning problem. In the following, we provide some background on the bucketization problem, by revisiting the classic number partitioning problem [34].

Given a finite set of positive integers $S = \{n_1, n_2, \ldots, n_k\}$, the number partitioning problem is to partition S into two subsets $S_1 = \{n_1^1, n_2^1, \ldots, n_{k_1}^1\}$ and $S_2 = \{n_1^2, n_2^2, \ldots, n_{k_2}^2\}$, such that the following conditions hold

1. $S_1 \cup S_2 = S$,
2. $\sum_{n_i^1 \in S_1} n_i^1 = \sum_{n_j^2 \in S_2} n_j^2$

The optimization version of the number partitioning problem is to modify the second condition by minimizing the difference between the two sums, that is, $\min\{\sum_{n_i^1 \in S_1} n_i^1 - \sum_{n_j^2 \in S_2} n_j^2\}$.

Number partitioning belongs to a category of NP-hard problems that has been studied extensively [34]. One of its variants is balanced multi-way number partitioning (BMNP). The input of BMNP is a set S of n numbers and a positive integer k; the output is a partition of S into subsets. The *subset sum* is the sum of numbers in a subset, and the *subset cardinality* indicates how many numbers it contains. The objective in BMNP is to partition S into k subsets such that (i) the cardinality of each subset is either $\lfloor \frac{n}{k} \rfloor$ or $\lceil \frac{n}{k} \rceil$ numbers, and (ii) the *spread* (i.e., the difference) between the maximum and minimum subset sums is minimized. BMNP has a wide range of applications, including multiprocessor scheduling [28] and VLSI manufacturing [74].

The KK algorithm [44] is the best approximate method for 2-way partitioning, generating a spread in subset sums in the order of $O(1/n^{\alpha \log n})$ for some constant α. [45] proposed CKK, an exact algorithm based on the principles of KK. CKK produces a global optimal solution for 2-way and multi-way partitioning, by exhaustively searching a binary tree that covers all possible combinations of subsets. However, none of the above algorithms can ensure a balanced partitioning, i.e., equal subset cardinalities. Furthermore, the exact algorithms like CKK are applicable only to small problem sizes.

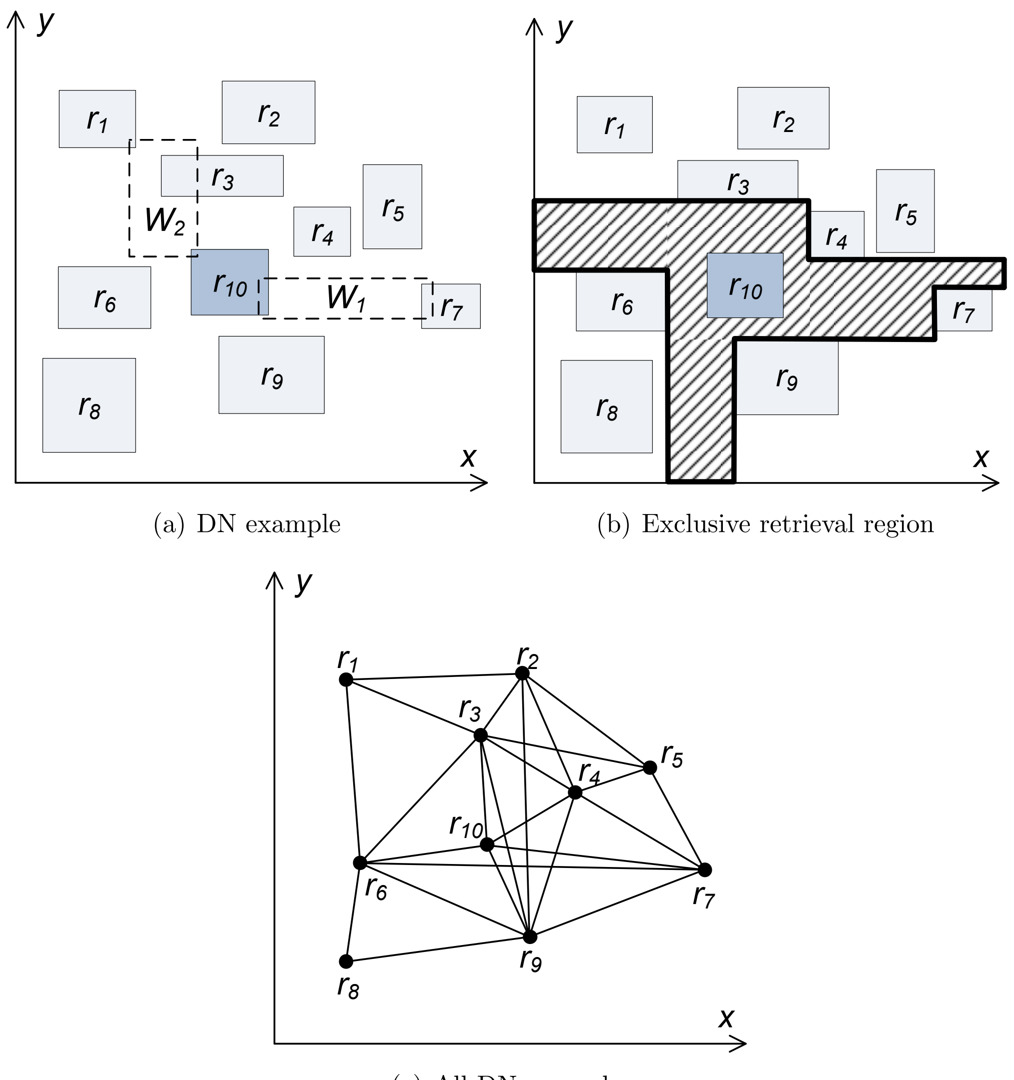
The *balanced largest-first differencing method* (BLDM) is a modification of KK that guarantees the cardinality of the two subsets produced to be $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ [81]. There have been several attempts to extend BLDM from balanced 2-way to balanced multi-way number partitioning. In [51], a generalized BLDM, with a time complexity of $O(n \log n)$, is proposed to perform balanced k-way partitioning for $k > 2$. BLDM, however, does not perform well on datasets with skewed distributions, or datasets with normal distribution and the number of partitions being odd at the same time.

Chapter 3

Direct Neighbor Search for Preference-Overlap Recommendation Support

In this chapter we study a novel query type, called *direct neighbor* query, which can provide additional and valuable information to the users for recommendation support. Two objects in a dataset are direct neighbors (DNs) if a window selection may exclusively retrieve these two objects. Given a source object, a DN search computes all of its direct neighbors in the dataset. The DNs define a new type of affinity that differs from existing formulations (e.g., nearest neighbors, nearest surrounders, reverse nearest neighbors, etc) and finds application in domains where user interests are expressed in the form of windows, i.e., multi-attribute range selections.

Drawing on key properties of the DN relationship, we develop an I/O optimal processing algorithm for data indexed with a spatial access method. In addition to plain DN search, we also study its K-DN and all-DN variants. The former relaxes the DN condition – two objects are K-DNs if a window query may retrieve them and only up to $K-1$ other objects – whereas the all-DN variant computes the DNs of every object in the dataset. Using real, large-



(a) DN example (b) Exclusive retrieval region

(c) All-DN example

Figure 3.1: DN and all-DN search

scale data, we demonstrate the efficiency and practicality of our approach, and show that it vastly outperforms a competitor constructed from previous work.

3.1 Motivation

We focus on systems and applications where users browse databases via window queries. Consider a database where objects correspond to available services or products and are represented as rectangles in a d-dimensional space. A window query retrieves all the objects that fall inside a user-specified axis-parallel rectangle. Fig. 3.1(a) illustrates a database with 10 objects in two-dimensional space. W_1, shown with a dashed border, is an example of a window query that returns r_7 and r_{10} in the result.

Alternative query types, such as nearest neighbors (NN) [64] and reverse nearest neighbors (RNN) [46], browse data based on the notion of spatial distance, provided that objects bear geographic coordinates. Inherent in the distance notion is the assumption that different dimensions can be combined in a predetermined way into Euclidean distance or another L^p metric.

However, in many settings the data dimensions represent different aspects of the problem and are not directly comparable to each other. Thus, it is not meaningful to combine dimensions into a distance measure in determining the similarity between objects. In such settings, window queries are the only reasonable representation of user interests. An example is on-line property agencies like propertyguru.com.sg, on which owners, agents and developers post details of units for rental/sale. Potential buyers/tenants may browse available options by specifying ranges of their desired price and floor-area requirements (i.e., via window queries). For instance, in Fig. 3.1(a) the two dimensions could correspond to the rent and floor area, respectively. Another example is kayak.com. In this portal, users planning to fly between two cities may browse the available flight options by specifying acceptable ranges for the price and duration of the flight.

Since user interests are captured by window queries, similarity ought to be defined based on windows and, specifically, *on the potential of data objects to co-exist in the same query result.* Assume that the objects in Fig. 3.1(a) correspond to alternative services/products. To identify the immediate alternatives to r_{10} (called the *source*), its provider/manufacturer would want to know which objects are likely to be retrieved together with r_{10} by user queries. Consider alternatives r_3 and r_1. On one hand, there exist windows that would retrieve only r_3 and the source (r_{10}). On the other hand, for a query to report the source and object r_1, it must necessarily report r_3 as well. In this aspect,

r_3 is a more immediate competitor/alternative to r_{10} than r_1. To capture this fact, we define direct neighbors as follows.

Definition 1 *Given a dataset S, we define as **direct neighbor** (DN) of a source object q any other data object $r \in S$ which may be exclusively retrieved (along with q) by a window query. In other words, there exists an axis-parallel window that overlaps only with q and r.*

A DN query at source q retrieves all its DNs in S. In the example of Fig. 3.1(a), for source object r_{10}, the result comprises r_3, r_4, r_6, r_7 and r_9. Applications of the DN query include competitor and marketability analysis, recommendation of alternatives, etc.

Competitor Analysis: Identifying the DNs of q could be used to improve its competitiveness with respect to directly comparable products/services [33], e.g., via competitor-aware advertisement or appropriate reconfiguration/redesign of q itself. For instance, the marketing team behind a property (or the airline offering a flight) would be interested in knowing which its immediate competitors are with respect to the rent-area criteria (airfare-duration), and potentially reconsider its pricing.

In certain dimensions there may be a clear preference direction (i.e., higher/larger values may be more desirable). For example, in the property scenario one could assert that lower price (equivalently, larger size) is generally preferable. The DN query, being independent of preference directions (if any), would also report properties that are costlier and smaller than the source q, i.e., theoretically less preferable. Such DNs are also useful for competitor analysis because they may indicate a potential to, say, mark up the price of q, or more aggressively advertise it against these competitors, or take into account (qualitative) factors other than price/area that may be involved in a client's decision. While clear preference directions may or may not exist in the data dimensions[1], this

[1]For instance, a dimension could be the storey number where preference of lower, higher or middle floors is a personal choice [1].

is irrelevant to DN retrieval, its semantics and its applicability.

Exclusive Retrieval Region: The DNs of a source object q also demarcate its *exclusive retrieval region.* Any window query that completely lies in this region and overlaps q is guaranteed to only overlap q. That is, the exclusive retrieval region defines the maximal search area where q is the only result of a window query, and by itself provides an indication of the competitiveness and marketability of q. Fig. 3.1(b) shows the exclusive retrieval region of source r_{10}. The region is delineated by the DNs of r_{10} (i.e., r_3, r_4, r_6, r_7 and r_9). Its derivation is discussed later in this chapter.

Recommendation of Alternatives: In a system where user queries are expressed by windows, the DNs are natural candidates for alternative recommendations. That is, if a user is currently viewing object q, the search portal could suggest the DNs of q as alternatives for consideration. Alternative recommendations are common in property, flight or hotel room search systems, such as tripadvisor.com and booking.com (where users may browse accommodation options based on price and average user ratings).

The rationale behind DN formulation is that (since user interests are captured by windows) the similarity or comparability between two objects q and r is determined by the number of intervening objects retrieved by any window query overlapping q and r. In this regard, the definition can be generalized to provide a partial ordering of competitors based on the number of intervening objects. That is, the fewer the intervening objects, the more immediate threat posed by a competitor. This motivates the K-DN formulation, which reaches a broader set of alternatives by relaxing the DN condition.

Specifically, an object $r \in S$ is a K-DN of source q if there is a query window that intersects r, q and fewer than K other objects in S. K may or may not be known in advance. The latter case entails *incremental* K-DN processing where K can be incremented iteratively without the need to run the query from scratch, instead resuming it from where it last stopped. K-DN search finds application in scenarios similar to plain DN, the difference being that the scope of, say, competitor analysis is wider so that more alternatives are taken into consideration.

Another variant of DN search with practical relevance is the all-DN query. An all-DN query computes the DN set of every object in dataset S. As we explain later, the DN relationship is symmetric. Therefore, the output of the all-DN query may be visualized as an undirected graph in which the nodes correspond to objects, and the edges to instances of DN relationship. Fig. 3.1(c) illustrates the all-DN result for our 10-rectangle example. A straightforward, yet inefficient way to answer the all-DN query is to perform a plain DN search for each $r \in S$. We develop an algorithm that improves performance by three orders of magnitude compared to this naïve solution.

The DN definition is irrelevant to the shape of data objects. We center on rectangular objects with axis-parallel sides, although our techniques also apply to point data and arbitrarily shaped objects (see Sec. 3.3.4). Our focus on rectangles is because they are more general than points and probably the most common/intuitive representation of services and products in multi-dimensional spaces. The extent of a data object in a dimension could capture a degree of fuzziness or its intrinsic association with a range of values. For example, each flight in kayak.com is associated with the range of prices that are offered by different online ticketing agencies (and potentially in different ticket classes). Hotel options in tripadvisor.com are associated with a range for price per night, depending on the exact dates of visit, etc.

As we show later in this chapter, the DN problem and its variants are meaningful in low-dimensional spaces. We therefore assume that dataset S is indexed by a spatial access method, such as the R-tree. Our contributions are summarized as follows:

- We introduce and formalize a new query (DN) and its variants (K-DN and all-DN);

- We devise I/O optimal algorithms for DN and K-DN queries over data organized by a spatial index;
- We develop a sophisticated algorithm for all-DN processing that outperforms by orders of magnitude a repetitive application of DN search.

The rest of the chapter is organized as follows. Section 3.2 gives preliminary knowledge of related work. Section 3.3 introduces DN processing, while Section 3.4 and 3.5 present our K-DN and all-DN algorithms, respectively. Section 3.6 then extends DN search to higher dimensions. Section 3.7 empirically evaluates our techniques, and Section 3.8 summarizes this chapter.

3.2 Preliminaries

As DN processing per se has not been studied before, here we review related query types, such as nearest neighbor, nearest surrounder, and skyline queries. We also survey the segment tree, a data structure that we adapt for our framework.

A related problem is *nearest surrounder* search (NS) [47]. Given a spatial dataset S and a source point q, an NS query retrieves a set $\mathcal{R}_{\mathcal{NS}} \subseteq S$ of objects, each being the nearest neighbor of q with the scope of interest constrained at some range of angles around q. Objects in $\mathcal{R}_{\mathcal{NS}}$ collectively cover the whole angle range $[0°, 360°]$ around q; these objects have a clear line of sight from q, unblocked by other objects. The NS query differs from the conventional NN query in taking into account directional information of the nearest neighbors. The method proposed in [47] uses an angular sweeping technique to process the R-tree that indexes S. This approach also extends to K-tier NS retrieval, where the line of sight between q and each NS object may cross up to $K-1$ others. NS (and K-tier NS) methods exist only for two dimensions and for point (zero-extent) source objects. NS search and similar visibility queries (e.g., [68]) are different by definition from our problem, as we also elaborate in

Sec. 3.3. In that section, however, we devise a baseline DN approach (which works only for two-dimensional data) that uses NS as a building block.

Work on *skyline* processing ([16, 72]) is also relevant to ours, as will become clear in the technical description of our contribution. Consider a dataset S where each object has two attributes, x and y. An object here could correspond to a transportation option between two specific cities, with attributes price (x) and total duration (y). Assume that all options have different x and y values. An object (travel option) r is said to *dominate* another object r' if both of r's attributes are no larger than those of r'. Essentially, this implies that option r is preferable to r' because the former is both cheaper and faster. The *skyline* of S comprises all objects that are not dominated by any other object. *Branch-and-bound skyline* (BBS) is an I/O optimal skyline algorithm [58] that utilizes an R-tree on S. BBS accesses the tree nodes in ascending *mindist* order from the most "preferable" corner of the data space. In our example, this corner is the origin of the data space. Once a data object is found, it is added to the skyline. Subsequently encountered R-tree nodes (or objects) are accessed (included in the skyline, respectively) only if they are not dominated by any object currently in the skyline. K-skyband is a generalization of the skyline that includes all objects dominated by fewer than K others. BBS extends to K-skyband computation, retaining its I/O optimality.

The skyline query can be used for recommendation of alternatives. However, its semantics (and therefore its domain of applicability) is different from DN. The skyline operator is not input-sensitive, meaning that the result is always the same and it does not depend on any user input. In our transportation options example, the skyline options (i.e., those not dominated by any other in the input data) are only dependent on the dataset itself. No input is input-sensitive–the DN result depends on the source object q and varies with its extent and location. Another key difference is that the DN query pertains (and offers an auxiliary decision support mechanism) to systems where the users browse options via window queries. In contrast, the skyline operator requires simply a fixed and monotonic preference order in each data dimension (e.g., the smaller the price/duration of a travel option the better). Despite the differences in semantics, nature, and application domain, our processing techniques utilize on (adapted) skyline algorithm as a building block to derive a subset of DNs.

The *dynamic skyline* receives as input, in addition to data, a set of query objects[2]. Each data object is represented by the vector of its distances from every query object. A data object belongs to the dynamic skyline if its distance vector is not dominated by that of any other data object. The distances between data and query objects can be Euclidean [69], road network distances [30] or general metric distances [22]. The problem differs from ours in that (i) DN search involves a single input dataset (data objects only), (ii) dynamic skylines are defined over (distance) vectors whereas the DN relationship is defined over rectangles and, most importantly, (iii) DN search captures the exclusive co-existence of two objects in the result of a window query instead of the dominance (or not) between them.

In a sense, DN search is related to influence set computation, i.e., identification of objects that could affect or be affected by a source object q. The concept of influence sets was introduced in [46], and formulated as a *reverse nearest neighbor* search (RNN) at q. The RNN set of q includes those objects r in a dataset S that have q as their nearest neighbor. RNN processing has received significant attention; [57] provides a comprehensive survey of existing work.

By definition, the DN query retrieves different objects than RNN. There is an interesting similarity though – window queries in DN play the role of NN queries in RNN. Specifically, given a source object q, DN search (or RNN search) discovers those objects around which, if a window query (a NN query,

[2]The concept of dynamic skyline was introduced in [58] to refer to a more general problem. We focus on its spatial versions due to their higher relevance to DN formulation.

respectively) is issued, the result will exclusively include q. This correspondence implies that the type of influence stemming from the DN relationship is meaningful in domains where user interests are expressed by window queries, whereas the influence derived from the RNN relationship is meaningful in applications where users browse data by NN queries.

There is a similar analogy with the *reverse top-K* query [77] too. However, the problem cannot be mapped to ours. For example, the top-K queries/functions are known in advance, and the influence of q relates to functions rather than other objects in S. However, it is interesting that a reverse query notion (top-K in this case) is used to discover entities (functions) influenced by the source.

The *reverse skyline* query is defined in [29]. That work considers a spatial version of dominance before reversing it. Specifically, the input includes a source point q and a set of d-dimensional data points. A data point r is said to *dynamically dominate* another r' with respect to q if the projection of r on each of the d axes lies closer to q than the corresponding projection of r'. Now, a data point r belongs to the reverse skyline of q if q is not dynamically dominated by any other point with respect to r. Although again a connection exists with our problem, here we consider window queries instead of dynamic dominance. To further stress the difference, note that reverse skyline is not a symmetric relationship (i.e., the fact that r belongs to the reverse skyline of q does not mean that the converse holds too). In contrast, the DN relationship is symmetric, as explained in Sec. 3.5.

The definition of the all-DN graph resembles to some extent the concept of *Gabriel graph* in computational geometry [27]. Given a set S of points in the Euclidean plane, the Gabriel graph G uses S as its vertex set. There is an edge between nodes (i.e., points) r and r' if and only if the *circle* whose diameter has r and r' as endpoints is empty. A fundamental difference between all-DN graph and Gabriel graph is that the latter is defined strictly for points, not rectangles. Another distinction is that its edges represent empty circles

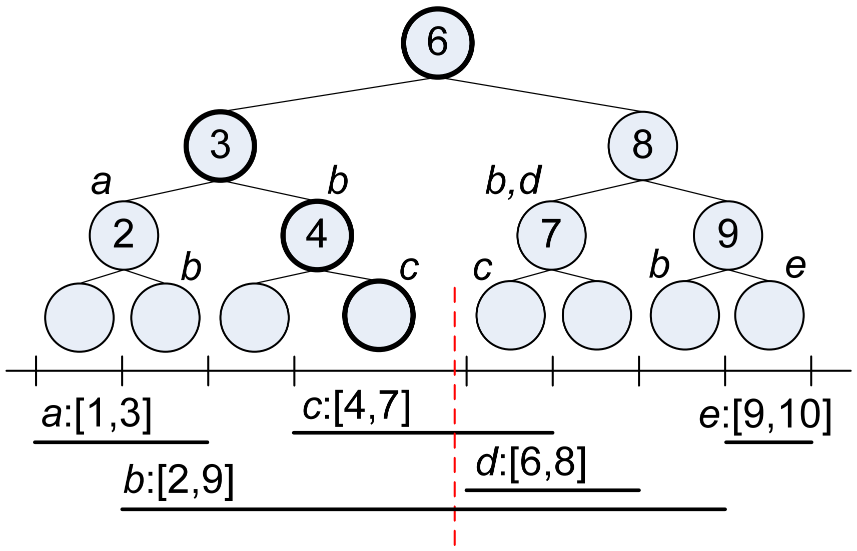


Figure 3.2: Example of segment tree and stabbing query at $q = 5.8$

(with specific, fixed diameter) versus axis-parallel rectangular windows (with arbitrary diagonal). The different edge definition leads to different topologies; for instance, the Gabriel graph is planar (no edge intersects another) which is not the case in all-DN graph (e.g., see Fig. 3.1(c)). The Gabriel graph could be defined under the L^∞ norm (instead of the Euclidean), where an edge between points r and r' exists if and only if the *square* with diagonal corners r and r' is empty. The problem is still defined only for points (and the Gabriel graph remains planar). The edges correspond to empty squares with a specific diagonal versus completely arbitrary axis-parallel windows (with unknown diagonal and arbitrary side-length proportions).

Our algorithms rely on the *segment tree* [12]. This is a balanced binary tree used for efficiently answering stabbing queries on a set of line segments, i.e., reporting all segments in a one-dimensional space that envelop a given query point. Consider a set of line segments $S = \{s_1, s_2, ..., s_N\}$, each delimited by two endpoints. To construct a segment tree on S, we sort the $2N$ different endpoints into an ascending sequence $P = (p_1, p_2, ..., p_{2N})$. The endpoints in P divide the one-dimensional space $(-\infty, \infty)$ into $2N + 1$ *atomic intervals*. A binary tree T is constructed bottom-up, with the leftmost leaf node covering the leftmost interval $(-\infty, p_1]$, the second leaf covering $(p_1, p_2]$, and so on.

Table 3.1: Notation

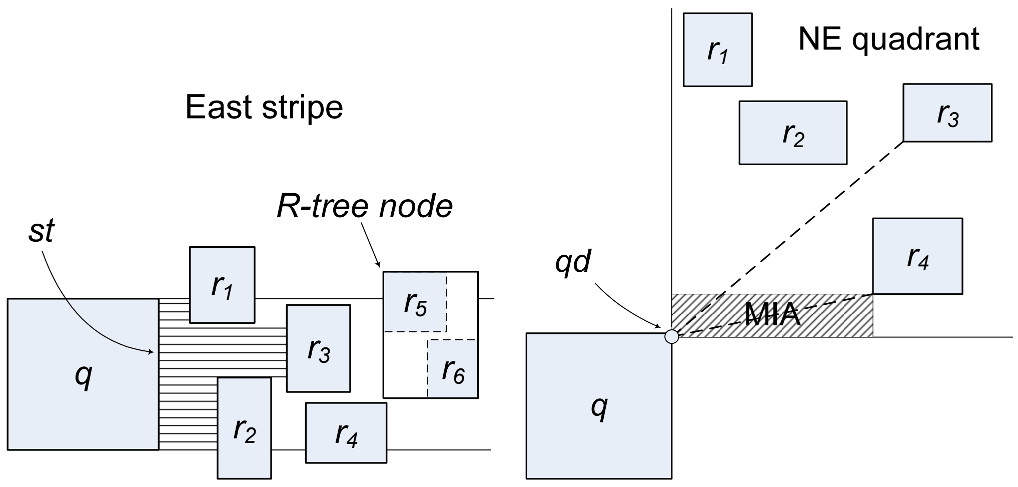
Symbol	**Description**
S	set of data objects
N	cardinality of S
T_x, T_y	segment trees on x- and y-extents of objects
L_i	the object list of leaf node n_i in a segment tree
I_i	interval covered by leaf node n_i in a segment tree
n_-, n_+	boundary leaf nodes in a segment tree

Each internal node in T covers the union of the intervals of its two children. Every (internal or leaf) node n_j in T stores a *segment list* L_j containing those segments of S that completely cover the node's interval but not the interval of its parent. To answer a stabbing query at point q, T is traversed from the root, reporting all the segments in the lists of those nodes n_j whose interval envelops q. The segment tree has $O(N \log N)$ construction time, $O(\log N)$ insertion time, and $O(\kappa + \log N)$ query time where κ is the number of segments in the result.

Fig. 3.2 illustrates a segment tree built on five line segments, a, b, c, d, and e. The line segments and their exact intervals are depicted at the bottom of the figure. Inside each tree node we draw the corresponding splitting value (implicitly defining its covering interval), and next to the node we present its segment list. Suppose the user issues a stabbing query at point $q = 5.8$, shown as a vertical dashed line. The search begins from the root. Comparison with the splitting value therein (i.e., 6) directs the search to the left child, and in turn, from that node to its right child, and so on until it reaches a leaf node. The visited nodes are shown in bold border. The union of the segment lists in visited nodes forms the query result, i.e., segments $\{b, c\}$.

3.3 Direct Neighbor Search

Given a dataset S and a source object q in a d-dimensional space, a data object $r \in S$ is a *direct neighbor* (DN) of q if there exists a (d-dimensional) axis-parallel window that intersects only q and r. Our focus is on low-dimensional



(a) Stripe DN (b) Quadrant DN

Figure 3.3: DN search in quadrants and stripes

spaces – as we explain in Sec. 3.6.1, the DN problem is meaningful when dimensionality is low, because the number of DNs grows quickly with d. For ease of presentation, we consider two dimensions before extending our methodology to more (in Sec. 3.6). We assume rectangular (source and data) objects with axis-parallel sides. Notwithstanding this, our work applies to point data, which may be treated as zero-extent rectangles, and to arbitrarily shaped objects (discussed in Sec. 3.3.4). The objects may or may not overlap. We target disk-resident datasets S, organized by a spatial index like the R-tree.

Suppose that the data space is $[0, X_{max}][0, Y_{max}]$ and the extent of the source q is $[q.x_l, q.x_h][q.y_l, q.y_h]$. We perform DN retrieval in the four *stripes* and four *quadrants* of q. The east stripe is the area defined by the right edge of q, extending horizontally to the right border of the data space, i.e., the area $[q.x_h, X_{max}][q.y_l, q.y_h]$. The east stripe of an example source q is shown in Fig. 3.3(a). The north-east quadrant (NE) is the axis-parallel area extending diagonally from the NE corner of q to the NE corner of the data space, i.e., $[q.x_h, X_{max}][q.y_h, Y_{max}]$. Fig. 3.3(b) shows the NE quadrant of an example source q. The other stripes and quadrants are defined accordingly. Note that q, the four stripes and the four quadrants define a partition of space into 9 regions. We first consider DN search inside the quadrants, proposing two methods for this, then in the stripes. Objects that intersect q are directly reported as DNs (we establish the convention that all of these objects belong to the DN set).

For simplicity, we initially assume that objects in S fall completely inside a single stripe or quadrant. This assumption is relaxed in Section 3.3.3. Our objective is to minimize the total processing cost, comprising I/O and CPU time. Table 3.1 lists the frequently used notation.

3.3.1 DN Search in Quadrants

In Fig. 3.3(b), point qd marks the NE corner of q. A preliminary approach to derive the DNs in the NE quadrant is to issue a *constrained* NS search at qd, limiting the visibility search to the 90° angle NE of qd. The surrounders derived are a superset of the DNs (in NE quadrant). To see this, if rectangle r is a DN of q, by definition there is a query window that exclusively intersects r and q. Since r is in the NE quadrant, this query window must include qd. Also, because this window intersects no other object, there is visibility between qd and r. Therefore, the quadrant DNs are also NSs. On the contrary, an NS is not always a DN; e.g., r_3 is visible from qd but not a *DN* (every query window overlapping r_3 and q necessarily intersects r_2 and r_4 too). Before presenting how false positives (NSs that are not DNs) can be eliminated, we define the notion of minimum intersection area.

Definition 2 *The* ***Minimum Intersection Area (MIA)*** *of an object r (lying in a specific quadrant) is the axis-parallel rectangle defined by qd and the closest corner of r to qd.*

The striped area in Fig. 3.3(b) is the MIA of r_4. It is easy to see that any query window that intersects an object r and q *must* envelop the MIA of r. This leads to the following crucial observation.

Observation 1 *An object r (lying in a specific quadrant) is a DN if and only if its MIA intersects no other object.*

Observation 1 helps to disqualify NSs that are not DNs. Let $\mathcal{R}_{\mathcal{NS}}$ be the set of constrained NSs. In a straightforward application of the observation, we may check for every $r \in \mathcal{R}_{\mathcal{NS}}$ whether its MIA intersects any other object in S; if so, r is disqualified. The remaining NSs are the DNs. This approach would require multiple window queries in S (as many as the total number of NSs), incurring considerable I/O overhead. On a closer inspection, the overhead can be eliminated. Specifically, if there exist objects in S that overlap the MIA of a candidate $r \in \mathcal{R}_{\mathcal{NS}}$, at least one of them will be visible from q, and therefore already in $\mathcal{R}_{\mathcal{NS}}$. This implies that we need to check the MIA of r only against objects in $\mathcal{R}_{\mathcal{NS}}$, rather than against the entire S.

We term the above approach *constrained NS* (CNS). Its main drawback is that it accesses a superset of the strictly needed objects (e.g., r_3 in Fig. 3.3(b)). In our experiments with real and synthetic data (Table 3.4) the number of NSs is an order of magnitude larger than that of actual quadrant DNs, leading to a large false positive ratio. Accessing these false positives translates to unnecessary I/Os. Also, NS queries require a significant amount of computations, due to the angular sweeping mentioned in Sec. 3.2. Observation 1 paves the way for a more efficient (and I/O optimal) method, named *skyline DN* (SDN), via Lemma 1.

Lemma 1 *The DNs of q in a quadrant are exactly the skyline objects in this quadrant, with qd as origin and each object represented by its closest corner to qd.*

Proof 1 *By definition, a point p is dominated by those and only those points p' that fall inside the rectangle defined by p and the origin, called dominating rectangle of p. Thus, p is a skyline point if and only if its dominating rectangle is empty. In our context, since the skyline is defined on the objects' corners* *that are closest to qd, the dominating rectangles of these corners correspond to the MIAs of the respective objects. From Observation 1 it follows that the skyline and quadrant DN sets are identical.*

Based on Lemma 1, SDN computes the DNs in each quadrant using the I/O optimal BBS algorithm [58].

3.3.2 DN Search in Stripes

An object r is a DN of q in a stripe, say the east, if a rectangle can be found to exclusively intersect r and q's right edge st, i.e., there exists a horizontal ray from some point on st to a point on r's left edge that intersects no other object. In Fig. 3.3(a), for example, the east stripe DNs of q are r_1, r_2, r_3. To identify such DNs, we sweep from st to the right, examining objects r (in the east stripe) in increasing $r.x_l$ order. For each object r encountered, we check if the previously considered objects in this stripe (collectively) block r from st. If not, r is a DN. The search terminates when the DNs discovered so far block all the remaining objects in the stripe.

To iteratively discover objects in increasing $r.x_l$ order, we utilize the R-tree on S. The process is similar to a best-first incremental NN search at q, as described in Sec. 3.2, where the sorting key of the min-heap is *mindist* from st. The difference is that the search is constrained to the east stripe and that R-tree nodes that are blocked from st by already encountered DNs are pruned (i.e., not visited). In Fig. 3.3(a), for instance, the illustrated R-tree node that holds r_5 and r_6 is pruned (not accessed at all) during the search because it is fully blocked by the previously reported DNs r_1, r_2, r_3. Object r_4, although popped from the search heap, also fails the visibility check and is excluded from the DN set. The I/O optimality of the best-first NN algorithm in [39], in conjunction with the pruning of blocked nodes, guarantees optimality for the stripe DN search too.

Lemma 2 *The stripe DN algorithm is I/O optimal.*

Proof 2 *As long as the left edge of (the MBR of) an R-tree node n is not collectively blocked by the DNs of q, the node must be accessed because it may hold DNs. To prove the lemma, we must show that only such nodes are accessed. In our algorithm, a prerequisite to visit a node n is that the DNs found* thus far *do not completely block it from q. To complete the proof, we show that the remaining DNs, i.e., those found* after *visiting n, cannot block any part of its left edge. This is obvious – since R-tree entries and DNs are encountered (i.e., popped from the search heap of the incremental NN search) in increasing order of distance from q, all DNs found after visiting n are further from q and hence cannot possibly block any part of n.*

To efficiently perform the "visibility" check for objects and R-tree nodes encountered during the incremental NN search, we use an adaptation of the segment tree. When the search begins, we initialize an empty segment tree T_E. The first NN of q in the stripe is a DN (by definition it cannot be blocked by any other object), and its y-extent is inserted into T_E. Before our incremental NN search visits any R-tree node, the y-extent of the node's MBR is probed against T_E to detect whether any part of its left edge is exposed to st. If completely blocked, it is pruned (ignored). The same check is performed for every discovered NN r. If r is not (completely) blocked, it is reported as a DN, and its y-extent is inserted into T_E.

In our adapted segment tree, segments are stored only at the leaves of T_E. Those segments that are included in the list of an internal node in a conventional segment tree are instead replicated in the lists of all its descendant leaves. To check an object (or internal R-tree node) with extent $[y_l, y_h]$, we issue a stabbing query on T_E at point y_l. Let n_- be the leaf of T_E that covers y_l. We traverse the leaves to the right of n_- until we reach the leaf n_+ that covers y_h. If any of the segment lists between n_- and n_+ is empty, the object (or R-tree node) passes the test, i.e., a part of its left edge is exposed to st.

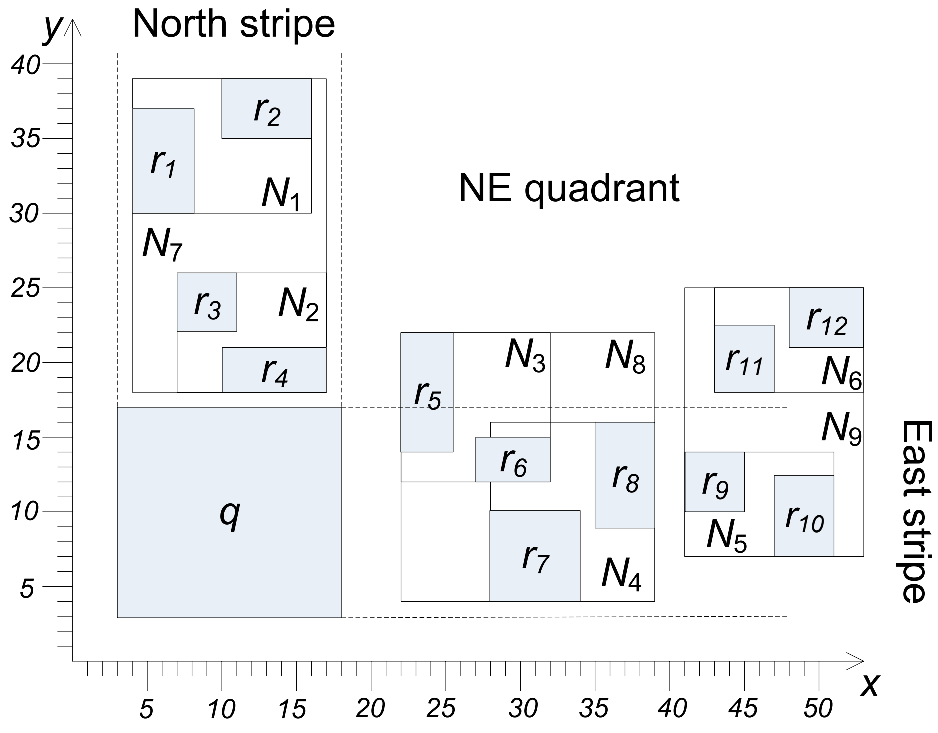


Figure 3.4: Example of DN search

DN search in the other three stripes is similar. Note that the y-extents are used in searching the east and west stripes, while the x-extents are used for the north and south ones.

3.3.3 Complete DN Algorithm

In general, an object may intersect q and/or more than one quadrant or stripe. Essentially, the extent of q, the four stripes and the four quadrants partition the data space into 9 disjoint regions. If an object r intersects more than one region, r is conceptually divided into parts, each falling entirely within one of the regions[3]. The parts are processed independently for the corresponding stripe/quadrant. The final set of reported DNs is the union of the objects that intersect q, and the DNs that are found in the stripes and quadrants. The following discussion refers to our advanced DN algorithm that uses the efficient SDN technique for quadrant search (the case for CNS is similar).

[3]Note that this is an implicit partitioning used only for the processing of the specific DN query. It is not persistent, i.e., it does not affect the representation of r on the disk.

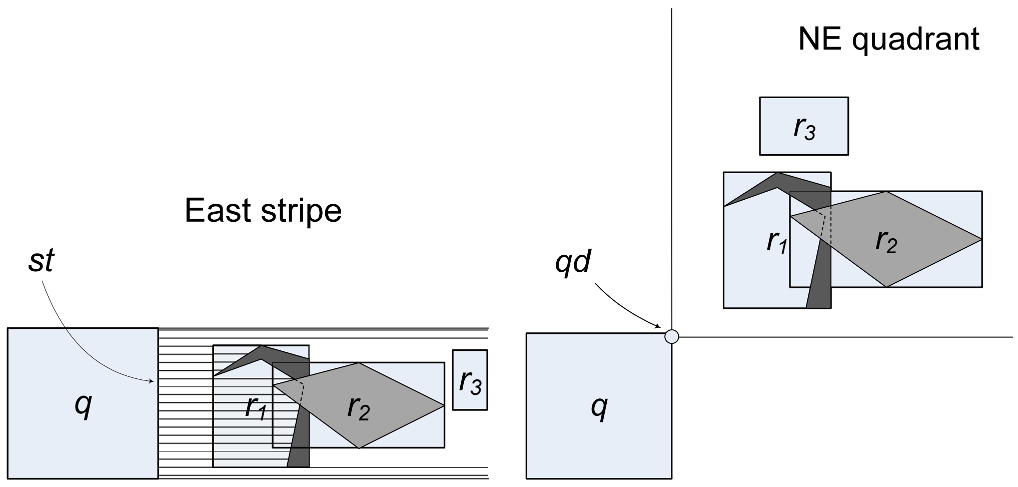
Table 3.2: Heap contents and result formation

Step	**Heap Contents**	**Result**
1	$< N_7, 1 >< N_8, 4 >< N_9, 23 >$	$\emptyset$
2	$< N_2, 1 >< N_8, 4 >$ $< N_1, 13 >< N_9, 23 >$	$\emptyset$
3	$< r_4, 1 >< N_8, 4 >< r_3, 5 >$ $< N_1, 13 >< N_9, 23 >$	$\emptyset$
4	$< N_8, 4 >< r_3, 5 >$ $< N_1, 13 >< N_9, 23 >$	$\{r_4\}$
5	$< N_3, 4 >< r_3, 5 >< N_4, 10 >$ $< N_1, 13 >< N_9, 23 >$	$\{r_4\}$
6	$< r_5, 4 >< r_3, 5 >< r_6, 9 >$ $< N_4, 10 >< N_1, 13 >< N_9, 23 >$	$\{r_4\}$
7	$< N_4, 10 >< N_1, 13 >< N_9, 23 >$	$\{r_4, r_5, r_3, r_6\}$
8	$< r_7, 10 >< N_1, 13 >$ $< r_8, 17 >< N_9, 23 >$	$\{r_4, r_5, r_3, r_6\}$
9	$< r_1, 13 >< r_8, 17 >$ $< r_2, 18 >< N_9, 23 >$	$\{r_4, r_5, r_3, r_6, r_7\}$
10	$< N_9, 23 >$	$\{r_4, r_5, r_3, r_6, r_7, r_1, r_8\}$
11	$\emptyset$	$\{r_4, r_5, r_3, r_6, r_7, r_1, r_8\}$

The search for objects that intersect q, and for DNs in the stripes and quadrants can be performed concurrently, in a single traversal of the R-tree in order to avoid unnecessary I/Os in re-reading R-tree nodes. This is possible, because the search in each quadrant or stripe visits R-tree nodes in increasing distance from q. This means that there can be a single search heap (sorted on $mindist(e, q)$ of encountered R-tree entries e) that serves all the 8 quadrant/stripe searches. During the R-tree traversal, four skyline lists (one per quadrant) and four segment trees (one per stripe) are maintained. Entries (objects) with zero *mindist* intersect q, and are therefore accessed (reported as DNs) directly. A detailed pseudo-code for this complete, single-traversal DN search is provided in the Appendix.

Since the BBS and stripe DN components of our algorithm are I/O optimal (as proven in [58] and Lemma 2), and since we also avoid re-fetching the same nodes from disk, the overall DN method is I/O optimal, i.e., it performs the minimum possible number of I/Os for the given R-tree structure.

Example 1 *We illustrate our complete DN algorithm with Fig. 3.4. For*



(a) Stripe DN (b) Quadrant DN

Figure 3.5: DN search for arbitrarily shaped objects

simplicity, we only show the NE quadrant, and north and east stripes. Table 3.2 shows the search heap contents and the DN set in various stages. First, we read the root of the R-tree on S, and push its three entries N_7, N_8, N_9 into the search heap. We then pop the top entry N_7, fetch it from disk and en-heap its children. This process continues until we pop object r_4 in step 3. It falls in the north stripe and we directly insert it into the result set and into the north segment tree T_N. In step 6, we pop r_5 which intersects the NE quadrant and the east stripe. We conceptually partition r_5 and treat each portion as a separate object in the respective quadrant/stripe. It is a DN in both the NE quadrant and the east stripe, and is thus appended to the result. It is also inserted into the NE skyline and into T_E (the east segment tree). Subsequently popped objects r_3 and r_6 are also DNs, and are inserted into T_N and T_E, respectively. In step 9, object r_2 is popped after r_1 and r_8. It fails the visibility check against T_N (because it is completely blocked by r_3 and r_4) and is discarded. Step 10 pops index entry N_9 which intersects the NE quadrant and the east stripe. We conceptually divide N_9 into two portions. Its portion in the east stripe fails the visibility check against T_E, and its second portion is dominated by the DNs already in the NE skyline (i.e., r_5). Thus, N_9 is ignored (not read from disk). At that stage the heap is empty, and the DN search terminates with result set

$\{r_4, r_5, r_3, r_6, r_7, r_1, r_8\}$.

An issue worth mentioning regards the exclusive retrieval region (ERR) of q, described in Introduction. By definition, this region is bounded by the DNs. In a quadrant, the bound is the skyline over the quadrant's DNs. In a stripe, the ERR includes the area that is not blocked by the stripe's DNs. In other words, the ERR can be derived by iteratively subtracting from the entire space the area that is dominated by each quadrant DN and the area that is blocked by each stripe DN (where dominance and horizontal blocking are defined by the specific quadrant or stripe, as explained in Sec. 3.3 and 3.3.2).

3.3.4 Arbitrary Object Shapes

So far we have focused on rectangular objects. Our techniques, however, extend easily to arbitrary object shapes.

Stripe DNs. The stripes are processed similarly to rectangular objects. The difference is that the object lists of the segment tree (used for a specific stripe) hold both the MBR and the exact geometry of each discovered DN. In checking the visibility of an R-tree node, the exact geometries in the object lists are only taken into account when their MBRs intersect that of the R-tree node in question. Visibility check for a leaf node entry (i.e., MBR of a data object) proceeds similarly, except that if the MBR intersects (the actual geometry of) a DN, the object's exact geometry must also be fetched to complete the check. Consider, for example, Fig. 3.5(a) and DN search in the east stripe. Assume that r_1 is the only DN found so far. The next encountered node is r_2, which overlaps the MBR of r_1. The exact geometry of r_1 is also overlapping the MBR of r_2, and we can only determine whether the latter is a DN by fetching its own exact geometry too. The comparison reveals that a part of r_2 is unblocked and therefore it is a DN. On the other hand, we may infer that r_3 is not a DN without fetching any exact object geometries, because its MBR does not overlap that of r_1, and it is fully blocked by it.

Quadrant DNs. The quadrant search traverses the R-tree of S following the BBS strategy as normal. An internal R-tree node is loaded only if it is not dominated by any existing skyline object. The MBR of a skyline object can be used for quick dominance check. If the R-tree node does not intersect the MBR, dominance (or not) is definitively decided based on their closest corners to q. If there is overlap, the skyline object's exact geometry must be taken into account.

When a leaf node entry is popped (i.e., the MBR of a data object), we check whether the MBR is dominated by any object already in the skyline. If not, we fetch the exact geometry of the corresponding object from the disk and push it into the heap with its *actual* minimum distance from q as the key value. If the object is popped subsequently, we check (using its exact geometry) whether it is dominated by any existing skyline object. If not, it is added to the skyline and to the DN set.

In Fig. 3.5(b), for instance, assume that we have discovered one DN so far, r_1. We can infer that r_3 is not a DN using only MBR information; the MBR of r_3 does not overlap that of r_1 and is also dominated by it. In contrast, when we encounter r_2 we cannot make a safe conclusion based on its MBR (because it overlaps r_1); therefore, we fetch its exact geometry, and en-heap it with its actual distance as key. When the latter is popped again, we compare its exact geometry with that of r_1 and determine that a part of it is not dominated. Thus, r_2 is also a DN. A detail regards the dominance check between exact geometries represented as polygons: object r_2 is not dominated by r_1 because one of its polygon vertices (the left-most in this case) is not dominated by any vertex of r_1.

3.4 K-Direct Neighbor Search

An intuitive way to compute a broader spectrum of "neighbors" is to relax the DN condition to allow up to some number of intervening objects in the query window (in addition to q and each of its DNs). Specifically, an object $r \in S$ is a K-DN of q if and only if there exists a query window that intersects q, r, and no more than $K - 1$ other objects. The K-DN query is a generalization of plain DN (the latter corresponds to $K = 1$).

Stripe K-DNs. Following a similar definition to plain DN, an object r is a K-DN of q in a stripe, say the east, if there exists a horizontal ray from any point on q's right edge st to a point on r's left edge that cuts through fewer than K other objects. In Fig. 3.3(a), for example, the 2-DNs of q in the east stripe include the 1-DNs r_1, r_2, r_3, plus objects such as r_4, r_5, and r_6 which can be "reached" from q via a horizontal ray that cuts across exactly one other object. In the case of r_4 the intervening object is r_2.

Identifying K-DNs in a stripe is similar to plain DN. Take the east stripe as example. A stripe-constrained, incremental NN query is posed at st (i.e., the right side of q), examining objects r in increasing $r.x_l$ order. Whenever a K-DN is found, it is inserted into the segment tree T_E. When the constrained NN search considers whether to visit (i.e., read from disk) an internal node of the R-tree, we probe T_E and examine all its leaf nodes that overlap the y-extent of the R-tree node. If at least one of them has fewer than K objects in its list, the R-tree node is visited. Data objects r discovered in the NN search are reported as K-DNs if they pass the same test, or disqualified otherwise. The stripe DN operation is I/O optimal, with the constrained NN search following the best-first paradigm [39] in conjunction with pruning R-tree nodes that fail the segment tree test. The proof is similar to Lemma 2.

Quadrant K-DNs. The definition of MIA and Observation 1 extend to $K > 1$. Let r be an object in a quadrant. As established in Sec. 3.3, any query window that intersects both q and r must completely envelop the MIA

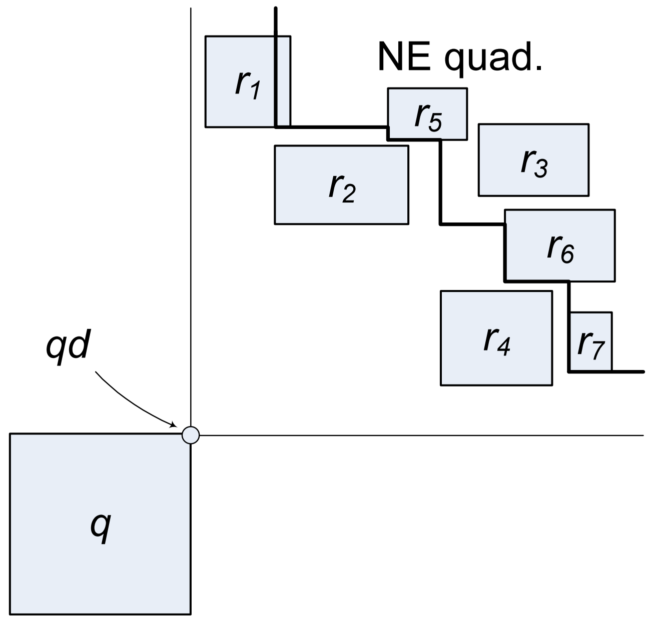


Figure 3.6: 2-DNs in NE quadrant (2-skyband)

of r. Thus, the necessary and sufficient condition for r to be a K-DN is that its MIA intersects fewer than K other objects.

CNS can be used to retrieve such objects using a K-tier (90°-constrained) NS query. However, this inherits the deficiencies identified in Sec. 3.3, namely that a superset of the actual K-DNs is returned, which entails unnecessary I/Os and a post-processing (filtering) step. As we show in the experiments, the problem is exacerbated as K increases, because a larger fraction of the NSs are not DNs.

On the other hand, Lemma 3 extends SDN to $K > 1$, enabling I/O optimal processing. Let qd be the corner of q that anchors the quadrant to be processed. Recall that the K-skyband is a generalization of the skyline that includes objects that are dominated by fewer than K others.

Lemma 3 *The K-DNs of q in a quadrant are exactly the K-skyband objects in the quadrant, with qd as origin and each object represented by its closest corner to qd.*

Proof 3 *Let $r \in S$ be an object in the quadrant of qd. The necessary and sufficient condition for r to be a K-DN is that its MIA intersects fewer than*

K other objects. Any object r' in the quadrant of qd that overlaps r's MIA has, by definition, its closest corner to qd inside the MIA, and vice versa. Hence, r is a K-DN if and only if its MIA contains fewer than K closest-to-qd corners of other objects, i.e., it is dominated by fewer than K objects.

Fig. 3.6 illustrates a 2-DN search in the NE quadrant of q. The bold staircase line corresponds to the 2-skyband boundary, implying that anything that does not touch this boundary and lies further NE from it (i.e., further to the right and top), is dominated by at least 2 objects. The 2-skyband comprises objects whose closest corner to q either falls on the staircase line (like r_6) or lies SW from it (like r_1). The 2-DN set (equivalently, the 2-skyband) includes r_1, r_2, r_4, r_5, r_6, r_7. Object r_3 is not a 2-DN because it is dominated by r_2 and r_4. The K-DNs in each quadrant can be computed with the I/O optimal K-skyband BBS algorithm of [58].

Similar to the complete plain DN case in Sec. 3.3.3, all the K-DNs (in the stripes or quadrants, or those intersecting q) can be retrieved in a single traversal of the R-tree that indexes S, yielding an overall I/O optimal solution. Furthermore, adopting the branch-and-bound paradigm in the SDN-based K-DN method for the stripes and the quadrants lends two desirable properties, namely, that the method is *progressive* and *incremental.* Progressive implies that DNs can be reported as they are discovered, without waiting for the algorithm to terminate. The incremental nature of our method implies that there is no need to fix K in advance. Should the user originally specify a K value that turns out to be too small, the retrieval of DNs for a larger K does not need to start from scratch, but can resume from where the previous (smaller K) search stopped. Of course, to support incremental processing, pruned R-tree nodes and rejected objects cannot be discarded, but must be kept (sorted on increasing distance from q) in anticipation of an increase in K.

Example 2 *Consider the example in Figure 3.4, and assume q retrieves K-DNs with $K = 2$. The K-DN search differs from DN search in the way it* *prunes non-result objects. We focus on pruning and omit other details for simplicity. The search is exactly the same as before until step 10 in Table 3.2, where r_2 is kept as a 2-DN because, while using the north segment tree T_N to verify the visibility of r_2, we find that only r_4 is blocking it from q. Next we have to check N_9, since parts of it are blocked by only one object according to the east segment tree T_E. Children r_9 and r_{10} of N_5 pass the check, since none of them is separated by two or more objects from q. In the NE quadrant, r_{11} is a 2-DN whereas r_{12} is not, because the MIA of r_{12} intersects r_5 and r_{11}. Now the heap becomes empty, and the search terminates with result set* $\{r_4, r_5, r_3, r_6, r_7, r_1, r_8, r_2, r_9, r_{11}, r_{10}\}$.

3.5 All-Direct Neighbor Query

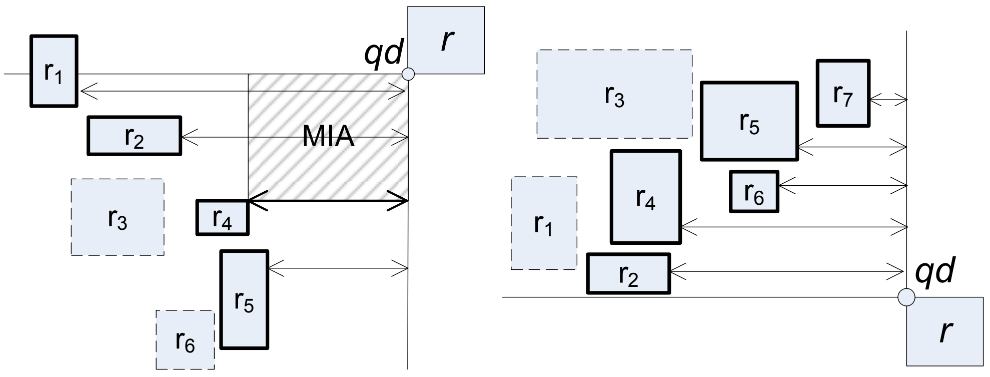
Another extension to DN search is the *all-DN* (ADN) query. Given a set of objects S, an ADN query computes for each object in S all its DNs. A straightforward way to process ADN queries is to apply a plain DN search using, in turn, each object $r \in S$ as the source. Clearly, this approach is inefficient. Performance can be improved if the entire R-tree is loaded in memory to avoid multiple reads from the disk (since all of its contents need to be accessed anyway). However, the CPU cost remains a major drawback. In the following we describe an algorithm for ADN processing that significantly reduces the processing time of individual DN retrievals, by sharing computations among them.

3.5.1 Fundamental Properties of DNs

We begin with observations/properties of DN relationship.

Observation 2 *The DN relationship is symmetric.*

By definition, if object r' is a DN of another object r, there exists a query window intersecting only r' and r, regardless of whether r' lies in a stripe or



(a) DNs in SW quadrant (b) DNs in NW quadrant

Figure 3.7: Properties of quadrant DNs

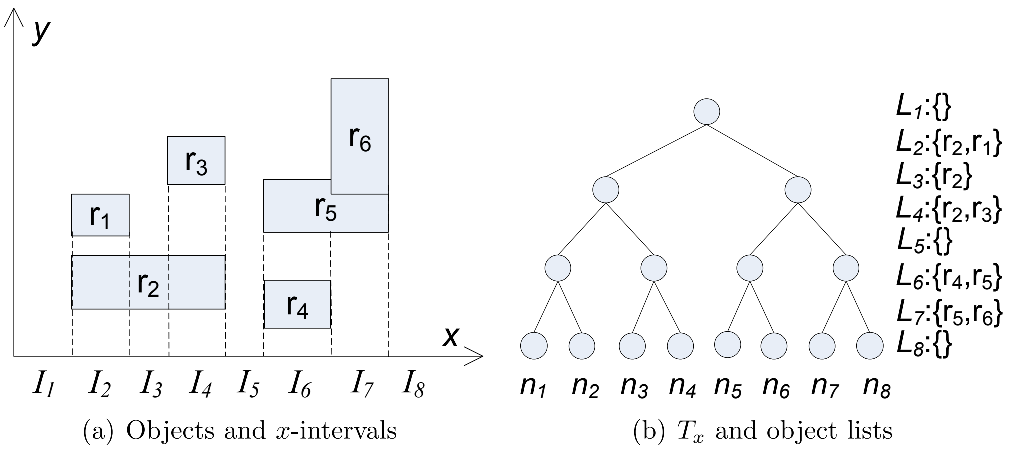
quadrant of r. Thus, r is a DN of r' too.

According to Observation 2, the ADN query requires finding the DNs in only two quadrants and two stripes of each object instead of four. For example, we may consider only the NW and SW quadrants, and only the west and south stripes. The rationale is that if another object r' is a DN with respect to, say, the NE quadrant of r, then r' will definitely identify r as a DN in its SW quadrant.

Observation 3 formulates an important property of quadrant DNs. We take the SW and NW quadrants as an example, and illustrate in Fig. 3.7.

Observation 3 *Given a rectangle r and its set of DNs in the SW (or NW) quadrant, there exists a portion on the right side of each of these DNs that is horizontally unblocked from the vertical edge of the quadrant.*

Fig. 3.7(a) illustrates that the DNs in the SW quadrant of r are all visible via horizontal rays shot from the vertical quadrant edge (i.e., the vertical half-line with qd as initial point). The DNs are shown with solid border, and the horizontal rays are shown as arrows. Observation 3 follows from Observation 1. DN r_4, for instance, is definitely "visible" by some horizontal ray, because its MIA (and therefore the bottom edge of the MIA) intersects no other object. The converse, however, is not true; in other words, not all objects visible via horizontal rays are DNs. The two objects with dashed border (r_3, r_6) are both



(a) Objects and x-intervals (b) T_x and object lists

Figure 3.8: Segment tree on the x-extents of objects

visible but are not DNs. The situation in the NW quadrant is similar (see Fig. 3.7(b)).

Our ADN algorithm, presented next, builds on the above observations to achieve efficient processing.

3.5.2 The All-DN Algorithm

Our approach aims to maximize *computation sharing* among DN retrievals of the objects in S. It achieves this by using two segment trees, T_x and T_y, and performing a sweep of the data space from left to right. Based on Observation 2, we compute for each $r \in S$ its DNs in the SW and NW quadrants, and the west and south stripes.

Fetching all the objects[4] from disk, we first construct a segment tree T_x on their x-extents. Each leaf node n_k^x in T_x is associated with a list L_k^x of objects whose x-extents overlap the interval corresponding to n_k^x. Objects stored in L_k^x are maintained in ascending order on their lower y-values. Fig. 3.8 illustrates a segment tree on the x-extents of 6 objects. Fig. 3.8(a) shows on the x-axis the intervals I_k^x associated with each leaf, and Fig. 3.8(b) presents the tree structure and the object lists L_k^x of the leaves. Note that the superscript x indicates the dimension indexed by the segment tree, but is omitted from the

[4]Note that any ADN algorithm has to read S from the disk. Also, the size of T_x is manageable, as we show in experiments. In seriously memory-constrained systems, a disk-based segment tree can be used [4].

figure for clarity.

In constructing T_x, we do not insert objects one by one. Instead, T_x is bulk-loaded. After sorting the objects in S on their lower x-values, we use the resulting $2|S| + 1$ x-intervals (and their object lists) as leaf nodes. Then, we build the segment tree bottom-up.

After constructing T_x, we scan from its leftmost leaf node to the rightmost. For each leaf n_k^x, we consider the objects in its list in ascending lower y-value and compute their DNs (in the SW and NW quadrants, and in the west and south stripes). After processing all the objects in the list, we proceed to the next leaf node, until all the leaves are visited. The details are as follows.

South stripe DNs. Suppose $\{n_1^x, n_2^x, ..., n_{2|S|+1}^x\}$ is the sequence of leaf nodes in T_x. Starting from the leftmost leaf's list and moving to the right, we identify as DNs (in the south stripe) every pair of successive objects $\langle r_i, r_{i+1}\rangle$ in the list. Correctness is obvious, since by definition there is no object between r_i and r_{i+1} in the x-interval of the corresponding leaf. Consider, for example, Fig. 3.9. The list for node n_k^x contains (among others) objects r_{10} and r_{11}. Due to the sorting of the list on lower y-value, r_{10} and r_{11} are placed consecutive to each other and, hence, they are correctly reported as DNs.

The remaining DNs (west, SW, NW) are also discovered during the aforementioned left-to-right scanning of T_x. This task is facilitated by a second segment tree T_y, similar to T_x, which however (i) is built on the y-extents of objects, and (ii) is incrementally populated. T_y is empty initially. On encountering an object r in some T_x node for the first time, we insert it into T_y (according to its y-extent). The objects in the leaves of T_y are sorted in ascending order on their upper x-value. In the following we explain how T_y enables the retrieval of the remaining three types of DNs and helps in sharing computations in the ADN retrieval.

West stripe DNs. Whenever an object r is inserted into a leaf node n_k^y of T_y, we immediately identify the last object in n_k^y as a DN with respect to

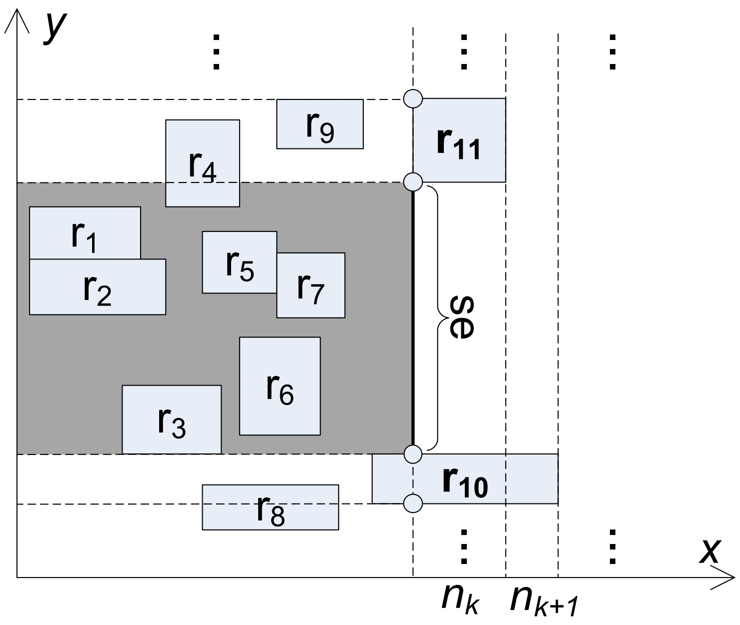


Figure 3.9: Finding west, SW and NW DNs of r_{11}

the west stripe of r. This is because that object has the largest upper x-value among all the objects to the left of r, in the y-interval that corresponds to n_k^y. Continuing the example in Fig. 3.9, Fig. 3.10(a) and 3.10(b) illustrate the leaf node lists in T_y before and after the insertion of r_{11}. For clarity, the superscript y is omitted from the leaf names, and leaves irrelevant to the example are not shown. The insertion creates a new leaf in T_y, and r_{11} is appended in the object lists of $n_{13}^y, n_{14}^y, n_{15}^y$. The rightmost objects in these lists are r_{11}'s west DNs (r_4 in L_{13}^y, along with r_9 in L_{14}^y and L_{15}^y).

NW and SW DNs. T_y is used for efficient NW and SW DN search too. Without loss of generality, assume that no pair of objects have exactly the same lower or upper x-value. Consider again the scanning of leaf nodes in T_x in the process of retrieving the south and west DNs. For each leaf n_k^x, the left bound of its associated interval is due to either the lower or upper x-value of an object. In Fig. 3.9, for instance, the left bound of n_k^x is due to the *lower* x-value of r_{11}, whereas the left bound of n_{k+1}^x is due to the *upper* x-value of r_{11}. For leaf nodes of the former category, we compute the NW and SW DNs of the responsible object; i.e., when considering n_k^x we compute the NW/SW

Leaf of T_y	***Object list***
n_1	{r_8}
n_2	{r_8, r_{10}}
n_3	{r_{10}}
n_4	{r_3}
n_5	{r_3,r_6}
n_6	{r_6}
n_7	{}
n_8	{r_2,r_7}
n_9	{r_2,r_5,r_7}
n_{10}	{r_1,r_5}
n_{11}	{r_1}
n_{12}	{r_4}
n_{13}	{r_4,r_9}
n_{14}	{r_9}

(a) Before insertion

Leaf of T_y	***Object list***
n_1	{r_8}
n_2	{r_8,r_{10}}
n_3	{r_{10}}
n_4	{r_3}
n_5	{r_3,r_6}
n_6	{r_6}
n_7	{}
n_8	{r_2,r_7}
n_9	{r_2,r_5,r_7}
n_{10}	{r_1,r_5}
n_{11}	{r_1}
n_{12}	{r_4}
n_{13}	**{r_4,r_{11}}**
n_{14}	**{r_4,r_9,r_{11}}**
n_{15}	**{r_9,r_{11}}**

(b) After insertion

Figure 3.10: Inserting r_{11} into T_y

DNs of r_{11}, whereas no NW/SW processing is needed for n^x_{k+1}.

We now focus on n^x_k and the SW processing for r_{11}. By definition, all the SW DNs of r_{11} are inside the gray area in Fig. 3.9, i.e., inside the region $[0, r_{11}.x_l][r_{10}.y_h, r_{11}.y_l]$. Note that this "stripe" is delimited by r_{10}, the preceding object in the list of n^x_k. Since the left bound of n^x_k is attributed to $r_{11}.x_l$ and r_{10} is in L^x_k too, it holds that $r_{10}.x_l < r_{11}.x_l$, i.e., there is a portion of r_{10} protruding to the left of n^x_k. This portion of r_{10} disqualifies any object lower than the gray area from being a SW DN of r_{11}.

To identify the DNs in the gray area, we use Observation 3 as a filtering step to produce an (inclusive) list $\mathcal{R}$ of candidate SW DNs. Specifically, we utilize T_y to retrieve those objects in the gray area that are horizontally unblocked from the left of line segment se (shown in the figure). The process is similar to retrieving an object's west DNs using T_y. Fig. 3.10(b) shows the state of T_y when processing r_{11}. The leaves that overlap se (on the y-extent) are n^y_4 up to n^y_{12}. $\mathcal{R}$ is formed by collecting the last (i.e., rightmost) objects in the lists of these T_y leaves, i.e., $\mathcal{R} = \{r_4, r_1, r_5, r_7, r_6, r_3\}$. The leaves of T_y are scanned from n^y_{12} towards n^y_4 (i.e., from higher y-values to lower ones), leading to an inherent sorting of $\mathcal{R}$ according to upper y-values. This sorting facilitates subsequent refinement.

The refinement step eliminates non-DN entries from $\mathcal{R}$ based on Lemma 1, i.e., by finding the skyline objects in it. Due to the inherent ordering in $\mathcal{R}$, the time needed for skyline processing is linear to $|\mathcal{R}|$. We scan $\mathcal{R}$ from the first to the last object, i.e., in decreasing upper y-value. The first object, r_4, must be a SW DN as its highest y-value means that it cannot be dominated by any other candidate. For each subsequent candidate r in $\mathcal{R}$ to be a skyline object (i.e., a DN), it must have an upper x-value larger than that of the *last identified DN*. Note that this single comparison saves considerable computations compared to checking r for dominance against all the skyline objects found. In our example, the second candidate, r_1, is not a DN because its upper x-value is smaller than that of r_4. The third candidate, r_5, passes this check against r_4 and becomes the last reported DN. Thus, the fourth candidate, r_7, is checked against r_5 *only* (as opposed to the entire skyline). Object r_7 is reported as a DN, and subsequently disqualifies r_6 and r_3, leading to r_4, r_5 and r_7 being the final SW DNs of r_{11}.

A symmetric filter-and-refine procedure is used to retrieve the NW DNs of r_{11}. The difference is that the corresponding (gray) search area is refined by r_{12}, the next object in the list of n^x_k. If r_{11} is the last object in L^x_k, the search area extends upwards to the boundary of the data space. Also, the traversal of T_y nodes that fall in the search area is performed in increasing y-order, leading to an $\mathcal{R}$ that is sorted on the lower (instead of the upper) y-value of the candidate objects.

Note that the filtering and refinement steps (for either SW or NW DNs) have a cost linear to the size of $\mathcal{R}$, since no sorting is required in the skyline computation. The latter is taken care of by the structure of T_y. There is

Algorithm 3.1: ADN

```
Input: a set $S$ of objects
Output: set $\mathcal{R_{DN}}$ of DN pairs in $S$
1  build segment tree $T_x$ on $x$-extents of objects in $S$;
2  $\mathcal{N}_x \leftarrow$ sequence of leaf nodes of $T_x$;
3  initialize an empty segment tree $T_y$;
4  $\mathcal{R_{DN}} = \emptyset$;
5  for $i = 1$ to $|\mathcal{N}_x|$ do
6      let $L_i^x$ be the object list of node $n_i^x$;
7      for each object $r_j$ in $L_i^x$ do
8          append the pair $\langle r_{j-1}, r_j \rangle$ to $\mathcal{R_{DN}}$;
9          if left bound of $n_i^x$ is due to $r_j.x_l$ then
10             $\mathcal{R_{DN}} = \mathcal{R_{DN}} \cup$ SearchGrayArea($T_x, T_y, r_j$);
11             insert $r_j$ into $T_y$ according to its $y$-extent;
12             for each leaf $n_k^y$ covered by $[r_j.y_l, r_j.y_h]$ do
13                 append the pair $\langle r_j, r' \rangle$ to $\mathcal{R_{DN}}$, where $r'$ is the last
                   object in the list of $n_k^y$;
14 return $\mathcal{R_{DN}}$;
```

thus only a one-time cost in inserting each encountered object into T_y which, once spent, is utilized by all the subsequently considered objects. This avoids many unnecessary computations (compared to independent DN retrievals for every $r \in S$). Finally, the algorithm extends trivially to all-K-DN processing, following a methodology similar to Sec. 3.4.

The detailed ADN algorithm is given in Algorithm 3.1. Line 8 finds DN pairs from the north and south stripes, line 10 searches for SW and NW DNs in the gray area (using Algorithm 3.2), whereas line 13 identifies the west and east stripe DNs.

3.6 DN Search in Higher Dimensions

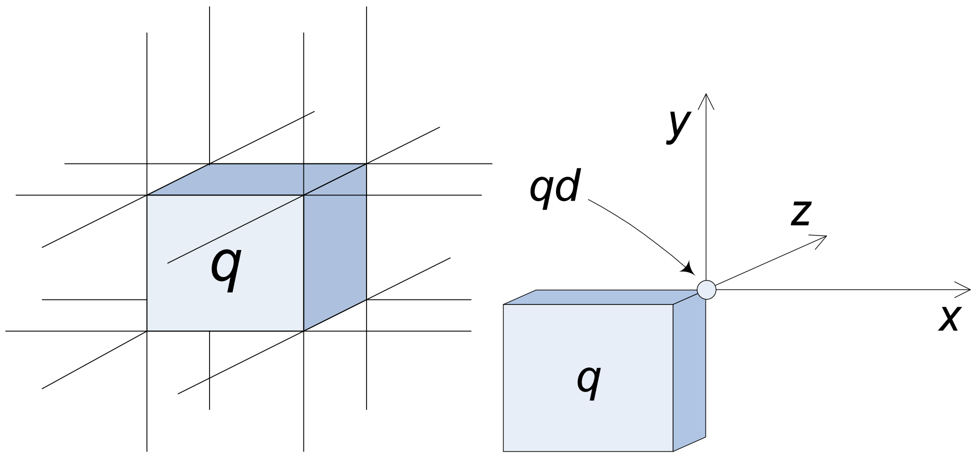
Our methodology extends beyond two dimensions while retaining its I/O optimality. In this section we discuss the three-dimensional case (i.e., $d = 3$) and then generalize to more dimensions. Prior to that, however, we establish that DN search is meaningful primarily in low-dimensional spaces, and justify our focus on this setting.

Algorithm 3.2: SearchGrayArea

```
   Input: $T_x, T_y$, an object $r_j$
   Output: NW and SW DNs of $r_j$
1  $\mathcal{R}_{SW} = \emptyset$;
2  issue a stabbing query on $T_y$ at $r_j.y_l$;
3  let $n_+$ be the resulting leaf node in $T_y$;
4  scan from $n_+$ towards the leaves with smaller $y$-values until reaching the
   boundary of a leaf $n_-$ whose last object's upper $x$-value satisfies
   $x_h > r_j.x_l$;
5  let $\mathcal{R}$ be the sequence of the last objects in the lists for the leaves
   between $[n_+, n_-]$;
6  while $\mathcal{R}$ is not empty do
7      remove the first object $r'$ from $\mathcal{R}$;
8      let $r''$ be the last added DN to $\mathcal{R}_{SW}$;
9      if $r'.x_h > r''.x_h$ then append $\langle r_j, r' \rangle$ to $\mathcal{R}_{SW}$;
10 $\mathcal{R}_{NW} = \emptyset$;
11 issue a stabbing query on $T_y$ with $r_j.y_h$;
12 let $n_-$ be the resulting leaf node in $T_y$;
13 scan from $n_-$ towards the leaves with greater $y$-values until reaching the
   boundary of a leaf $n_+$ whose last object's upper $x$-value satisfies
   $x_h > r_j.x_l$;
14 let $\mathcal{R}$ be the sequence of the last objects in the lists for the leaves
   between $[n_-, n_+]$;
15 while $\mathcal{R}$ is not empty do
16     remove the first object $r'$ from $\mathcal{R}$;
17     let $r''$ be the last added DN in $\mathcal{R}_{NW}$;
18     if $r'.x_h > r''.x_h$ then append $\langle r_j, r' \rangle$ to $\mathcal{R}_{NW}$;
19 return $\mathcal{R}_{SW} \cup \mathcal{R}_{NW}$;
```

3.6.1 Effect of Dimensionality

A hyper-rectangle q in d dimensions has 2^d vertices – in two-dimensional terms, vertices correspond to corners. Equivalently, each of the vertices defines a space partition. In a way similar to quadrants, the DNs in each of these 2^d partitions coincide with the skyline objects (the details of why this is the case are discussed later in this section). On the other hand, it is known that the number of skyline points increases exponentially with dimensionality [83]. Since the DN set is a superset of 2^d skylines (each of exponential cardinality with d), the number of DNs increases at least exponentially with d. In other words, in high dimensionality the DN set includes a large fraction of the dataset S. This limits the usefulness of the query, since the very motivation of the DN



(a) Partitions (b) Vertex partition at qd

Figure 3.11: DN search in three dimensions

problem is to identify a small subset of S as immediate competitors of q. Its diminishing selective power with d suggests that it is more meaningful in low dimensional spaces (a feature common in many queries, like skylines [83], NNs [14], etc). Furthermore, in practice, window queries in applications like propertyguru.com.sg and kayak.com (mentioned in Introduction) typically do not involve more than three or four dimensions. Notwithstanding this, our DN processing methodology extends to $d > 2$ and remains I/O optimal (regardless of dimensionality).

3.6.2 Processing in Three Dimensions

In three dimensions, the source and data objects are three-dimensional boxes, and S is indexed by a 3-D R-tree. The DN relationship is expressed in terms of 3-D window queries, i.e., two objects are DNs if and only if they can be exclusively intersected by some axis-parallel box. The geometry of the source object q includes 8 vertices, 6 faces, and 12 edges. Each of these elements defines a partition, which leads to 27 partitions in total (including box q itself) as shown in Fig. 3.11(a).

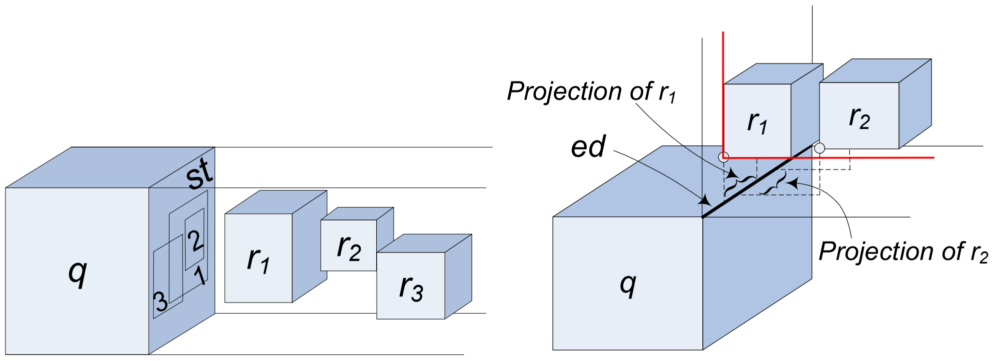
The treatment of vertices is similar to that of quadrants in two dimensions. Consider the partition defined by vertex qd in Fig. 3.11(b). The MIA of an object r that falls in this partition is the three-dimensional box with a diagonal from qd to the vertex of r that is closest to qd. Lemma 1 extends trivially to three dimensions; the DNs in a vertex partition are the skyline objects in this partition with the vertex (i.e., qd) as origin. BBS can be used to compute the 3-D skyline in an I/O optimal way [58]. Note that CNS is no longer applicable, because there is currently no NS algorithm for more than two dimensions.

Faces are handled like stripes in 2-D, the difference being that a spatial access method (e.g., a main-memory 2-D R-tree) is used instead of a segment tree for visibility check. An incremental NN search is initiated at the face, directed outside of q. Denote the face as st and its 2-D R-tree as T_{st}. The projections of encountered DNs on st (projections are necessarily rectangular) are inserted into T_{st}. The incremental NN search in the index of S prunes nodes and ignores objects whose projections on st are entirely covered by DN projections already in T_{st}.[5]

In Fig. 3.12(a), the rectangles on face st are the projections of three objects (numbered according to the subscripts of the objects). r_1 is the first NN of st, and is thus a DN. Object r_2 is not a DN because its projection is fully covered by that of r_1. If r_2 were the MBR of an index node, it would be pruned. On the other hand, r_3 is a DN because it is not fully blocked by r_1. At this stage, T_{st} includes the projections of r_1 and r_3 (i.e., of the DNs found so far).

Edges require special treatment. Each edge ed has two *fixed* dimensions and one *variable*. In Fig. 3.12(b), the x and y dimensions are fixed (every point on ed has the same x and y coordinates) while z is variable. The MIA of an object r in the partition of ed is the box defined by ed and the closest edge of r that is parallel to ed. Objects that could potentially disqualify r from being a DN must first of all dominate it in the fixed dimensions (x and y). Such a dominating object $r' \in S$ disqualifies the part of r that overlaps with its z

[5]To perform this check, when considering an object r (or an index node), we process a window query in T_{st} to retrieve all the DN projections (rectangles) that overlap with that of r. After subtracting these rectangles from the projection of r using a polygon clipping algorithm [76], we check whether there is a residual. If so, the object (or node) passes the test.



(a) Face partition at *st* (b) Edge partition at *ed*

Figure 3.12: DN search in face and edge partition

extent (if there is any overlapping). In Fig. 3.12(b), for instance, r_1 dominates r_2 in the subspace of fixed dimensions (i.e., the $x : y$ plane). In the variable dimension (z), the z-extent of r_1 only partially covers that of r_2 (see their projections on *ed*), and disqualifies only that part of r_2. The remaining part of r_2 may still be exclusively intersected (along with q) by a three-dimensional window query, i.e., r_2 is a DN.

Formally, an object r in the partition of edge *ed* is a DN if and only if the objects that dominate it in the fixed dimensions do not collectively cover its extent in the variable dimension. The proof resembles that of Lemma 1 and is omitted. We adapt BBS to perform DN search in the partition of *ed*. The algorithm proceeds like a skyline computation, i.e., with an incremental NN search at *ed*. A node is visited (or an object included in the DN set) if the DNs found so far that dominate it in the fixed dimensions, do not collectively cover its entire extent in the variable dimension. The search inherits the I/O optimality of BBS algorithm.

The DN search inside q and in the partitions defined by vertices, faces and edges can be completed in a single R-tree traversal, similar to Sec. 3.3.3, which guarantees overall I/O optimality.

3.6.3 Beyond Three Dimensions

The geometry of a d-dimensional hyper-rectangle q includes d types of elements; e.g., in two dimensions we have vertices and edges (defining quadrants and stripes), while in three dimensions we additionally have faces. Every element has m fixed dimensions and $d - m$ variable ones, where $1 \leq m \leq d$. When $m = d$ the element is a vertex, and a skyline search retrieves exactly the DN set in the corresponding partition. In all other cases ($m < d$), an object r can only be disqualified by objects that dominate it *in the fixed dimensions*. Each of these dominating objects disqualifies the portion of r that overlaps with it in the variable dimensions. Note that this applies also to the $m = 1$ case (e.g., stripes in two dimensions, or faces in three) where there is a single fixed dimension, and dominance degenerates to closeness to q (thus the incremental NN search on the fixed dimension that we used previously for stripes and faces). Each of these partitions can be processed with BBS where a node or object is pruned if the already discovered DNs that dominate it in the fixed dimensions, collectively cover its entire extent in the variable dimensions. The DN search for all the partitions (and for DNs inside q) can be performed in a single R-tree traversal, thus guaranteeing I/O optimality.

Regarding K-DN search in higher dimensions, it follows the principles presented in this section. Under the same generalized definition of dominance (which takes into account fixed and variable dimensions), K-DN search disqualifies/prunes R-tree nodes and data objects that are dominated by at least K result objects found so far. All-DN extension is similar to Sec. 3.5, where all objects are loaded into a segment tree on one dimension, and an (incrementally populated) R-tree on the remaining dimensions plays the role of T_y to facilitate dominance checking.

3.7 Empirical Evaluation

In this section, we evaluate our algorithms using real and synthetic data. For the two-dimensional experiments, we obtained three real datasets from the R-tree portal[6], namely LB, TCB and CA. LB and CA contain 53K and 2.2M MBRs, respectively, of roads and streets in Long Beach and California. TCB contains 556K MBRs of residential blocks in four American states. The distribution in CA is highly skewed, whereas LB and TCB are more evenly spread out. The MBRs in each of them are treated as data objects in the experiments. To control the dataset cardinality, we also produced synthetic data with the generator of Theodoridis et al. from the R-tree portal[7]. The generated rectangles are distributed in a $[0, 10000][0, 10000]$ space uniformly, and have a side-length of 10 units on average.

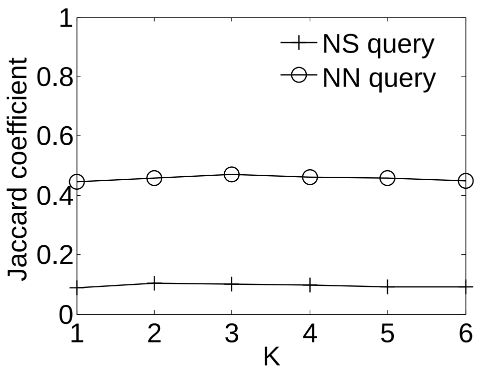
Our evaluation also includes higher-dimensional experiments. For these, we use the HOTEL dataset (from *hotelsbase.org*), which contains 418,843 hotel records with four attributes, namely stars, price, number of rooms, and number of facilities. We normalized the dataset to a $[0, 10000]^4$ space. Since hotel records correspond to points, we extended them to hyper-rectangles with average side-length of 10 units.

All the datasets are indexed with R*-trees [8], using a page size of 4KBytes. The algorithms are implemented in C++, and run on a Ubuntu machine with a 2GHz Intel Core Duo CPU and 2 GBytes of main memory.

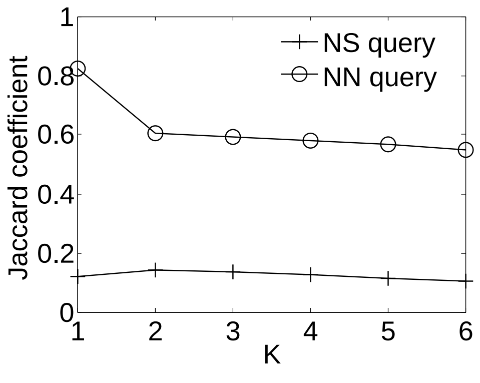
3.7.1 Comparison with Related Query Types

Before investigating processing performance, it is essential to quantify how different the results of this new query (DN) are from those of related query types, namely, nearest neighbors (NN) and nearest surrounders (NS). First, we process K-DN queries for $K = 1$ to $K = 6$ and record the result sets (in

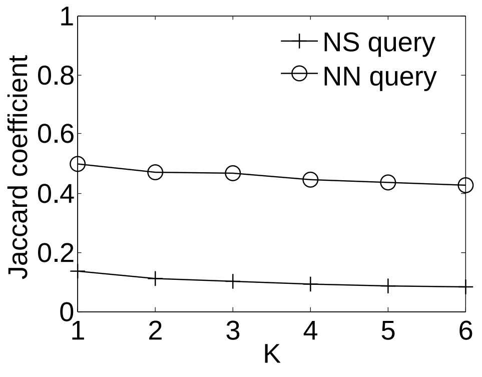
[6]http://www.rtreeportal.org
[7]http://www.rtreeportal.org/software/SpatialDataGenerator.zip



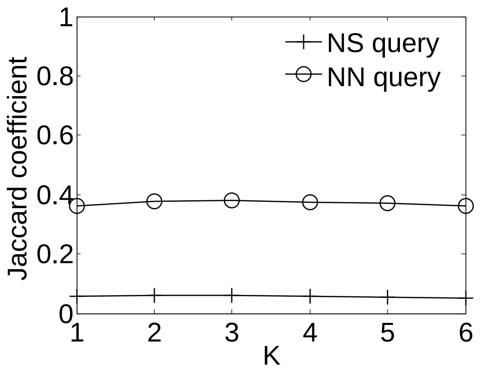
(a) LB



(b) TCB



(c) CA



(d) Synthetic

Figure 3.13: Jaccard coefficient of NN and NS sets w.r.t. DN results

this experiment we focus merely on the composition of the result). Then, we process NN and NS queries at (the centroid of) the same source objects as DN search[8]. In the NN case, we produce as many NNs as the cardinality of the corresponding K-DN set. In the NS case, we retrieve K-tier NSs for the same K value as the respective K-DN set.

To compare the result sets, we compute the Jaccard similarity coefficient, a standard means to measure similarity between sets [49]. The Jaccard coefficient of two sets A and B is defined as the cardinality of their intersection divided by the cardinality of their union, i.e., as $\frac{|A \cap B|}{|A \cup B|}$. Fig. 3.13 plots the Jaccard coefficient of NN and NS results with DN sets for different K values.

There are three key observations. First, NN sets have a higher similarity to DNs than NSs. A reason for this is that we "favor" NN by producing the same number of NNs as DNs (this number could not be known in advance, unless we

[8]Recall that there exist NS methods only for point sources in two-dimensional domains.

run a K-DN query first); on the other hand, K-tier NS sets have very different (i.e., larger) cardinality than K-DNs. Second, the more oblong the objects in a dataset, the more the NN and NS results deviate from the DNs. This explains why in LB, CA, and Synthetic the Jaccard similarity is considerably smaller that TCB – objects in TCB have an average aspect ratio of 1.908, whereas the average aspect ratios in LB, CA, and Synthetic are 6.631, 5.920, and 3.857, respectively. The final observation is that similarity generally drops for larger K. This is expected, because as K increases, so do the sizes of the compared result sets, and therefore the differences between the semantics of the queries become more pronounced.

Table 3.3: Plain DN results

Dataset	# of I/Os		CPU time (msec)	
	SDN	CNS	SDN	CNS
LB	29	61	24	306
TCB	17	129	61	313
CA	22	174	67	400
Synthetic	55	112	74	226

Table 3.4: Number of DNs and NSs

Dataset	Stripe DNs	Quad. DNs	NSs
LB	12	6	72
TCB	14	3	53
CA	11	10	69
Synthetic	11	13	126

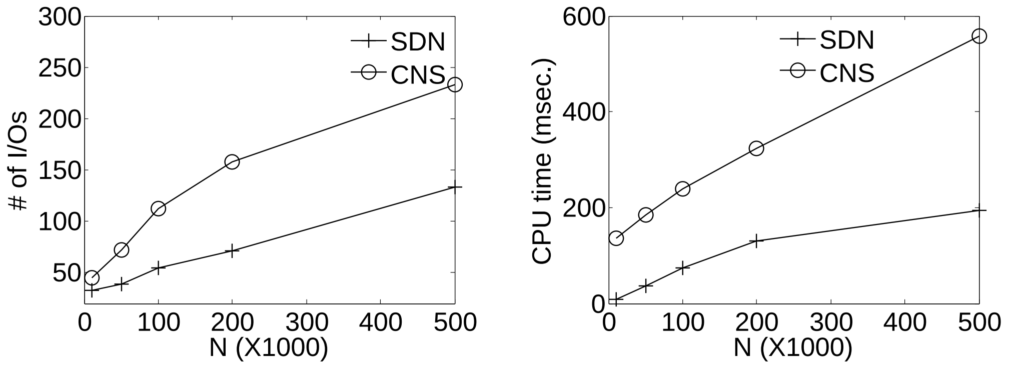
3.7.2 Experiments in Two Dimensions

Plain DN results. We first consider plain DN search in two dimensions. We compare our SDN-based algorithm versus the baseline CNS-based method, denoted in the charts as SDN and CNS, respectively. For fairness, we enhanced the latter with a single-traversal optimization (similar to that in Sec. 3.3.3) to avoid multiple reads of the same R-tree nodes. Moreover, its NS search component uses the most efficient algorithm proposed in [47] (termed *Sweep* in that work). Performance is measured in terms of I/O cost and CPU time. Note that due to the single traversal feature of both algorithms, the existence or not of a cache does not affect the I/O cost (because the R-tree nodes are accessed at most once).

For each line in Table 3.3, we use one of the four datasets as S and choose the source object at random among its rectangles. In this experiment, the cardinality of Synthetic is set to 100K. Every reported measurement is the average over 100 queries (at different source objects). CNS incurs 2 to 8 times more I/Os (in LB and CA, respectively). The reason is that the majority of quadrant NSs are false positives, i.e., they are not actually DNs. Table 3.4 illustrates the average number of stripe DNs, quadrant DNs, and NSs. The number of NSs is an order of magnitude larger than actual quadrant DNs, which implies a large false positive ratio in CNS and translates to a considerable number of (unnecessary) I/Os. Furthermore, turning again to Table 3.3, CNS requires significantly more CPU time. The reason is not only that there are too many NSs, but also that angular search in CNS is more complex than skyline computation.

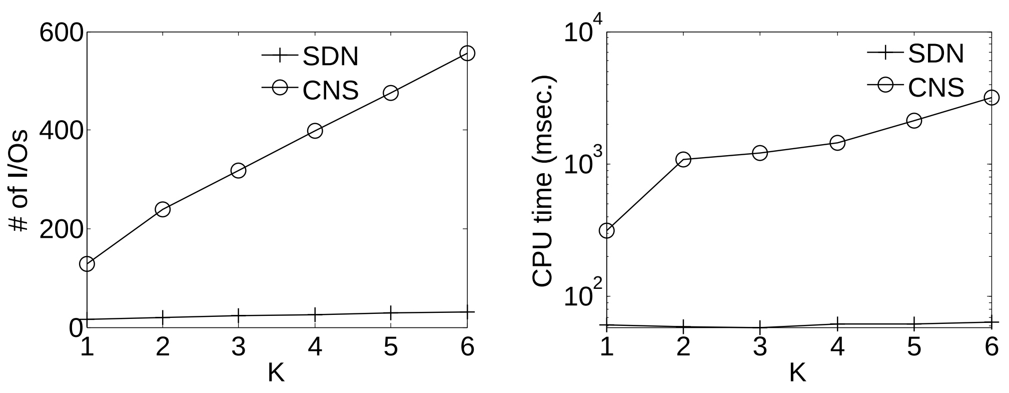
In Fig. 3.14, we examine the effect of dataset cardinality N on the performance of CNS and SDN. We use synthetic data to effectively vary N from 10K to 500K. The results show that although both algorithms incur proportionally higher costs with a larger N, SDN continues to maintain a substantial lead over CNS. At $N = 500$K, for instance, SDN incurs around half the I/O cost of CNS, and one third the CPU time.

***K*-DN results.** We evaluate SDN and CNS for K-DN search. For brevity, we present results only for the TCB dataset (the trends and relative performance for the other datasets are similar). In Fig. 3.15, we vary K from 1 to 6 and measure the I/O cost and CPU time of the algorithms. Each plotted value is the average over 100 randomly chosen source objects. CNS incurs one to two orders of magnitude more I/Os than SDN (in Fig. 3.15(a)). The gap



(a) # of I/Os (b) CPU time

Figure 3.14: Plain DN, effect of N (Synthetic dataset)



(a) # of I/Os (b) CPU time

Figure 3.15: K-DN search, effect of K (TCB dataset)

in CPU time is even wider (in Fig. 3.15(b)); CNS requires several seconds to process a DN query, whereas SDN spends less than 100msec in all cases. SDN's superior performance is due to its I/O optimal and CPU-efficient BBS search. In contrast, CNS wastes I/O and CPU time on the numerous false positives (NSs that are not DNs). The problem is exacerbated as K increases from 1 to 6 (corresponding to 1-tier to 6-tier NS retrieval); the average number of NSs found by CNS grows from 53 to 6593 per query, among which there are only 3 to 10 actual DNs.

Next, we repeat the previous experiment and examine the space requirements in the incremental version of SDN versus knowing K in advance[9]. Table 3.5 presents the peak memory consumption of the incremental and plain (i.e.,

[9]Note that the two versions have identical I/O cost and practically the same CPU time, i.e., the performance of incremental SDN matches the SDN curves in Fig. 3.15.

Table 3.5: Memory usage of Incremental vs. Non-Incr. SDN (KB)

SDN	$K = 1$	$K = 2$	$K = 3$	$K = 4$	$K = 5$	$K = 6$
Incr.	21.3	22.7	24.7	26.4	28.7	30.6
Plain	20.8	21.9	24.1	25.8	27.5	28.9

Table 3.6: All-DN results

Dataset	CPU time (sec)		Memory (MB)	
	ADN	sADN	ADN	sADN
LB	1	1714	3.6	2.6
TCB	40.5	25389	52.6	25
CA	67.8	49967	82.3	46.9
Synthetic	5.1	10823	9.5	4.8

non-incremental) SDN for different K values. The former utilizes only 2% to 6% more space. The dominant factor in space consumption is heap size, which is the same in both methods. The extra space in the incremental version is due to storing nodes and objects that would normally be pruned. Compared to the heap size, this overhead is minimal. When $K = 6$, for example, the maximum number of heap entries is 1165, while the number of pruned nodes/objects is 69.

All-DN results. We examine the performance of our all-DN algorithm described in Section 3.5, which we denote as ADN. For baseline, we construct a competitor, termed *straightforward ADN* (sADN), that invokes SDN for each object in the dataset. For fairness, our implementation of sADN utilizes Observation 2, i.e., DNs are computed only for two of the stripes and two of the quadrants. Here we focus on CPU time as the main performance metric, since both algorithms load the entire dataset in memory. We also measure the space requirements to verify practicality. As shown in Table 3.6, ADN achieves substantial performance gains over sADN for all datasets, owing to two factors.

First, for every object $r \in S$, sADN constructs four segment trees to store the DNs in the stripes. This is repeated for every object. ADN avoids this deficiency by maintaining two global segment trees. Furthermore, ADN finds the stripe DNs with negligible effort (recall that it identifies as south stripe

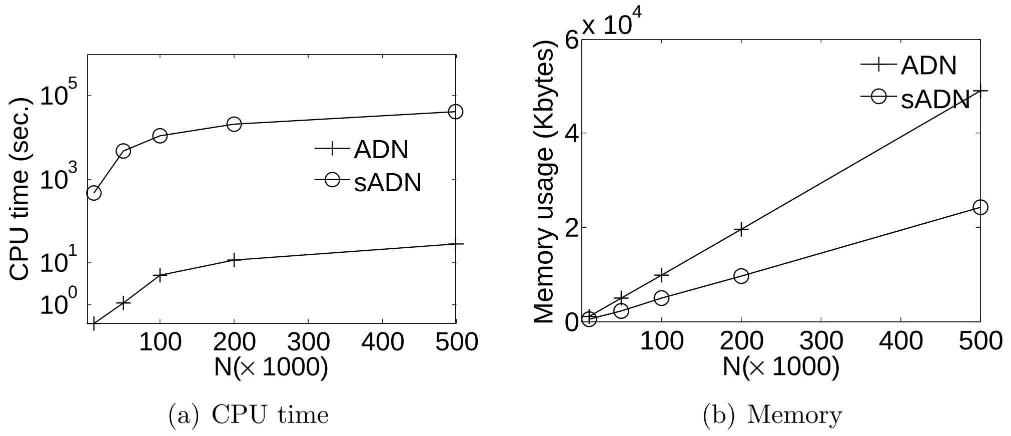


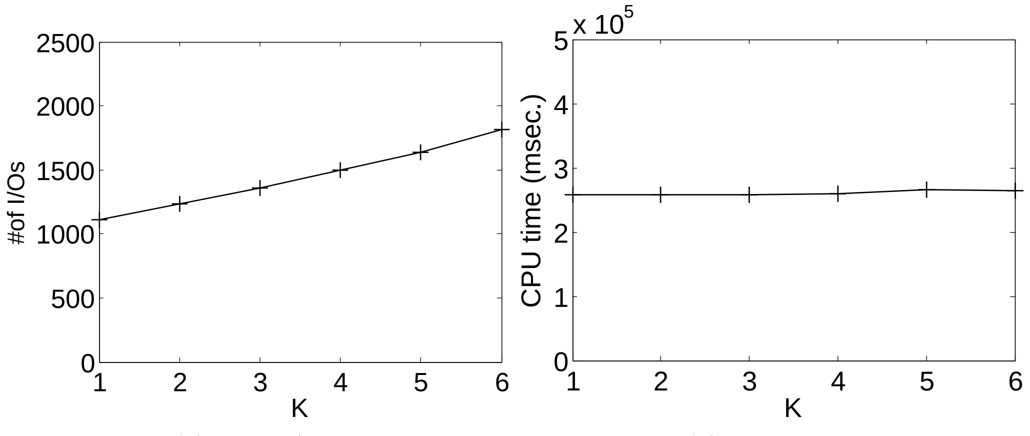
Figure 3.16: All-DN, effect of N (Synthetic dataset)

DNs every pair of successive objects in the leaf node lists in T_x). The west stripe DNs are also computed inexpensively, while inserting r into T_y.

Second, sADN needs to perform dominance checks in two quadrants of every object $r \in S$. In contrast, ADN finds the candidate SW (or NW) DNs of each object r_i with a linear scan of the leaf nodes in T_y that cover the y-extent between r_i and r_{i-1} (r_{i+1}, respectively), i.e., between r_i and the preceding (succeeding) object in the object list L_k^x that includes r_i. Filtering false positives is also performed with a simple comparison per candidate. Obviously, an arithmetic comparison is much cheaper than a dominance check. In LB, for instance, ADN scans 62 leaves in T_y per object on the average, whereas sADN performs 515 dominance checks per object. This leads to another significant gain for ADN.

Turning to the memory consumption in Table 3.6, we observe that ADN uses about twice the space of sADN. This is because ADN maintains two segment trees in memory. Even so, the space requirements of ADN are well within the capacity of modern PCs. For example, ADN occupies around 82 MBytes for the CA dataset, which contains 2.2M MBRs. As explained in Sec. 3.5.2, under strict memory constraints, we could resort to a disk-based segment tree [4].

Continuing the evaluation of ADN, in Fig. 3.16 we examine scalability with dataset cardinality N. ADN is consistently three orders of magnitude faster



(a) # of I/Os (b) CPU time

Figure 3.17: K-DN search in 3-D, effect of K (HOTEL dataset)

than sADN (between 1300 and 4500 times), with double the memory footprint.

3.7.3 Experiments in Higher Dimensions

The remaining experiments evaluate our methodology in higher dimensions. We use HOTEL as the four-dimensional dataset, and we also extract its first three dimensions to form the three-dimensional dataset. CNS does not apply here (because NS methods exist only for two dimensions). Hence, we focus on the nature of the problem and on the performance of SDN.

In Fig. 3.17 and 3.18 we assess the performance of SDN versus K on three-dimensional and four-dimensional data, respectively. The trend is similar to Fig. 3.15, but the cost is considerably higher than in two dimensions. The main reason is that the number of DNs increases with d, as elaborated in Sec. 3.6.1. For instance, the number of DNs ($K = 1$) in the two-dimensional synthetic dataset with 100K objects is 24, versus 673 in three dimensions (for the same dataset cardinality). In the HOTEL dataset, the number is 7308 in three dimensions, and 10385 in four dimensions. Another reason is that, for a fixed cardinality, space becomes sparser in higher dimensions [14], which causes the search area to expand. Finally, the R-tree structure itself degrades with dimensionality, thus reducing the effectiveness of pruning. For a given dataset and R-tree structure, however, SDN is I/O optimal, i.e., the shown

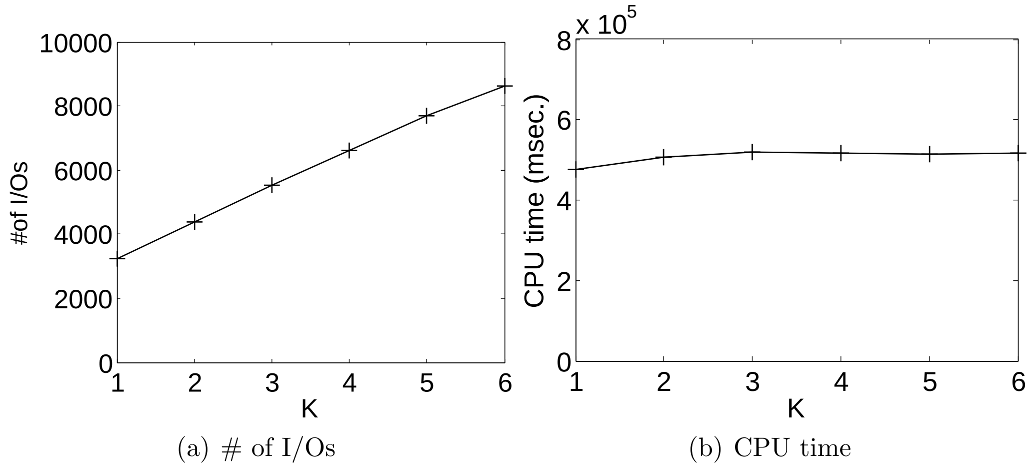


Figure 3.18: K-DN search in 4-D, effect of K (HOTEL dataset)

I/O cost is the smallest possible by any exact, non-precomputation algorithm.

3.8 Summary

In this chapter, we introduce a new query, called direct neighbor (DN) search, which can be used as a tool to provide the users with additional and valuable information for recommendation support. Two objects in a dataset are DNs if it is possible for a window query to overlap these objects and no other. A DN query retrieves all the DNs of a given source object. DN search and its variants, K-DN and all-DN, have wide applicability in competitor analysis. We present algorithms for DN, K-DN and all-DN search. Experiments on real and synthetic data verify that our algorithms vastly outperform baseline solutions built upon existing work.

Chapter 4

Global Immutable Region for Top-k Recommendation Support

A top-k recommendation, or equivalently top-k query, shortlists the k records in a dataset that best match the user's preferences. To indicate her preferences, the user typically determines a numeric weight for each data dimension (i.e., attribute). We refer to these weights collectively as the *query vector*. Based on this vector, each data record is implicitly mapped to a score value (via a weighted sum function). The records with the k largest scores are reported as the result. In this chapter we propose an auxiliary feature to standard top-k query processing. Specifically, we compute the maximal locus within which the query vector incurs no change in the current top-k result. In other words, we compute all possible query weight settings that produce exactly the same top-k result as the user's original query. We call this locus the *global immutable region* (GIR). The GIR can be a powerful tool for top-k recommendation support, and it can be used as a guide to query vector readjustments, as a sensitivity measure for the top-k result, as well as to enable effective result caching. We develop efficient algorithms for GIR computation, and verify their robustness using a variety of real and synthetic datasets.

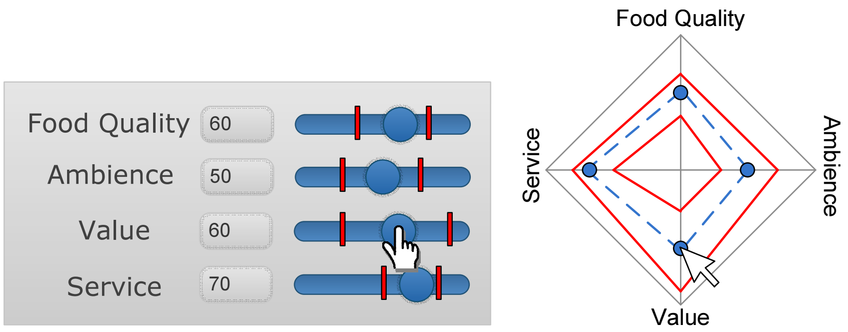
4.1 Motivation

Consider a service like HungryGoWhere.com or Yelp.com, where users rate and search for restaurants. The former, for instance, maintains for each registered restaurant the average user ratings in terms of *food quality*, *ambience*, *value for money*, and *service*. Users looking for restaurants base their decisions on these four factors, yet different users weigh each factor differently.

A user interested in dining options can provide a numeric weight for each decision factor and request for a personalized recommendation of, say, the top-10 restaurants according to her preferences. Through those weights, which we collectively refer to as the *query vector*, each restaurant is implicitly associated with a score value, computed as the weighted sum of its four average ratings. The online service may employ an off-the-shelf top-k processing algorithm to report the 10 restaurants with the largest scores.

In this work we propose to supplement the top-k result with a *global immutable region* (GIR). The GIR indicates all the possible weight settings for which the current top-k recommendation holds. For the common case of linear scoring functions, the GIR is a convex polytope in query space, wherein the query vector may freely shift without inducing any changes in the result. In our restaurant example, the query space involves four dimensions, each corresponding to the weight for a factor, e.g., the first axis refers to the weight w_1 for food quality, the second to the weight w_2 for ambience, etc; the GIR is a 4-dimensional polytope in that space. The GIR can be used to guide weight readjustment, for the purpose of sensitivity analysis, as well as for effective top-k result caching.

Suppose that the user in our restaurant example requests for a top-10 recommendation, using an interface like the text-boxes/slide-bars in Figure 4.1(a) or the radar chart in Figure 4.1(b), which captures preferences in the form of movable locations on each of the four axes. Assuming weights in the range from 0 to 100, the user requests for a top-10 recommendation by specifying



(a) Text input and slide-bars (b) Radar chart

Figure 4.1: Weight input and GIR-induced bounds

query vector $q = (60, 50, 60, 70)$ – this implies a weight $w_1 = 60$ for food quality, $w_2 = 50$ for ambience, etc. Should the user decide to explore alternative recommendations, she may change the weights and reissue the query. Primarily, she would want to avoid a blind readjustment that induces no change in the top-10 result. At the same time, she would need a sense of how drastically each weight affects the recommendation, so as to avoid overly radical changes. Using the GIR, we may derive a lower and an upper bound mark on each slide-bar (like those shown in Figure 4.1(a)) in between which the corresponding weight value induces no change in the result. Furthermore, we can inform the user what the new result will be at each of these bounds. Figure 4.1(b) represents the same bounds in the form of an inner and an outer solid polygon that connect the “tipping points” on the four axes.

GIR computation finds application in sensitivity analysis as well. Effective decision support involves providing the user with both a recommendation, and a measure of its robustness [23, 65]. For example, a robust top-k result would offer the user higher confidence in her decision, while a sensitive one would trigger deeper deliberation. An intuitive robustness measure for a top-k result is the ratio of the GIR volume to the volume of the entire query space. This ratio determines the probability that a randomly and uniformly generated query vector would have the same top-k result as the user’s query. The higher this probability, the more robust the result. This measure of robustness was proposed in [70], without however considering efficient approaches to compute it.

Another application of GIR computation is result caching. Suppose that previous top-k results are maintained, along with their GIRs. If the query vector of a new request falls within the GIR of a cached result, the latter can be directly reported. Even if k in the new query is larger than that of the cached result, it is still desirable to report the available (highest-scoring) recommendations immediately [72], before producing the rest of the top-k list. Note that this is orthogonal to top-k view materialization [26, 80], since the requested result either matches exactly a cached one or not.

In this chapter we develop scalable algorithms for GIR computation. We determine the conditions under which changes in the query vector invalidate the result, represent them in computational geometric terms, and make crucial observations that enable fast processing. Along the way, our GIR algorithms also compute the new top-k result should the query vector shift to any point on the GIR boundary. We verify the generality and efficiency of our methods using real and synthetic data with different characteristics.

4.2 Preliminaries

As the notion of GIR builds on the top-k query, we first review top-k processing. Next, we survey safe regions for spatial queries, per-dimension (local) immutable regions for top-k queries, and sensitivity analysis in operations research. We also briefly cover convex hull computation, a foundation for our algorithms.

Given a database D and a scoring function, a top-k query retrieves the k records from D that achieve the highest scores. Top-k queries have been studied in various domains, including relational databases [40], middle-ware information systems [32], joins [75], and dynamic databases [5, 42, 82].

The most relevant existing study is [53], which determines immutable regions on individual decision factors. An immutable region there takes the form of a validity interval for an isolated query weight, *assuming that all the other weights are kept constant.* One interval is defined for each decision factor. We term those *local immutable regions* (LIRs) to distinguish from the GIR. In the context of Figure 4.1, [53] produces the same original marks and inner/outer polygons. However, due to the local nature of the LIRs, it cannot support simultaneous readjustments to multiple weights. More importantly, if a weight w_i is updated, the immutable regions for all the other factors are invalidated, *even if the new value of w_i remains within its LIR.* Referring to Figure 4.1 again, if w_3 shifts to 40 (which is still inside its permissible range) the technique in [53] needs to compute from scratch new LIRs for all the remaining factors. At the heart of LIR computation lie a pruning and a thresholding technique, both of which are tailored to LIRs and are inapplicable to GIR formation. Note that we may trivially derive LIRs from the GIR (as we discuss in Section 4.7.3), but the reverse does not hold.

Another related work is [70] which considers uncertain scoring functions and proposes methods to compute representative top-k results. It also introduces two sensitivity measures, STB (i.e., stability of an ordering wrt. weights) and LIK (i.e., ordering likelihood). Given a top-k query with a linear scoring function, STB computes the largest ball around the query vector (in query space) where the top-k result remains the same. This ball is enclosed in (i.e., a subset of) our GIR, the latter being the *maximal* locus that preserves the result. Moreover, STB requires a scan of the dataset, which is prohibitive for large disk-resident data.

LIK defines a sensitivity measure that is equivalent to the ratio of the GIR volume to the volume of the query space. Other than the definition, however, [70] is not concerned with efficiency. It sketches an approach based on half-space[1] intersection that scans the entire dataset. With a time complexity of $O(n^{2^{d-2}})$, it is impractical for sizable databases. In Section 4.3.3 we sketch a straightforward approach to compute the GIR which, although it has a superior complexity of $O(n^{d/2})$ (compared to LIK), it is still hugely impractical, thus motivating the elaborate techniques in this chapter.

In location-based services, while processing spatial queries such as nearest neighbors and window queries, servers face the problem of frequent index maintenance and result re-computation as data objects move around and update their locations. Safe region techniques are designed to alleviate this problem by assigning to each mobile object an area, known as *safe region*, within which it is guaranteed not to alter the result of any spatial query in the system [60, 55]. Safe region techniques are inapplicable to our problem, since they consider the notion of spatial proximity, as opposed to the score-based ranking involved in top-k processing.

Another topic that is related to GIR is sensitivity analysis in operations research, which addresses how changes in the input parameters affect the output of a model [67, 37]. This includes studying to what extent the input is allowed to change, so that the output continues to be optimal. There are two approaches. The first, one-factor-at-a-time (OFAT), varies one input factor while fixing the rest. Although simple to use, OFAT is known to miss out optimal combinations, as it cannot capture the interaction among multiple input parameters. This motivates the second approach, named multi-parameter sensitivity analysis (MPSA). MPSA varies multiple input factors simultaneously and observes the changes (in terms of variance) in the model output under the combined influence of the inputs. Techniques for MPSA include standardized regression, differential analysis, factorial experimental design, and Fourier amplitude sensitivity test. OFAT is also known as local sensitivity analysis, and MPSA as global sensitivity analysis. In a sense, the LIRs in [53] are analogous

[1]A half-space is either of the two parts into which a hyperplane divides a coordinate space. In two dimensions, it is a half-plane.

to OFAT, while our GIR to MSPA. Both the OFAT and MPSA techniques, however, consider the effect of different inputs to the variance of a value (i.e., the model's output). In our problem, instead, the output changes refer to updates in the order or composition of a top-k result.

Our solutions touch upon and employ convex hull computation algorithms. The convex hull of a dataset D in a d-dimensional space is the smallest convex set that encloses all the records in D. Given a set S of points in d-dimensional space. For any subset $S' = \{p_1, ..., p_k\} \subseteq S$, and any nonnegative numbers $\lambda_1, ..., \lambda_k$ such that $\sum_{i=1}^{k} \lambda_i = 1$, if the combination $\sum_{i=1}^{k} \lambda_i p_i \in S$, then S is a convex set. In two dimensions, the hull is a convex polygon, whereas in higher dimensions it is a convex polytope [13]. The boundary of the hull is represented by vertices (i.e., data records) and facets. Algorithms for efficient convex hull computation in two and three dimensions include gift-wrapping [19], Graham's scan [35], and Chan's algorithm [18]. For higher dimensions, Quickhull [7] and Clarkson's algorithm [24] are the most common, with a time complexity of $O(n^{d/2})$. Our solutions share some of the key operations in Clarkson's algorithm. Its crux is to incrementally build the hull by processing the data records one by one. If the current record lies above one or more facets of the hull (i.e., it is not enclosed by the hull), these facets are replaced by new ones that include the new record.

4.3 Computing Global Immutable Region

4.3.1 Definition of Global Immutable Region

We consider top-k processing in a low-dimensional space. The dataset D consists of n records. Each record $p \in D$ has an identifier and d numeric attributes $x_1, x_2, ..., x_d$ (also referred to as *dimensions*). The top-k query is defined by a vector of weights $q = (w_1, w_2, ..., w_d)$, called the *query vector*, which can be seen as a d-dimensional point. For ease of presentation, we assume that the data space and query space are normalized, i.e., data attributes and query weights have values in the range $[0, 1]$. The score of a data record p with respect to q is given by the dot product $\mathcal{S}(p, q) = q \cdot p = \sum_{i=1}^{d} w_i x_i$. This definition of $\mathcal{S}(p, q)$ is equivalent to what is also commonly referred to as a linear scoring function. The result R of the query is a list of k records with the highest scores in D, sorted in decreasing score order. In other words, $R = \{p_1, p_2, ..., p_k\}$ where p_i is the record with the i-th highest score ($1 \leq i \leq k$).

When the query vector q changes, we say that R is *preserved* if both its composition and the score order among its members are unaltered. The problem we address in this chapter is the computation of the maximal locus in the query space where R is preserved. We call this locus the *global immutable region* (GIR) of q.

Definition 3 ***Global immutable region (GIR)***. *Given a dataset D and a top-k query q with result $R = \{p_1, p_2, ..., p_k\}$, the GIR is the locus of all vectors q' in query space where*

1. $\mathcal{S}(p_i, q') \geq \mathcal{S}(p_{i+1}, q')$ *for each* $i \in [1, k)$, *and*
2. $\mathcal{S}(p_k, q') \geq \mathcal{S}(p, q')$ *for every record* $p \in D \backslash R$.

4.3.2 Nature of Global Immutable Region

A first step to understanding the problem is to determine the shape of the GIR. For the score order within result R to be preserved, $k - 1$ conditions must hold, each of the form $\mathcal{S}(p_i, q') \geq \mathcal{S}(p_{i+1}, q')$ (for $i \in [1, k)$). For the k-th result record p_k to remain ahead of all the non-result records p, another $n - k$ conditions must hold, each of the form $\mathcal{S}(p_k, q') \geq \mathcal{S}(p, q')$. Due to the form of the scoring function, each of these $n - 1$ conditions corresponds to a half-space in query space, whose defining hyperplane passes through the origin[2]. The

[2]Consider for example condition $\mathcal{S}(p_k, q') \geq \mathcal{S}(p, q')$, which in dot product notation is $p_k \cdot q' \geq p \cdot q' \Rightarrow (p_k - p) \cdot q' \geq 0$. Since vector $(p_k - p)$ is fixed (with attribute values in both records being constant), the inequality implies that q' lies on a half-space that is bounded by hyperplane $(p_k - p) \cdot q' = 0$. The latter passes through the origin of the query space.

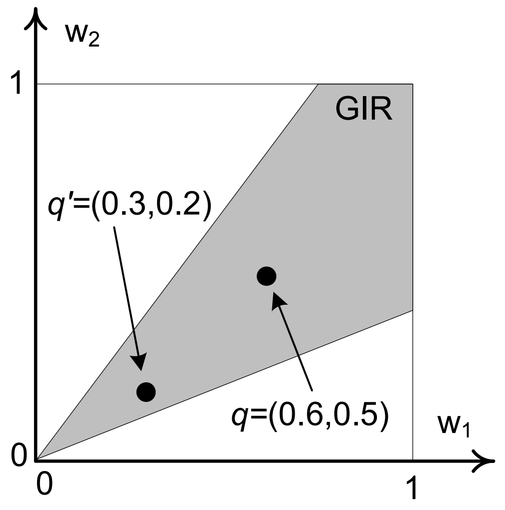


Figure 4.2: GIR example in 2-dimensional (query) space

top-k result is preserved if and only if the query vector remains within the intersection of these $n-1$ half-spaces. This intersection constitutes the GIR of the query. Any intersection of half-spaces (and therefore the GIR too) is by definition a convex polytope.

Figure 4.2 shows how the GIR looks in 2-dimensional space (i.e., $d = 2$). Query vector $q = (0.6, 0.5)$ represents the user's original weight setting. In two dimensions, each of the $n-1$ conditions derived from Definition 3 corresponds to a half-plane (instead of a half-space), whose defining line (instead of defining hyperplane) passes through the origin. Their intersection (i.e., the GIR) is a wedge like the one shown shaded in the figure. Any query vector lying inside this area (like the depicted $q' = (0.3, 0.2)$) is guaranteed to preserve the top-k result. Furthermore, this is the maximal locus in the query space where result R is preserved.

Without loss of generality, each line that bounds the GIR corresponds to one of the original $n-1$ conditions. This implicitly determines what the new top-k result will be if the query shifts to a particular line. For instance, assume that the upper bounding line of the GIR corresponds to condition $\mathcal{S}(p_k, q') \geq \mathcal{S}(p, q')$, where p is a non-result record. If the query is adjusted to fall on this line, the new result will be the same as the current one, except that p will replace the k-th record in R. If the bounding line corresponds to condition $\mathcal{S}(p_i, q') \geq \mathcal{S}(p_{i+1}, q')$, the resulting update in R is that p_{i+1} overtakes record p_i in score, i.e., the perturbation is a reordering between p_i and p_{i+1} in the result.

In the following we focus on the GIR computation. *Determining the exact result perturbation when the query moves to the boundary of the GIR happens along the way, by identifying the record responsible for each of the half-spaces that bound the GIR.*

4.3.3 Challenges, Assumptions and Setting

Our goal is to develop efficient GIR computation algorithms that are scalable to large datasets. The discussion in the previous session hints at a possible GIR computation approach, based on half-space intersection. However, deriving the $n - 1$ half-spaces stated in Definition 3 requires scanning the entire dataset, and incurs a large data access cost. Furthermore, performing the intersection of $n - 1$ half-spaces requires an excessive amount of computations, specifically, $\Omega(n^{d/2})$ which explodes for large datasets. The main challenge we address is how to reduce the number of records/half-spaces considered so as to minimize (i) the data access cost to retrieve those records, and (ii) the processing time for half-space intersection, while still guaranteeing correct/exact GIR computation.

Targeted at large scale, low-dimensional datasets, we assume that D is indexed by a spatial access method. Our implementation employs the ubiquitous R*-tree [9], though our techniques apply directly to other space- or data-partitioning indices. The index and data could reside in memory or on disk, the latter being our default setting (although we evaluate our techniques in both scenarios).

When a query q is posed, prior to GIR computation, its top-k result R must be retrieved. For this we employ the state-of-the-art BRS technique [73],

Symbol	**Description**
d	Data dimensionality
D	Dataset in $[0,1]^d$
n	Number of records in D
p	A data record in D
q	User query (vector in $[0,1]^d$)
$\mathcal{S}(p,q)$	Score of p w.r.t. q
R	Top-k result
p_i	The i-th record in R $(1 \le i \le k)$
$\mathcal{SL}$	Skyline of $D \backslash R$
$\mathcal{CH}$	Convex hull of $D \backslash R$
$\mathcal{CH}'$	Convex hull of set $\{p_k\} \cup D \backslash R$

Table 4.1: Notation

yet our work is not directly dependent on the choice of top-k algorithm. To facilitate subsequent GIR-related processing, we maintain all the data records encountered by BRS (but not included in the top-k result), as well as its search heap.

After the top-k result is produced, GIR computation commences. That comprises two phases. The first derives an interim GIR based on the first set of conditions in Definition 3 (considering only records in R). The second phase shrinks the interim GIR according to the second set of conditions (imposed by non-result records).

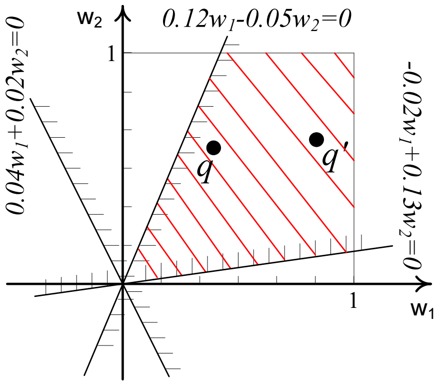
The examples given in this chapter may refer to either the query space or the data space. Thus, we explicitly indicate in the figure captions which of the two spaces is considered. In Table 4.1 we summarize the notation used in the chapter.

4.4 Processing in Phase 1

Given the original top-k result R, the first phase derives an interim GIR from the first set of conditions in Definition 3. There are $k-1$ conditions, each defining a half-space in query space. The interim GIR in Phase 1 is obtained by intersecting these half-spaces. Formally, the interim GIR is the following

	x_1	x_2	$p \cdot q$
p_1	.54	.5	.516
p_2	.5	.48	.488
p_3	.52	.35	.418
p_4	.4	.4	.4

(a) Top-k records



(b) Half-planes in query space

Figure 4.3: Phase 1 example ($k = 4$, $d = 2$)

polytope in query space:

$$\bigcap_{i=1}^{i=k-1} \{q' | q' \in [0,1]^d \text{ such that } (p_i - p_{i+1}) \cdot q' \geq 0\} \tag{4.1}$$

Consider a query with $q = (0.4, 0.6)$ and $k = 4$. Suppose that a top-k algorithm (BRS in our implementation) reports in the result records p_1, p_2, p_3, and p_4, whose attributes and scores are shown in Figure 4.3(a). Phase 1 commences by deriving the half-plane that preserves the order between p_1 and p_2, i.e., half-plane $(p_1 - p_2) \cdot q' \geq 0 \Rightarrow 0.04w_1 + 0.02w_2 \geq 0$. Figure 4.3(b) represents the half-plane by its bounding line ($0.04w_1 + 0.02w_2 = 0$) in the query space. Similarly, preserving the order between p_2 and p_3 defines the half-plane $(p_2 - p_3) \cdot q' \geq 0 \Rightarrow -0.02w_1 + 0.13w_2 \geq 0$. In turn, p_3 and p_4 define $(p_3 - p_4) \cdot q' \geq 0 \Rightarrow 0.12w_1 - 0.05w_2 \geq 0$. The interim GIR is the striped area at the intersection of the three half-planes. Any query vector in this area is guaranteed to uphold the score order among the four result records. We use a 2-dimensional example here for ease of presentation. The process is very similar in higher dimensions.

Phase 1 is fast because the number of result records k is typically much smaller than the number of non-result records that need to be considered. It is also uniform across all methods in our framework. The distinction among them lies entirely in Phase 2, the bottleneck in GIR computation.

4.5 Basic Methods for Phase 2

Phase 2 further shrinks the GIR to ensure that no non-result record can overtake the k-th result record p_k in score. As explained in Section 4.3, the efficiency of this phase hinges on the ability to reduce the number of non-result records examined. In this section, we present two basic methods for pruning the set $D \backslash R$ to safely discard records that could not affect the GIR.

4.5.1 Skyline Pruning Method

The first method relies on the *skyline* operator [16]. The skyline of a set includes only those members that are not *dominated* by any other member. In our context, we say that record p dominates another record p' if the value of p in every dimension is no smaller than the corresponding value of p', and the two records differ on at least one dimension. Due to the definition of dominance, record p could have a score no smaller than p' under any monotone scoring function [40] (which is a superclass of the linear scoring functions we assume). Since the score of p' never exceeds that of p regardless of the query vector, record p' cannot overtake the current p_k before p overtakes p_k. In other words, $\mathcal{S}(p, q') \geq \mathcal{S}(p', q')$ for any $q' \in [0,1]^d$ so, by satisfying the condition $\mathcal{S}(p_k, q') \geq \mathcal{S}(p, q')$, our GIR automatically also upholds $\mathcal{S}(p_k, q') \geq \mathcal{S}(p', q')$. Hence, it is safe to ignore p' in GIR computation.

To generalize, we may safely prune the set $D \backslash R$ by retaining only the records in its skyline $\mathcal{SL}$. The final GIR can be derived by intersecting the interim GIR from Phase 1 with the half-spaces formed from inequalities $(p_k - p) \cdot q' \geq 0$ for each record $p \in \mathcal{SL}$. We term this approach *Skyline Pruning* (SP). Figure 4.4 shows a 2-dimensional example where $k = 2$, the dataset comprises records $p_1, p_2, ..., p_{15}$, and the result includes p_1 and p_2. The skyline $\mathcal{SL}$ of the non-result records includes $p_3, p_4, ..., p_9$, which are the only records considered by SP in Phase 2. Records that fall in the shaded area are dominated by at

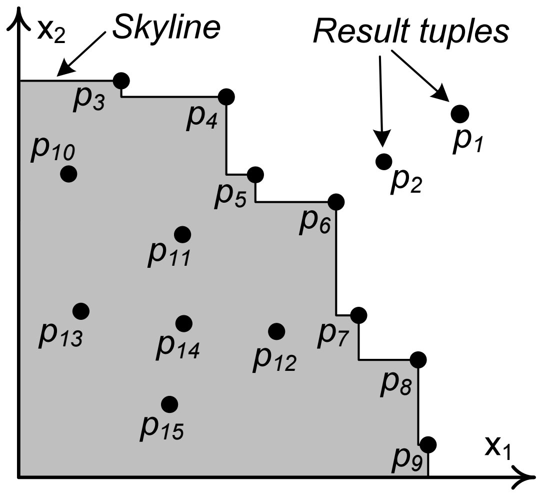


Figure 4.4: SP example ($k = 2, d = 2$, data space)

least one member of the skyline.

Although SP can potentially disqualify many records from $D\backslash R$, the cardinality of the skyline may still be large. As an indication, the number of records in $\mathcal{SL}$ is in the order of $O((\log n)^{d-1})$ [10] for independent data, while there are common (e.g., anti-correlated) distributions where the cardinality is even higher.

In terms of implementation, the state-of-the-art algorithm for skyline computation on spatial access methods is BBS [58], which follows the *branch-and-bound* paradigm. It utilizes an R-tree on the dataset to incrementally retrieve nearest neighbors (NNs) to the top corner of the data space, i.e., to point $(1, 1, ..., 1)$. The first NN is guaranteed to be in the skyline, and is used to prune the part of the space it dominates. Then, the next NN is retrieved in the remaining part of the space; it is also included in the skyline and used to further prune the search space. The process continues until no more NNs can be found in the non-dominated part of the space.

To see how BBS applies in our situation, recall that (before GIR computation) the top-k result was produced by BRS, and that we have retained its search heap and all the non-result records that were encountered during its execution. We initialize $\mathcal{SL}$ by computing the skyline of the encountered non-result records (using any main memory algorithm [16]), and then invoke BBS on the retained search heap to consider records that may belong to $\mathcal{SL}$ but were not accessed during BRS execution. Recall that the search heap of BRS is organized on *maxscore*. That is, in our BBS execution, instead of incrementally retrieving NNs to the top corner of the data space, we retrieve records in decreasing $\mathcal{S}(p, q)$ order. This does not affect the correctness of BBS, since the distance from the top corner of the data space (in vanilla BBS) can be replaced by any monotone scoring function to determine the retrieval order [58]. Another modification is that any record p retrieved by BBS is inserted into $\mathcal{SL}$ only if it is not dominated by any of its members, while if p dominates any existing members, the latter are removed from $\mathcal{SL}$.

4.5.2 Convex Hull Pruning Method

Our second basic solution for Phase 2 prunes non-result records based on the concept of the convex hull. If we treat the records of a dataset as points in d-dimensional space and compute their convex hull, the geometric properties of the hull guarantee that the top-1 record under any linear scoring function (defined over the same d dimensions) lies on the hull[3] [50, 20].

Let $\mathcal{CH}$ be the convex hull of the non-result records. The above property guarantees that for any query vector $q' \in [0, 1]^d$ and any record p' that is strictly enclosed by $\mathcal{CH}$ (as opposed to lying on it), there is at least one record p on the hull (i.e., $p \in \mathcal{CH}$) such that $\mathcal{S}(p, q') \geq \mathcal{S}(p', q')$. This implies that p' cannot overtake the current k-th result record p_k until some record on the hull has overtaken p_k. Hence, only records on $\mathcal{CH}$ could affect the GIR.

Utilizing the above observation directly would prune $D\backslash R$ by retaining only those records that fall on its convex hull. Nevertheless, we can do better. To exemplify, Figure 4.5 illustrates the convex hull of the non-result records, assuming the same setting/dataset as Figure 4.4. The records that lie on (as opposed to inside) $\mathcal{CH}$ are $p_3, p_4, p_6, p_8, p_9, p_{15}, p_{13}, p_{10}$. However, we observe

[3]When we say that a record "lies on" or "belongs to" the hull, we mean that it lies *on the boundary* of the hull.

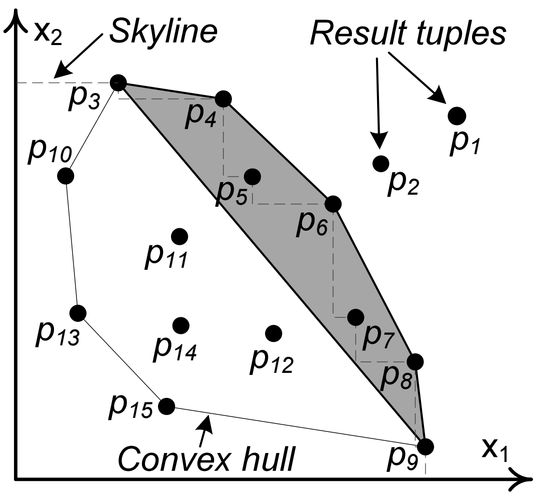
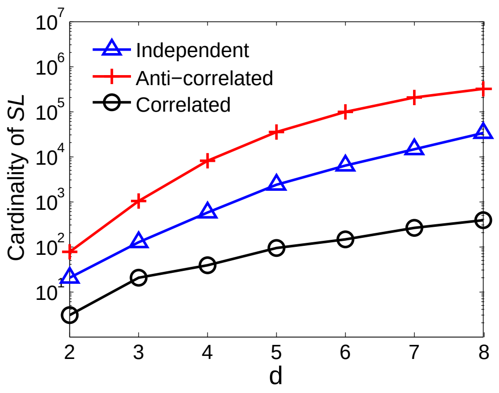


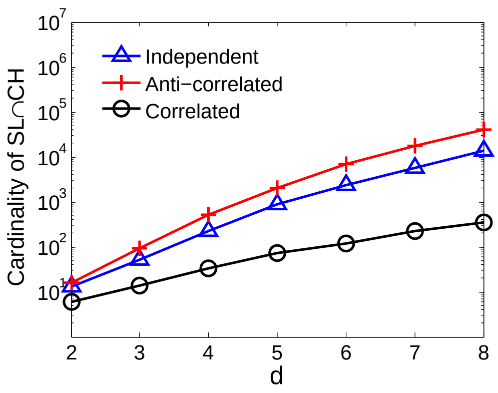
Figure 4.5: CP example ($k = 2, d = 2$, data space)

that p_{15}, p_{13}, p_{10} are dominated by p_4, and could therefore not affect the GIR (by the same reasoning as in SP). This observation gives rise to our second baseline algorithm, *Convex Hull Pruning* (CP), which considers in Phase 2 only those non-result records that belong to the skyline and at the same time fall on the convex hull of $D \backslash R$, i.e., records $p \in \mathcal{SL} \cap \mathcal{CH}$. Referring to the example in Figure 4.5, CP considers only records p_3, p_4, p_6, p_8, p_9.

In implementing CP, we first retrieve the skyline of $D \backslash R$, using the same BBS-based approach as SP. Following that, we compute the convex hull *of the skyline records only* (using Clarkson's algorithm [24]). This hull is shown shaded in the example of Figure 4.5. The records that lie on the hull are used for half-space intersection with the interim GIR from Phase 1 to derive the final GIR. An alternative approach would be to compute the convex hull before disqualifying the dominated among its records. This would be inefficient, because it would access parts of the space that are too far (and irrelevant) from the GIR, like the vicinity of p_{15}, p_{13}, p_{10} in Figure 4.5. Moreover, the only existing convex hull algorithm that utilizes a spatial index [15] applies only to 2-dimensional space.



(a) Cardinality of $\mathcal{SL}$



(b) Cardinality of $\mathcal{SL} \cap \mathcal{CH}$

Figure 4.6: SP and CP effectiveness ($n = 1$M, $k = 20$)

4.5.3 Performance Indications

We now provide preliminary indications of the performance (and shortcomings) of SP and CP. We use three types of synthetic datasets (independent, anti-correlated, and correlated) with cardinality 1M each. We defer the description of these data, but note that they are standard benchmarks for preference-based queries.

Figure 4.6(a) plots, for different dimensionalities, the number of records that belong to the skyline of $D \backslash R$, i.e., the non-result records that SP needs to process in Phase 2. As anticipated, although SP prunes a large fraction of $D \backslash R$, it still needs to consider numerous records. The problem is exacerbated with growing d.

CP retains only the records in $\mathcal{SL} \cap \mathcal{CH}$, i.e., a subset of those examined in SP. In Figure 4.6(b) we present the number of records remaining after CP pruning in the same setting as Figure 4.6(a). CP aggressively reduces the number of non-result records in Phase 2. However, its effective pruning comes at the price of a convex hull computation over $\mathcal{SL}$, which entails substantial processing time.

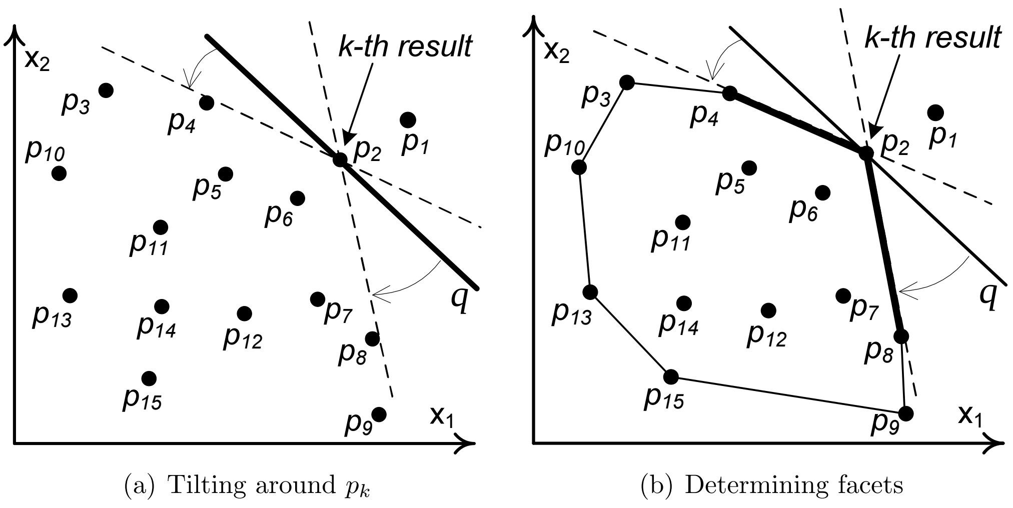


Figure 4.7: Intuition behind FP ($k = 2, d = 2$, data space)

4.6 Advanced Solution for Phase 2

The shortcomings of SP and CP motivate the development of an efficient and scalable Phase 2 algorithm that also copes better with dimensionality. We call this method *Facet Pruning* (FP). To provide the intuition behind it, we explore the nature of top-k query.

4.6.1 Rationale of Facet Pruning Method

In Figure 4.7(a) we assume the same dataset and top-k query as in Figures 4.4 and 4.5. Computing the top-k result can be seen as scanning the data space from its top corner (i.e., point $(1, 1)$) towards the origin, with a line (or hyperplane, in higher dimensions). The orientation of the line is fixed, and it is determined by the query vector[4]. The first encountered record has the highest score, the second encountered record has the next highest score, and so on. The search stops after finding k data records, which form the result R.

In our example, the sweeping line would first encounter p_1 and stop when it hits p_2 (recall that $k = 2$). The line position when it hits $p_k \equiv p_2$ is defined by equation $w_1x_1 + w_2x_2 = \mathcal{S}(p_k, q)$ and is drawn in bold in Figure 4.7(a).

[4] Vector q is a *normal vector* to the sweeping line (or hyperplane, for $d > 2$), meaning that it is perpendicular to the line (or hyperplane, respectively).

It partitions the data space into two regions – all records above the line have higher scores than p_k and belong to R, whereas records below the line have lower scores (than p_k) and are not among the top-k. Any records on the line have the same score as p_k (yet, without loss of generality, we assume that there are no ties).

For now, let us ignore reorderings within R, and focus on the second set of conditions in Definition 3, which ensure that no non-result record overtakes p_k in score. Returning to Figure 4.7(a) and the nature of top-k processing, a tilt in the orientation of the sweeping line is equivalent to a shift in the query vector. Assume that we pin the sweeping line at p_k but allow it to rotate. A rotated position of the line is *permissible* if it keeps all non-result records below it. This implies that p_k still scores higher than them, and the query vector q' that corresponds to the new line orientation preserves R.

Consider a clockwise rotation in Figure 4.7(a). The first record hit by the line (i.e., p_8) bounds the permissible clockwise rotations, because any further tilting would perturb the result (p_8 would score higher than the current $p_k \equiv p_2$). Similarly, the permissible anticlockwise rotations are bounded by p_4. Any other non-result record cannot provide a stricter rotation bound than p_4 and p_8.

We make a crucial observation that hints at a general methodology to identify records like p_4 and p_8. In Figure 4.7(b) we show the convex hull of set $\{p_k\} \cup D \backslash R$ (i.e., the set of non-result records extended by p_k). The two facets that are incident[5] to p_k on the hull (i.e., facets $\overline{p_4, p_2}$ and $\overline{p_2, p_8}$) correspond to the records of interest (p_4 and p_8, respectively).

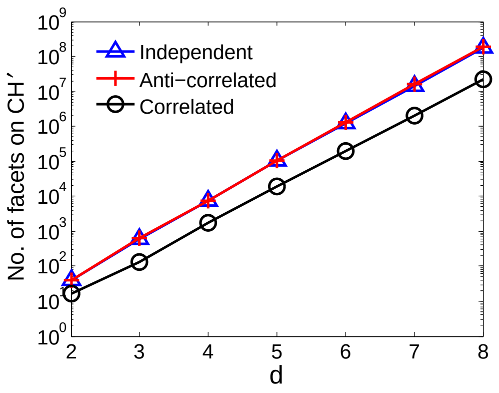
This is aligned with the property of the convex hull that each of its facets keeps all the records on one of its sides (specifically, the one toward the interior of the hull), which holds for any dimensionality [13]. Since the sweeping line

[5]A record and a facet are incident to each other if the record lies at a corner of the facet. In two dimensions, for example, a facet is a line segment, and it is incident to the two records at its endpoints.

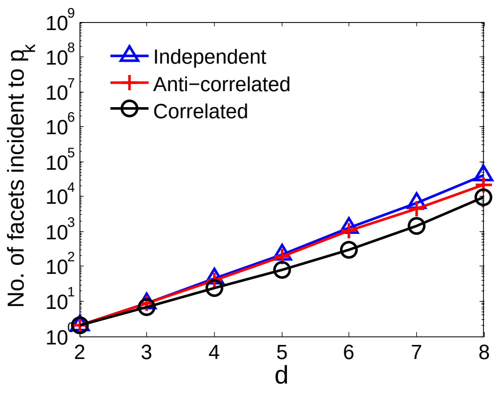
(or the sweeping hyperplane, in higher dimensions) is pinned at p_k, its permissible orientations are determined only by *the facets of the convex hull that are incident to p_k*. We call the records that are incident to those facets *critical*. They are the only non-result records that could affect the GIR.

Intuitive as this fact may sound, translating it into an efficient Phase 2 algorithm is challenging. Let $\mathcal{CH}'$ denote the convex hull of set $\{p_k\} \cup D \backslash R$. A naïve implementation would compute $\mathcal{CH}'$, get the hull facets that are incident to p_k, collect their incident records (i.e., the critical records), and derive the GIR using only these records for half-space intersection. Convex hull computation, however, has a time complexity of $\Omega(n^{d/2})$, which is equivalent to the complexity of the exhaustive half-space intersection described in Section 4.3.3 (actually, convex hull computation and half-space intersection are dual to each other [13]). The situation becomes worse if one considers that there is no off-the-shelf convex hull algorithm for disk-resident data in more than two dimensions.

To alleviate the problem, our FP approach computes *only the relevant part of the convex hull*, i.e., only the hull facets that are incident to p_k. This approach provides scalability with respect to both dataset cardinality and dimensionality. We thoroughly demonstrate this fact in Section 4.8, yet here we provide some preliminary empirical evidence that substantiates the FP rationale. In Figures 4.8(a) and 4.8(b) we plot the total number of facets in $\mathcal{CH}'$ and the number of facets that are incident to p_k, for the same setting as Figure 4.6. The charts suggest that FP needs to compute/consider only a very small fraction of the hull facets. We remark that a critical record may be incident to more than one facet, i.e., the number of critical records may be smaller than the number of facets shown in Figure 4.8(b). For example, for independent data and $d = 4$, there are 45 facets incident to p_k and 16 critical records, while for $d = 6$ the number of incident facets is 1258 and that of critical records is 98.



(a) Facets on $\mathcal{CH}'$



(b) Facets incident to p_k

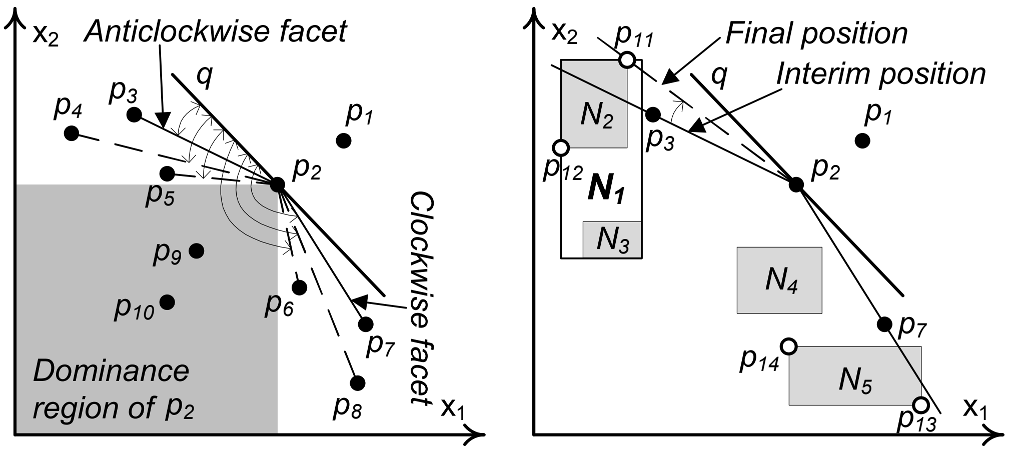
Figure 4.8: FP effectiveness (n = 1M, k = 20)

Next, we present the detailed FP algorithm. We distinguish between the FP versions for $d = 2$ (discussed in Section 4.6.2) and $d > 2$ (covered in Section 4.6.3), because the nature of 2-dimensional space allows for special-purpose enhancements.

4.6.2 Facet Pruning in Two Dimensions

The convex hull/incident facet observation exemplified in Figure 4.7(b) is general and particularly useful for $d > 2$. In the special case of 2-dimensional space, however, the visualization in Figure 4.7(a) already suggests an effective processing methodology. That is, FP needs to simply identify the two records that constrain the rotation of the sweeping line around p_k in the clockwise and anticlockwise directions. Recall that some of the records in $D \backslash R$ have already been fetched from the disk by BRS during the initial top-k computation, and kept in memory, as explained in Section 4.3.3. Let T denote this set of records. The remaining non-result records are on the disk and are accessible via the R-tree on D.

FP consists of two steps. The first considers T and identifies two *candidate* critical records or, equivalently, *interim* facets incident to p_k. The second refines those facets by exploring data from the disk (using the index), until it identifies the two actual critical records.



(a) First step (b) Second step

Figure 4.9: FP example ($k = 2, d = 2$, data space)

The first step starts by removing from T the records that are dominated by p_k, as they cannot overtake it in score under any query vector $q' \in [0,1]^2$. Then, it angularly sorts the remaining records in T. That is, for every record $p \in T$, it computes the angle by which the sweeping line must rotate (in the anticlockwise direction) in order to hit p. It then picks as candidate critical records the two with the minimum and maximum angles. The minimum-angle record corresponds to the anticlockwise interim facet, and the maximum-angle record to the clockwise interim facet.

We demonstrate the first step of FP in Figure 4.9(a), where T includes records $p_3, p_4, ..., p_{10}$. We first remove p_9 and p_{10} from T because they are dominated by $p_k \equiv p_2$. The angle for each of the remaining records in T is illustrated by a curved two-headed arrow. The minimum-angle record is p_3, while p_7 is the maximum-angle one. These records define the interim facets $\overline{p_3, p_2}$ and $\overline{p_2, p_7}$. Note that if the area above the dominance region of p_2 was empty (i.e., if p_3, p_4, p_5 were not there), the anticlockwise interim facet would be the line segment between p_2 and its projection on the vertical axis. Similarly, if p_6, p_7, p_8 were not there, the clockwise interim facet would connect p_2 with its projection on the horizontal axis.

In the second step of FP we consider non-result records that were not encountered during top-k computation (i.e., records not fetched from the disk as yet), utilizing the R-tree on D and the retained search heap of BRS. We iteratively pop the heap, using its *maxscore* key as is. If the popped entry corresponds to an R-tree node, and the minimum bounding box (MBB) of the node lies completely below the interim facets, we prune/ignore it. Alternatively (i.e., if at least a part of the MBB is above an interim facet), we fetch the node from the disk. If it contains index entries (i.e., it is an internal node of the R-tree), we push its children into the heap, with key equal to their *maxscore* according to q. On the other hand, if it contains data records (i.e., it is a leaf of the R-tree), we consider each of these records p as follows. If p lies above an interim facet, we update the facet to connect p_k with p; otherwise, we ignore p. The process terminates when the heap becomes empty, reporting the interim facets as the final ones. Our implementation uses the beneath-and-beyond technique in [7] to check whether an MBB or record lies below the interim facets.

In Figure 4.9(b) we demonstrate the second step of FP, continuing the example of Figure 4.9(a). The interim facets $\overline{p_3, p_2}$ and $\overline{p_2, p_7}$ from the first step are shown as solid lines. Records p_1, p_2, p_3, and p_7 that are already known to the algorithm, are represented as solid points. Records shown as hollow points are on the disk and have not been encountered yet. In the beginning of the process, the search heap of BRS is assumed to include index entries that correspond to R-tree nodes N_5, N_4, N_1 (stated here in decreasing order of *maxscore*). The first entry popped from the heap corresponds to N_5. A part of its MBB lies above the clockwise facet $\overline{p_2, p_7}$. Hence, node N_5 is fetched from the disk. It includes two records, p_{13} and p_{14}, none of which lies above $\overline{p_2, p_7}$. Therefore, the clockwise facet remains as is. The second popped entry corresponds to N_4, which lies completely below both interim facets and is thus ignored.

The next popped entry corresponds to N_1. The node is read from the disk, since part of its MBB is above the anticlockwise interim facet $\overline{p_3, p_2}$. The two child entries in N_1 point at N_2 and N_3, which are pushed into the heap (with key equal to their *maxscore* according to q). The heap is popped again, producing the entry of N_2. Node N_2 is fetched from the disk, because it is not completely below facet $\overline{p_3, p_2}$. Between records p_{11}, p_{12} contained in the node, p_{11} lies above the anticlockwise facet. The facet is therefore updated to $\overline{p_{11}, p_2}$, shown as a dashed line in the figure. The last entry popped from the heap corresponds to N_3, which is below both the current facets and hence ignored. The heap is now empty and the final facets derived are $\overline{p_{11}, p_2}$ and $\overline{p_2, p_7}$. The correctness of the second step relies on the fact that (i) any pruned index entry or data record lies below both interim facets and therefore is unable to update either of them, and (ii) the process terminates only when the search heap becomes empty.

The facets derived by FP indicate the critical records, e.g., in Figure 4.9 the critical records are p_{11} and p_7. From these records, we derive half-planes $(p_2 - p_{11}) \cdot q' \geq 0$ and $(p_2 - p_7) \cdot q' \geq 0$, respectively, that embody the second set of conditions in Definition 3. The GIR from Phase 1 is intersected with those two half-planes to produce the final GIR. We summarize FP in Algorithm 4.1. Lines 1-2 implement the first step of FP to compute the interim anticlockwise and clockwise facets f_a and f_c, respectively. Lines 3-16 correspond to the second step of FP, while Lines 17-19 intersect the interim GIR from Phase 1 with the half-planes derived from critical records p_a, p_c to produce the final GIR.

4.6.3 Facet Pruning in Higher Dimensions

In more than two dimensions, a top-k query q can be seen as a sweeping hyperplane (to which the query vector is perpendicular). Following the FP paradigm, if we pin the sweeping hyperplane at p_k and allow it to rotate freely in any direction, the critical records are those (among the non-result

Algorithm 4.1: Facet Pruning

```
   Input: heap $H$ of BRS, query vector $q$, $k$-th result $p_k$
   Output: Final GIR
1  $T \leftarrow$ set of non-result records encountered by BRS;
2  $f_a, f_c \leftarrow InterimFacets(T, q, p_k)$;
3  while $H$ is not empty do
4      Pop the top entry $e$ from $H$;
5      if $e$ is below both facets $f_a, f_c$ then
6          Continue;
7      if $e$ corresponds to a leaf node then
8          Fetch the node from the disk;
9          for each record $p$ in the node do
10             if $p$ is above $f_a$ then
11                 Update $f_a$ to $\overline{p, p_k}$;
12             if $p$ is above $f_c$ then
13                 Update $f_c$ to $\overline{p_k, p}$;
14     if $e$ corresponds to an internal node then
15         Fetch the node from the disk;
16         Push all the child entries of the node into $H$;
17 $p_a, p_c \leftarrow$ records incident to $f_a$ and $f_c$, respectively;
18 Intersect GIR from Phase 1 with half-planes $(p_k - p_a) \cdot q' \geq 0$ and
   $(p_k - p_c) \cdot q' \geq 0$;
19 Return GIR;
```

records) that bound its movement, so as to keep all non-result records under the hyperplane. To visualize, we use a 3-dimensional example in Figure 4.10. The figure illustrates the convex hull $\mathcal{CH}'$ of set $\{p_k\} \cup D \backslash R$ and the sweeping plane that is pinned at p_k. The goal in FP is to compute the hull facets that are incident to p_k so as to collect the critical records. In our example, there are five such facets (shown shaded) that lead to five incident/critical records, excluding p_k.

Before we present algorithmic details, we note that a facet is a $(d-1)$-dimensional object that is generally defined by d records. A *ridge* is a $(d-2)$-dimensional object representing the intersection of two neighboring facets. For $d = 3$, the ridge is a line segment where two facets meet, and it is defined by the two records at its endpoints. In Figure 4.10 we point out a ridge at the intersection of a shaded and a normal facet.

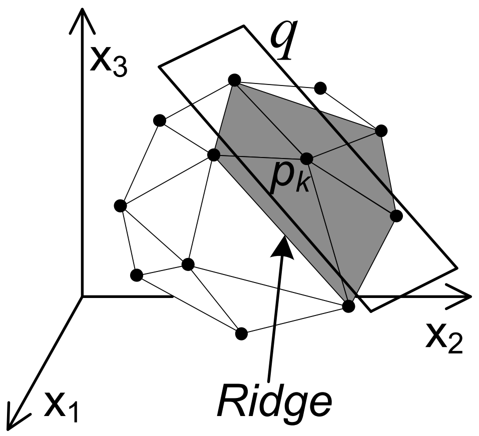
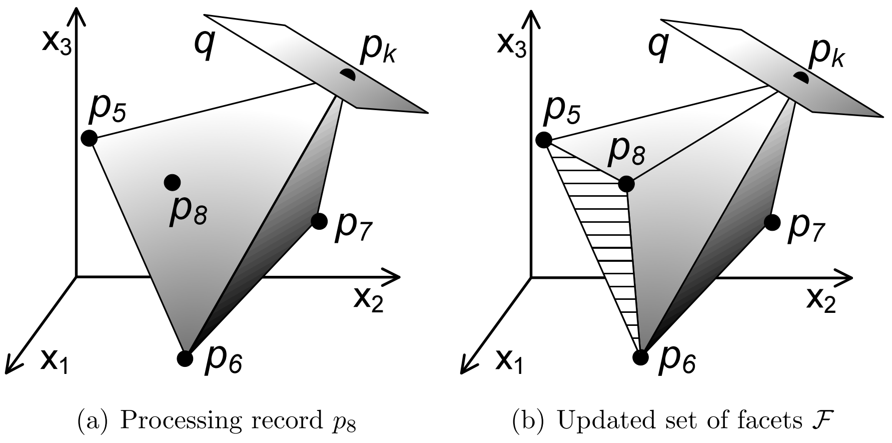


Figure 4.10: Facets incident to p_k ($d = 3$, data space)



(a) Processing record p_8 (b) Updated set of facets $\mathcal{F}$

Figure 4.11: FP example ($d = 3$, data space)

The main idea in FP is to avoid computing the entire hull by maintaining only the facets that are incident to p_k. This is carried out in two steps, at the heart of which lies a strategy that incrementally updates the set of incident facets as new records are considered. The first considers the set T of non-result records encountered by BRS, and the second those still on the disk.

First Step of FP

The first step starts by discarding/removing from T those records that are dominated by p_k. Then, it draws d records[6] from T and computes a convex hull on these records and p_k (note that the cost to compute the convex

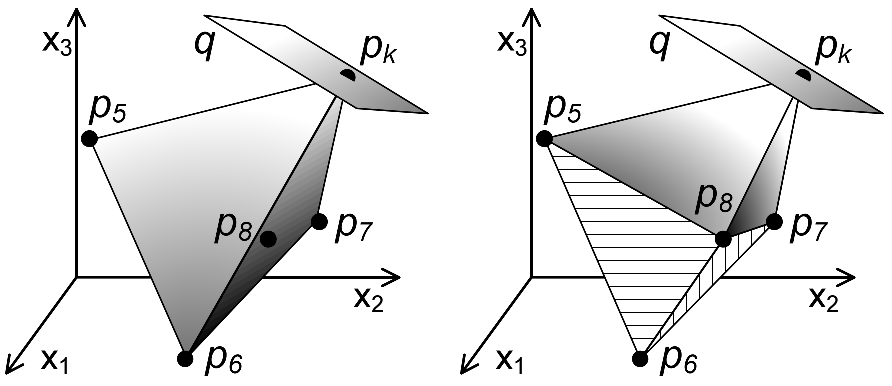
[6]If T contains fewer than d records, we may use instead the projections of p_k on each of the d axes. This is equivalent to our strategy in Section 4.6.2 when the first step of FP encountered empty areas around the dominance region of p_k.

hull of $d+1$ records is trivial). The hull facets that are incident to p_k are maintained in set $\mathcal{F}$, while the rest are discarded. Consider the 3-dimensional example in Figure 4.11(a) where p_5, p_6, p_7 are drawn from T. From the convex hull of p_5, p_6, p_7, p_k, we keep in $\mathcal{F}$ the three facets incident to p_k, i.e., $(p_k, p_5, p_6), (p_k, p_6, p_7)$, and (p_k, p_7, p_5). The bottom facet (p_5, p_6, p_7) is not incident to p_k and is discarded.

Next, we process each record p in T and incrementally update $\mathcal{F}$. If p lies below all the facets in $\mathcal{F}$, it is discarded. Otherwise, we update $\mathcal{F}$ following a process reminiscent of Clarkson's algorithm, yet focused on facets incident to p_k only.

Specifically, we initialize a set $\mathcal{F}_v$ and place in it all the facets from $\mathcal{F}$ that p lies above of. In Figure 4.11(a) the only facet that is below p_8 is (p_k, p_5, p_6). Then, we collect all the ridges that are shared between a facet in $\mathcal{F}_v$ and a facet in $\mathcal{F} \backslash \mathcal{F}_v$. In the literature these are called *horizon ridges*. In our example the horizon ridges are $\overline{p_5, p_k}, \overline{p_6, p_k}$ and $\overline{p_5, p_6}$. Among them, we keep only those incident to p_k (i.e., the former two). We update $\mathcal{F}$ by (i) removing the facets that belong to $\mathcal{F}_v$, and (ii) inserting one new facet for each of the retained horizon ridges (the new facets are formed by connecting p with the respective ridge). Figure 4.11(b) shows the updated $\mathcal{F}$ after processing p_8. Observe that facet (p_k, p_5, p_6) is removed and is replaced by two new facets. The first is defined by p_8 and horizon ridge $\overline{p_5, p_k}$, and the second by p_8 and $\overline{p_6, p_k}$. Note that the striped facet was never created nor inserted in the updated $\mathcal{F}$. That facet would not be incident to p_k, and we avoided its unnecessary formation through our strategy to discard those horizon ridges that were not incident to p_k (i.e., ridge $\overline{p_5, p_6}$).

In Figure 4.12(a) we apply the above methodology to an alternative scenario where p_8 lies above two facets instead of just one, i.e., in this case $\mathcal{F}_v$ includes (p_k, p_5, p_6) and (p_k, p_6, p_7). We stress that $\overline{p_6, p_k}$ is *not* a horizon ridge in this case, because it is formed by facets that are both in $\mathcal{F}_v$. The horizon ridges



(a) Processing record p_8 (b) Updated set of facets $\mathcal{F}$

Figure 4.12: Second FP example ($d = 3$, data space)

are $\overline{p_5, p_6}, \overline{p_6, p_7}, \overline{p_5, p_k}$ and $\overline{p_7, p_k}$. The former two are discarded because they are not incident to p_k. The latter two are used in tandem with p_8 to create two new facets, (p_k, p_8, p_5) and (p_k, p_7, p_8), as shown in Figure 4.12(b). These new facets are inserted into $\mathcal{F}$, while those in $\mathcal{F}_v$ are removed from $\mathcal{F}$. Striped facets are shown for completeness, but were never formed nor placed into $\mathcal{F}$.

We optimize the first step with a heuristic. Recall that in the beginning of this step, we draw d records from T to (build a convex hull and) form the initial set $\mathcal{F}$. Instead of a random choice, we pick the d records from T with the maximum values along each of the d dimensions. The rationale is that many non-result points are likely to lie below the formed facets, and thus be pruned directly later on.

Second Step of FP

In the second step we refine $\mathcal{F}$ by considering non-result records that have not been encountered (not fetched into memory) before. $\mathcal{F}$ is updated gradually as we pop entries from the search heap of BRS. The exploration of the R-tree is similar to the 2-dimensional case. Index nodes are pruned (ignored) if they lie completely below each facet in $\mathcal{F}$; otherwise, they are read from disk. When an internal node is read, its child entries are pushed into the heap with key set to their *maxscore* according to q. When a leaf node is read, each of its records is checked against $\mathcal{F}$, the latter being updated if the record lies above some of its facets. The update process is identical to the description in Section 4.6.3. The process terminates when the heap becomes empty, and the critical records are collected from the final set $\mathcal{F}$. Every critical record p is mapped into a half-space of the form $(p_k - p) \cdot q' \geq 0$ and intersected with the interim GIR from Phase 1 in order to produce the final GIR[7].

Although the visual examples used in Section 4.6.3 are for $d = 3$, both steps of the FP methodology apply to higher dimensions without modification, by simply using the conventional notions of facets and ridges in the respective space.

We conclude the discussion about FP with a note on its complexity. According to [21], the number of facets on $\mathcal{CH}'$ is $O(n^{d/2})$, while the number of facets incident to a record on the hull (e.g., to p_k) is $O(n^{\frac{d}{2}-1})$. Computing these facets is the bottleneck in FP. FP uses a process based on Clarkson's algorithm, whose complexity for the entire hull is $O(n^{d/2})$. Following a similar reduction to [21], the cost to compute only the facets incident to p_k is $O(n^{\frac{d}{2}-1})$.

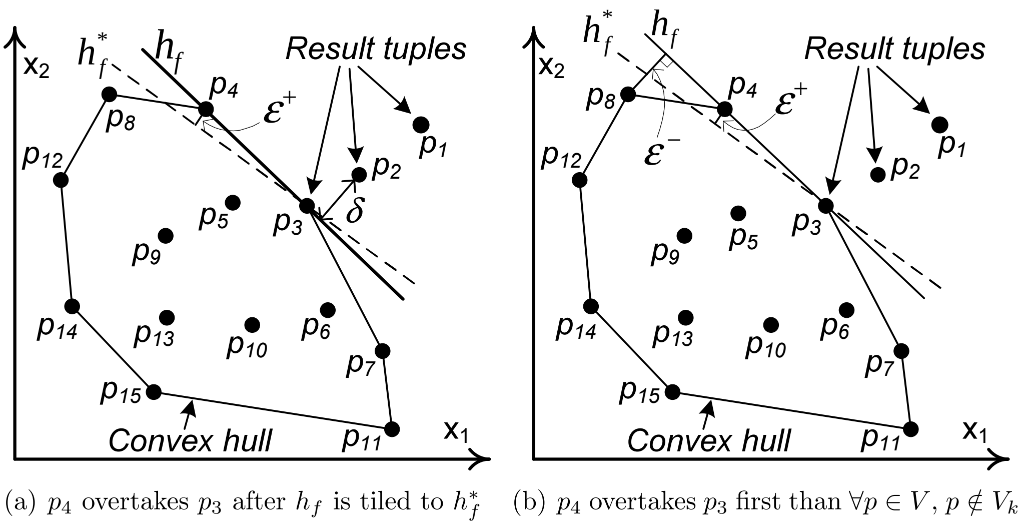
4.6.4 Correctness Proof of FP

Although the rationale behind FP looks intuitive, the correctness guarantee of FP is non-trivial. In the following, we give a formal proof of the correctness of FP for GIR computation.

Lemma 4 *Given a dataset D, a top-k query q with result $R = \{p_1, p_2, ..., p_k\}$, and the convex hull $\mathcal{CH}'$ of set $\{p_k\} \cup D \backslash R$. Suppose V and V_k are the vertices of $\mathcal{CH}'$ and the vertices incident to p_k respectively. To ensure that no point in $D \backslash R$ can overtake p_k, it suffices to consider V_k only for GIR computation.*

Proof 4 *According to Definition 3, GIR is determined by two sets of conditions, where the first set requires that p_{i+1} cannot overtake p_i in the top-k*

[7]An optimization is possible where the interim GIR from Phase 1 is mapped into facets and can be incorporated into the first step of Phase 2 to further "tighten" the criteria for fetching nodes from the disk in the second step of Phase 2.



(a) p_4 overtakes p_3 after h_f is tiled to h_f^* (b) p_4 overtakes p_3 first than $\forall p \in V, p \notin V_k$

Figure 4.13: Illustration of Lemma 4 ($d = 2$, $k = 3$, data space)

result for $i \in [1, k)$, whereas the second one ensures that any non-result point $p \in D \backslash R$ cannot overtake p_k. Here we consider non-result points for the second set of conditions.

Consider the convex hull $\mathcal{CH}'$ of set $\{p_k\} \cup D \backslash R$. Since the non-result points inside $\mathcal{CH}'$ cannot overtake p_k under any monotone scoring function, we discard them and consider the vertices of $\mathcal{CH}'$ only. Let V and V_k be the set of vertices of $\mathcal{CH}'$ and the set of vertices that are incident to p_k, respectively. Note that $p_k \in V$ and $V_k \subset V$. We prove that (1) every $p \in V_k$ may overtake p_k, and (2) by forcing every $p \in V_k$ not to overtake p_k, no point in $V \backslash V_k$ can overtake p_k.

Given a point $p \in V_k$, there exists a facet f of $\mathcal{CH}'$ that has p and p_k lie on it. Suppose the supporting hyperplane of f is h_f, then for any query hyperplane q that overlaps h_f, the scores of p and p_k are the same. The Euclidean distance from p to h_f is 0. Let the shortest distance from some point in $R \backslash \{p_k\}$ to h_f be $\delta > 0$. We anchor h_f at p_k and tile h_f a little bit below p, and stop immediately when p is no long on h_f (see a 2-dimensional example in Figure 4.13(a)). Assuming the tiled hyperplane is h_f^* and the distance from p to h_f^* is $\epsilon^+ > 0$, then it is possible that $\epsilon^+ \leq \delta$, e.g., $\epsilon^+ = \frac{\delta}{2}$. This means that after tilting, p and every point in $R \backslash \{p_k\}$ may be swept first by h_f^* before p_k. Thus, *by regarding h_f^* as a query hyperplane, p may overtakes p_k as the k-th result, i.e., the new top-k result will change to $R^* = R\backslash\{p_k\} \cup \{p\}$. Since p is chosen arbitrarily from V_k, every point in V_k has the potential to overtake p_k. Hence, while computing GIR we need to monitor each of the points in V_k, so as to ensure that none of them can overtake p_k.*

Next we prove that by forcing every $p \in V_k$ not to overtake p_k, no point in $V\backslash V_k$ can overtake p_k. Consider a points $p \in V_k$. Suppose f is the facet of $\mathcal{CH}'$ that has p and p_k lie on it, and h_f is the supporting hyperplane of f. According to definition of convex hull, all points of $\mathcal{CH}'$, other than p and p_k, lie below h_f. Let p' be the point in $V\backslash V_k$ that is nearest to h_f and ϵ^- be the distance from p' to h_f, then we have $\epsilon^- < 0$. Similarly, we anchor h_f at p_k and tilt h_f towards p'. We denote the tiled hyperplane by h_f^*, and let ϵ^+ be the distance from p to h_f^*. We stop tilting immediately when p lies above h_f^*, i.e., when ϵ^+ satisfies $0 < \epsilon^+ < |\epsilon^-|$. Then we must have (1) p' still lies below h_f^*, (2) p lies above h_f^*, and (3) p_k is on h_f^*. Given arbitrarily small positive value of ϵ^+ and by treating h_f^* as query hyperplane, among the three points p, p' and p_k, h_f^* will always encounter p first when sweeping through the data space. This means that if we monitor every point $p \in V_k$ such that the query hyperplane does not hit p before p_k, then every point $p' \in V\backslash V_k$ cannot be swept by the query hyperplane before p_k (see Figure 4.13(b)).

Based on the above discussion, we have proven that (1) every $p \in V_k$ may overtake p_k, and (2) by ensuring that no point in V_k can overtake p_k, every point $p' \in V\backslash V_k$ cannot overtake p_k. Therefore, when identifying critical records for GIR computation it is enough for FP to consider only the set V_k, i.e., the vertices of the convex hull of $\{p_k\} \cup D\backslash R$ that are incident to p_k.

4.7 Extensions and Visualization

In this section we extend our methodology to order-insensitive GIR, discuss the handling of non-linear scoring functions, and describe possible GIR visualization techniques.

4.7.1 Order-Insensitive GIR

A variant of the GIR problem arises when the user or application is concerned only about the composition of the top-k result (but not the order of records in it). The *order-insensitive* GIR is the maximal locus in query space where the composition of R is preserved. We denote it as GIR*.

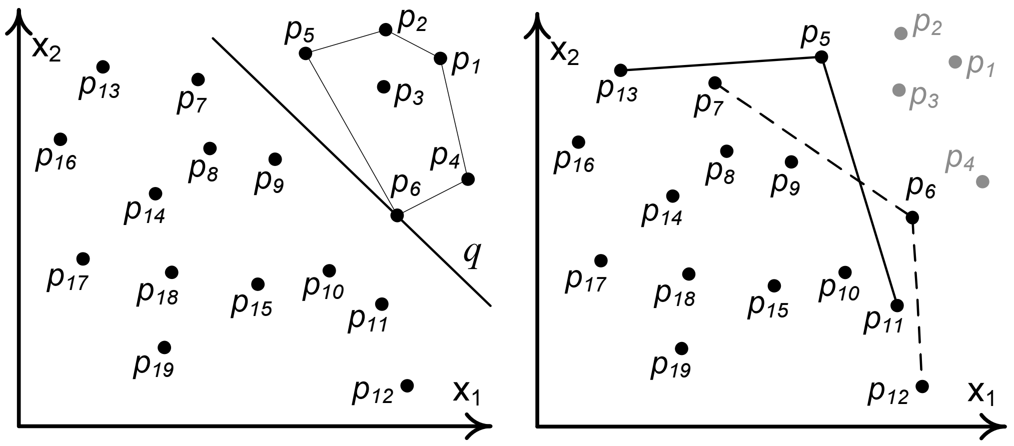
Definition 4 ***Order-insensitive GIR (GIR*).*** *Given a dataset D and a top-k query q with result $R = \{p_1, p_2, ..., p_k\}$, the GIR* is the locus of all vectors q' in query space where*

$$\mathcal{S}(p_i, q') \geq \mathcal{S}(p, q')$$

for each $i \in [1, k]$ and every record $p \in D \backslash R$.

The GIR* is defined by looser conditions than the (order-sensitive) GIR, and hence it fully encloses the latter. Definition 4 suggests a straightforward processing approach. Consider a result record $p_i \in R$. Let GIR$_i$ be the GIR derived if we (skip Phase 1 and) apply Phase 2 by having p_i play the role of p_k, using any of the methods in Sections 4.5 or 4.6. GIR$_i$ is the maximal region in query space where $\mathcal{S}(p_i, q') \geq \mathcal{S}(p, q')$ for every record $p \in D \backslash R$. By Definition 4, the GIR* is the intersection of the GIR$_i$ regions for each $i \in [1, k]$, i.e., GIR$^* = \bigcap_{i=1}^{i=k}$ GIR$_i$.

To optimize this process, we observe that not every record in R could affect the GIR*. In Figure 4.14(a), where $k = 6$ and $R = \{p_1, ..., p_6\}$, we consider the convex hull of R. Any record p_j inside the hull (e.g., p_3) can be ignored. The rationale is similar to CP in Section 4.5.2. For every query vector q' there is at



(a) Result pruning (b) Facet sets $\mathcal{F}_5$ and $\mathcal{F}_6$

Figure 4.14: GIR* computation ($k = 6, d = 2$, data space)

least one result record p_i that lies on the hull, such that $\mathcal{S}(p_i, q') \leq \mathcal{S}(p_j, q')$. Thus, for any non-result record p to overtake p_j in score, p would first have to overtake a result record that lies on the hull.

Further result pruning is possible. Observe that p_2 dominates p_5, i.e., for every query vector $q' \in [0, 1]^d$, p_5 scores lower than p_2. Hence, any non-result record would have to overtake p_5 before it can reach p_2 in score. That is, we can safely disregard all result records that dominate at least another record in R. This strategy prunes p_2, p_1, p_4 (the first dominates p_5 and the other two p_6).

In summary, we disregard result records that (i) lie inside the convex hull of R or (ii) dominate at least one other record in R. We denote by R^- the remaining result records. In our example, $R^- = \{p_5, p_6\}$. Subsequent processing follows SP, CP or FP.

SP and CP produce the GIR$_i$ region for each $p_i \in R^-$ in the same way as in Section 4.5, and report $\bigcap_{i=1}^{i=|R^-|} \text{GIR}_i$ as GIR*. Note that $\mathcal{SL}$ and $\mathcal{SL} \cap \mathcal{CH}$ over the non-result records (in SP and CP, respectively) are computed once and used for all GIR$_i$ derivations.

FP, in its first step, considers the set T of non-result records encountered during top-k computation. For every $p_i \in R^-$, it computes the set $\mathcal{F}_i$ of facets that are incident to p_i (on the convex hull of set $\{p_i\} \cup T$). This is done in the same fashion as in Sections 4.6.2 or 4.6.3, depending on dimensionality.

In the second step, FP maintains the $\mathcal{F}_i$ sets concurrently as it fetches new non-result records from the disk. The process is similar to Section 4.6. The main difference is that index nodes (that are popped from the search heap) are only pruned if they lie below all the facets in every $\mathcal{F}_i$ set. Also, each record p fetched from disk is checked against all the $\mathcal{F}_i$ sets and used to update those that include at least one facet below p. When the search heap becomes empty, each $\mathcal{F}_i$ set is used to derive the respective GIR$_i$ region. Finally, FP reports GIR$^* = \bigcap_{i=1}^{i=|R^-|}$ GIR$_i$.

Continuing our example, Figure 4.14(b) illustrates facet sets $\mathcal{F}_5$ and $\mathcal{F}_6$. Each facet in $\mathcal{F}_5$ determines a half-space of the form $(p_5 - p) \cdot q' \geq 0$. The intersection of these half-spaces is GIR$_5$. Region GIR$_6$ is derived similarly, and FP reports GIR* = GIR$_5 \cap$ GIR$_6$.

4.7.2 Non-Linear Scoring Functions

Our main focus in this chapter is on linear scoring functions. While CP and FP (rely on convex hull properties that) may not extend to more general function types, SP can handle a broader class of functions. The following discussion considers the original (order-sensitive) GIR, but it translates easily to the GIR* context too.

There are two components in SP, namely, (i) pruning non-result records, and (ii) using the remaining non-result records to form the GIR. SP pruning applies to any monotone[8] scoring function. In other words, the only non-result records that could overtake the k-th record in R, under any such function, are guaranteed to belong to the skyline of $D \backslash R$ [40].

Identifying the non-result records that could affect the GIR helps to limit

[8]We follow the convention that the larger a record's attributes the higher its score. A scoring function $\mathcal{S}(p, q)$ is monotone iff for any dimension $i \in [1, d]$ and for any pair of records p, p' with $p.x_i \geq p'.x_i$ and $p.x_j = p'.x_j$ $\forall j \neq i$, it holds that $\mathcal{S}(p, q) \geq \mathcal{S}(p', q)$.

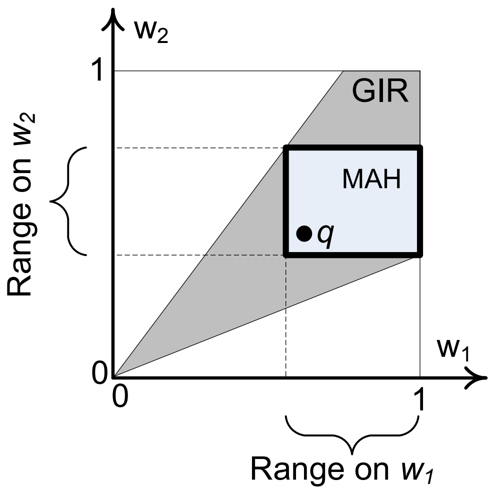
the number of conditions in Definition 3. Forming the GIR, however, requires translating these conditions to a locus in query space. If the scoring function is of the form $\mathcal{S}(p,q) = \sum_{i=1}^{d} w_i g_i(p)$, where each $g_i(p)$ is a function over the attributes of p, the GIR is derived using half-space intersection as per normal. To see this, the GIR is defined by conditions of the form $\mathcal{S}(p,q') \geq \mathcal{S}(p',q')$ for various pairs of records p, p'. Condition $\mathcal{S}(p,q') \geq \mathcal{S}(p',q')$ can be rewritten as $\sum_{i=1}^{d} w_i g_i(p) \geq \sum_{i=1}^{d} w_i g_i(p') \Rightarrow \sum_{i=1}^{d} w_i(g_i(p) - g_i(p')) \geq 0$. Since we are comparing specific records p and p', the factors $(g_i(p) - g_i(p'))$ are constants, and thus the condition corresponds to a half-space in query space. That is, Phase 1 and Phase 2 may employ plain half-space intersection to produce the GIR.

For scoring functions that do not belong to the above category, conditions of the form $\mathcal{S}(p,q') \geq \mathcal{S}(p',q')$ no longer correspond to half-spaces. This implies that the GIR is no longer a convex polytope but a general convex set. Exact representation of the GIR in such cases is computationally expensive or not possible at all, which would call for approximate GIR representation techniques, such as polytope approximation, Monte Carlo simulation, etc [70].

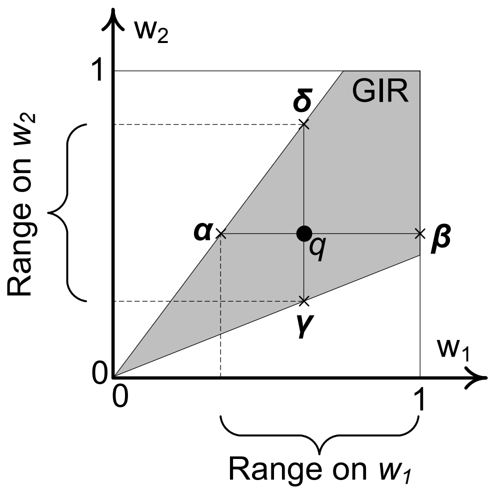
4.7.3 GIR Visualization

One of the GIR applications is to give the user a sense of the weight shift required to induce a change in the top-k result. Being a d-dimensional polytope, GIR is challenging to visualize for $d > 2$. We describe two possible visualization options.

Assuming that the GIR is already derived using any of our methods, the first visualization technique computes the maximum-volume axis-parallel hyper-rectangle (MAH) that (i) contains the query vector q and (ii) lies completely inside the GIR. This is an instance of the *bichromatic rectangle problem*, for which several algorithms are available [6, 31]. Figure 4.15(a) shows the MAH in a 2-dimensional query space. The MAH can be visualized easily



(a) MAH



(b) Interactive projection

Figure 4.15: GIR visualization ($d = 2$, query space)

by projecting its sides on the different axes, producing for example bounds like those in Figure 4.1. The advantage of this approach is that the bounds are fixed as long as the query vector remains inside the MAH. The disadvantage is that the MAH is a subset of the GIR.

An alternative is the interactive projection approach, which does not sacrifice maximality (i.e., it allows exploration of the full extent of the GIR) but requires on-the-fly readjustment of the bounds plotted on the interface. Consider the query vector in Figure 4.15(b). We first find the horizontal projections of q on the GIR, i.e., points α and β, and map them on the w_1 axis to derive an upper and lower bound for w_1 like those in Figure 4.1. Similarly, we project q vertically on the GIR (getting points γ and δ) and produce the bounds for w_2. Note that the derived ranges are equivalent to the LIRs in [53]. Should the user shift the query vector (by varying one or multiple weights), we may interactively re-project the new location of q on the GIR, and redraw on-the-fly the new permissible ranges for each factor. That is, as the user shifts q (within the GIR), she sees the bounds for each factor being adjusted in real time.

4.8 Experiments

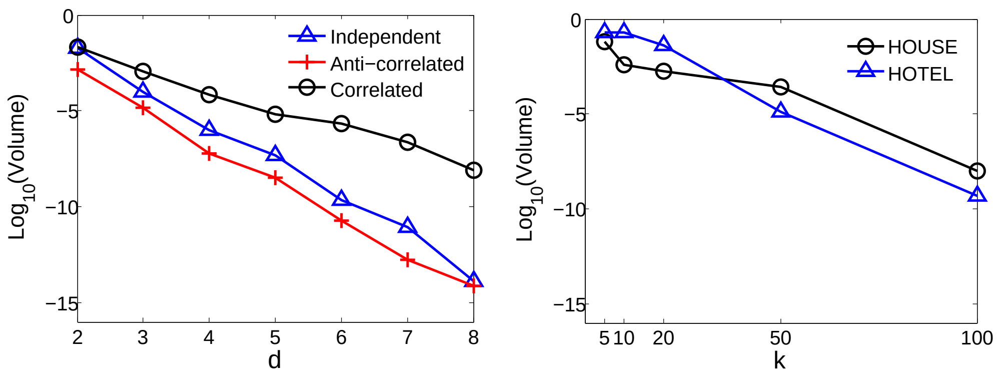
In this section we evaluate the efficiency of the SP, CP, and FP algorithms, using synthetic and real datasets. The synthetic datasets include *Independent* (IND), *Correlated* (COR), and *Anti-correlated* (ANTI), which are standard benchmarks for preference-based queries [16]. IND is uniformly and independently distributed. In COR, records that have a large value in one dimension tend to have large values in the other dimensions too. In ANTI, a record with a large value in one dimension tends to have small values in the rest. We also use real datasets HOUSE and HOTEL (from *ipums.org* and *hotelsbase.org*, respectively). HOUSE contains 315,265 records, each with six attributes representing an American family's expenditure in gas, electricity, water, heating, insurance, and property tax. HOTEL contains 418,843 hotel records with four attributes, namely stars, price, number of rooms, and number of facilities. All attributes are normalized to $[0, 1]$. The datasets are indexed by an R*-tree using 4KByte disk pages.

In the default setting, we place data and indices on the disk and evaluate performance in terms of CPU and I/O time[9]. However, the CPU charts in isolation indirectly also evaluate the scenario where data and indices are kept in memory. Unless otherwise specified, we compute the order-sensitive GIR. We present total CPU and I/O times, accounting for Phases 1 and 2. A buffer for disk pages cannot improve I/O time, since none of the methods fetches the same index or data page twice. Thus, we do not use one. All methods are implemented in C++ and use the Qhull library (*qhull.org*) for half-space intersection. Experiments are run on a PC with Intel Core2Duo 3GHz CPU. Table 4.2 summarizes the investigated parameters, along with their tested values and defaults (in bold). With the real datasets we control only the last parameter (i.e., k). Each reported measurement is the average over 100 random

[9]Note that SP and CP have identical I/O cost, because they access the disk through the same BBS process.

Parameter	**Range of values**
Dimensionality, d	2, 3, **4**, 5, 6, 7, 8
Dataset cardinality, n	0.5M, **1M**, 5M, 10M, 20M
Top-k result size, k	5, 10, **20**, 50, 100

Table 4.2: Experiment Parameters



(a) Varying d (Synthetic) (b) Varying k (Real data)

Figure 4.16: Ratio of GIR volume to query space volume

queries.

We first provide insight into the nature of the GIR. In Figure 4.16 we present the ratio of GIR volume to the volume of the query space, which coincides with the sensitivity measure discussed in the Introduction and the LIK probability in [70]. In Figure 4.16(a) we use synthetic data and vary d. The GIR volume drops exponentially with d, and Figure 4.8(b) provides the reason. As d increases, so does the number of facets incident to p_k, which implies that more conditions (half-spaces) bound the GIR. The GIR is the largest in COR and the smallest in ANTI, because COR has the fewest incident facets among our synthetic data, while ANTI has the most (as shown in Figure 4.8(b)). The trends in Figure 4.16(a) also reveal an interesting fact about the nature of the top-k query itself; the alternative top-k results become dramatically less distinguishable as d grows. In Figure 4.16(b) we plot the volume ratio versus k for the real data. A larger k implies more half-spaces induced from the first set of conditions in Definition 3, leading to a smaller GIR.

Our solutions can be applied to general dimensionalities, while in our experiments we focus on dimensionalities up to $d = 8$. This is due to several reasons.

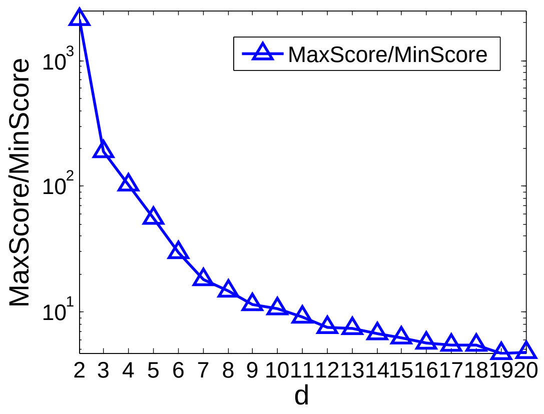


Figure 4.17: Effect of dimensionality d on MaxScore/MinScore ratio

First, performance deterioration (and lack of usefulness) with dimensionality is common to fundamental problems like nearest neighbors (NN) [14] and convex hull computation [7]. Top-k and GIR computation share that characteristic. For NN search, [14] shows that in as few as 10 dimensions, the distance (from a query location) to the nearest data record approaches the distance to the farthest record in the dataset.

This is also the case for top-k queries, i.e., *the score of the top record approaches the score of the lowest-scoring record in the entire dataset.* In Figure 4.17 we plot the ratio of the maximum score (MaxScore) to the minimum score (MinScore) across the entire dataset (using the default 1 million IND data); the y-axis of the figure is in logarithmic scale[10]. The resemblance to the trends in [14] is striking (see Fig. 7 in [14]). The increase in dimensionality tends to make all records equally preferable. Along the same lines, Figure 4.16(a) shows that the GIR volume drops exponentially with dimensionality, meaning that the distinguishability among alternative top-k results drops dramatically with d.

Next, we investigate the usefulness of GIR in providing recommendation support with alternatives. Specifically, GIR provides the user with alternative top-k results when the query vector shifts to any location on the boundary of the GIR. These results may be "one perturbation away" from her original re-

[10]For the experiment to be meaningful, the query vectors are normalized such that $\sum_{i=1}^{d} w_i = 1$ which ensures that the score of any record is between 0 and 1 in all dimensions.

Dataset	**Average No. of Alternative Results**
IND	8
ANTI	7.6
COR	8.9
HOTEL	8.25
HOUSE	14.41

Table 4.3: Alternative top-k results at the boundary of the GIR

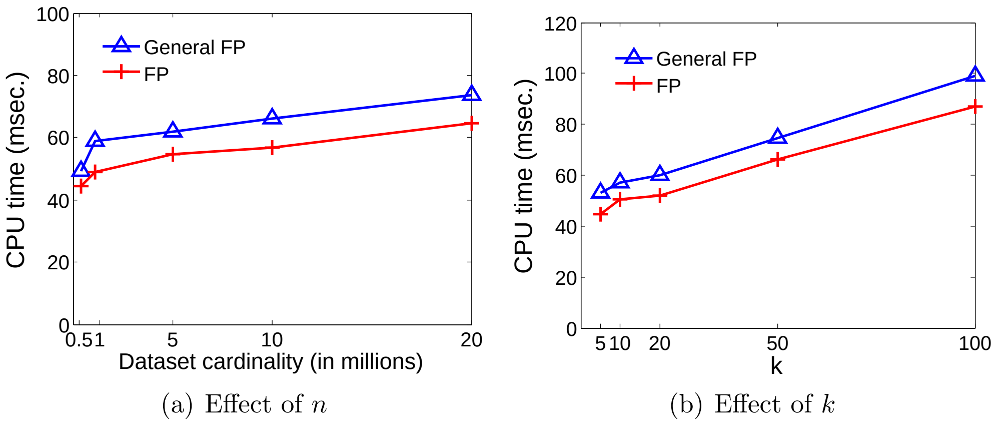
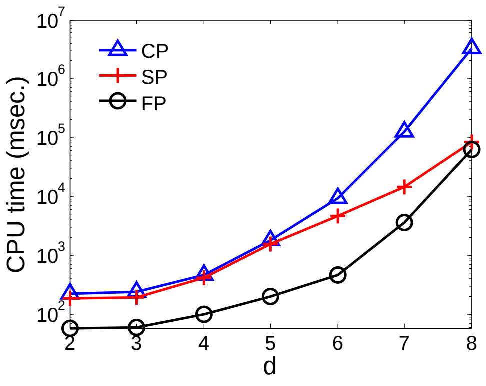


Figure 4.18: Specialized versus general FP for $d = 2$ (IND)

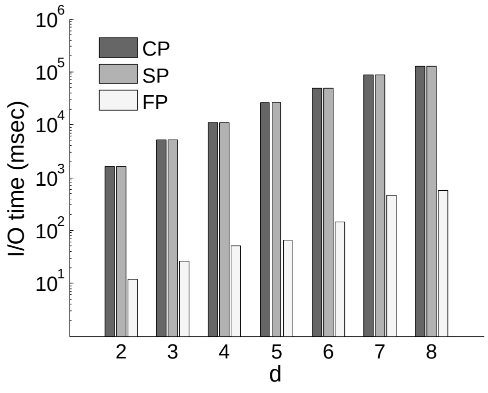
sult, but their number (and information conveyed) is considerable. In Table 4.3 we show how many these alternative top-k results are, for all datasets used in the experiments in the default setting. In addition to these alternatives, the GIR boundary also tells the user what circumstances (weight adjustments) are needed to derive each of them. The alternative results are derived along the way with GIR computation.

In Section 4.6.2 we present a FP algorithm specifically designed for 2-d case, while in Section 4.6.3 we give a general FP algorithm for general dimensionality. in Figure 4.18 below, we compare our specialized 2-d algorithm (as presented in Section 4.6.2, labeled “FP” in the figure) with the general-dimensionality FP applied for the 2-d case (labeled “General FP” in the figure). We vary n and k and measure the CPU time on the IND dataset. Note that the algorithms make identical accesses to the disk (thus I/O charts are omitted). Our specialized algorithm improves CPU time by 11% to 21%.

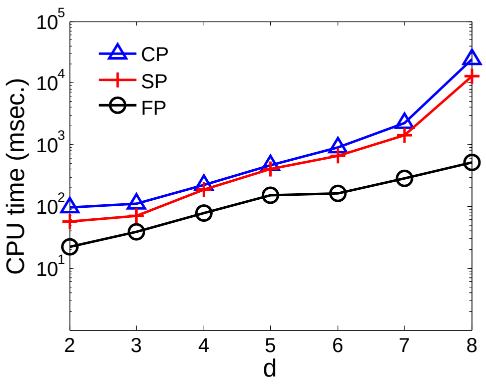
In Figure 4.19 we study the effect of dimensionality d on the performance of SP, CP, and FP, using synthetic data. All the charts for this experiment are



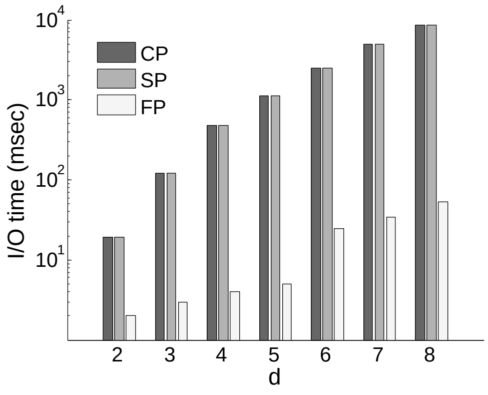
(a) CPU time (IND)



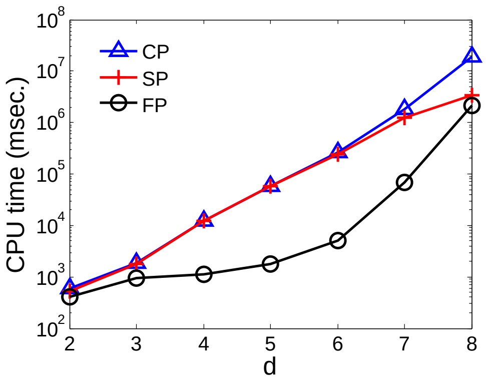
(b) I/O time (IND)



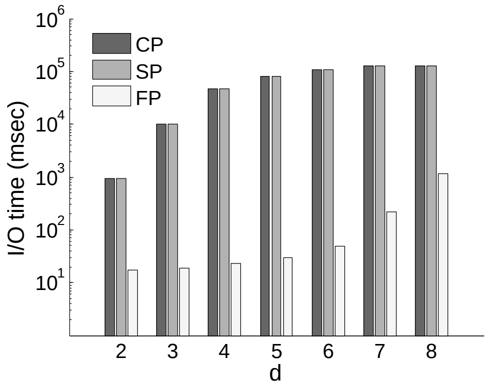
(c) CPU time (COR)



(d) I/O time (COR)



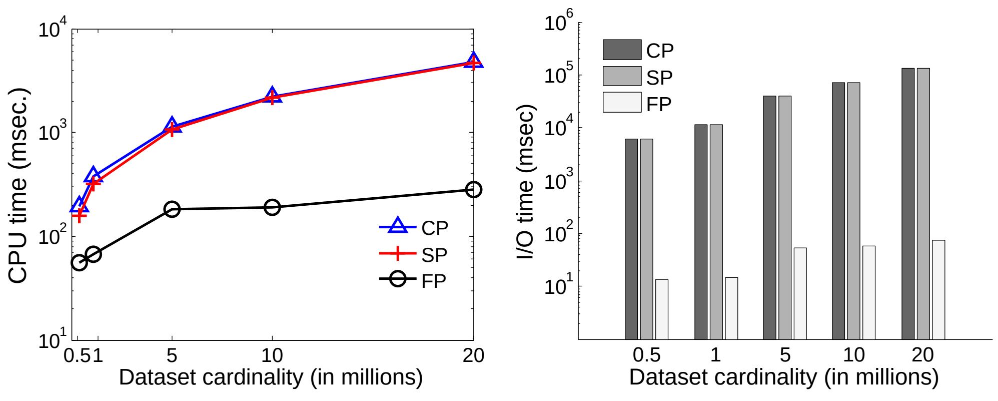
(e) CPU time (ANTI)



(f) I/O time (ANTI)

Figure 4.19: Effect of dimensionality d for synthetic data

in logarithmic scale. FP outperforms SP and CP in all cases, with SP being the runner-up. The largest differences are observed for ANTI, where FP takes 53 to 2700 times shorter I/O time than SP, and 1.3 to 47 times shorter CPU time. The difference is smaller in COR (because there are fewer skyline records than in IND and ANTI), with FP, however, still performing 9.6 to 224 times fewer I/Os and 1.8 to 24 times fewer computations. Interestingly, the CPU



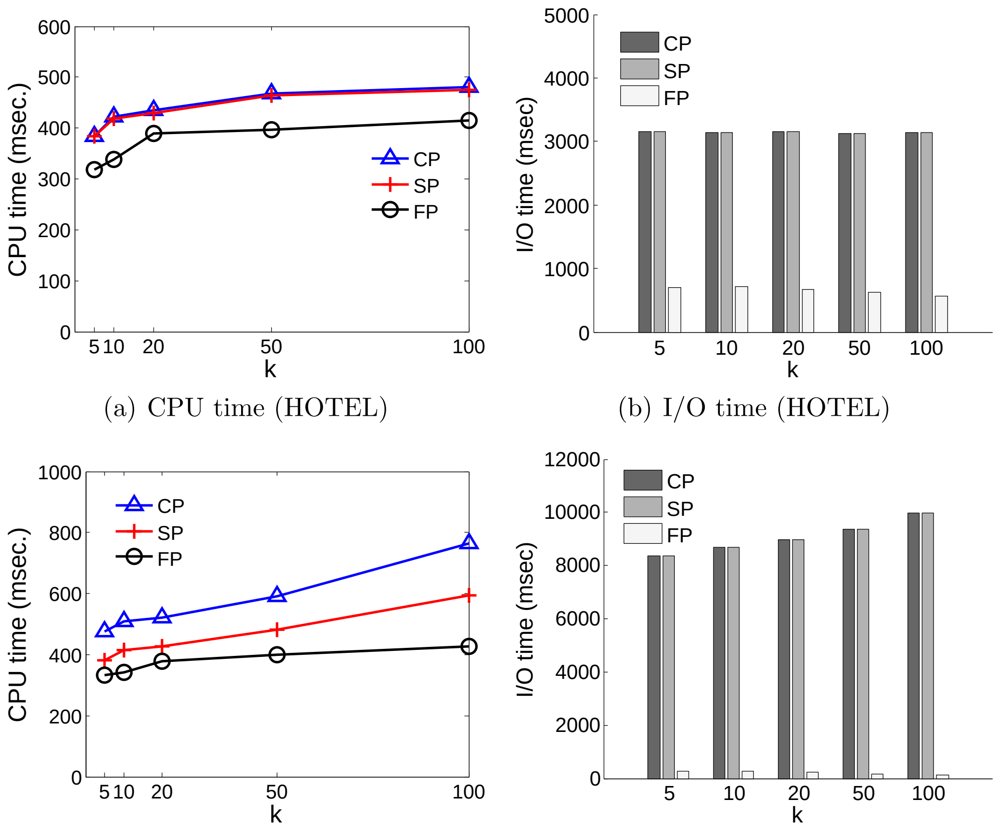
(a) CPU time (b) I/O time

Figure 4.20: Effect of dataset cardinality n (IND)

time of CP is longer than SP. Although CP prunes more records, its expensive convex hull computation outweighs the benefits of pruning (an issue discussed at the end of Section 4.5.3).

In Figure 4.20 we investigate the effect of dataset cardinality n, varying it from 0.5M to 20M tuples. Due to lack of space, we show results for IND only – the trends are similar for COR and ANTI. CPU and I/O times naturally increase with n in all methods. The important finding is that FP scales much better with cardinality, as a result of focusing only on the (relatively few) convex hull facets that are incident to p_k. In terms of I/O cost, it outperforms the runner-up (SP) by 460 to 1748 times, and by 2.8 to 16.5 times in terms of CPU cost.

In Figure 4.21 we assess the effect of k using the real datasets. A larger k implies more records in T, i.e., more non-result records encountered during top-k computation by BRS. The larger T leads to an increase in CPU time. On the other hand, the effect of k on I/O cost involves two conflicting factors. A larger T implies that most critical records (in FP) and skyline records (in SP/CP) have already been fetched from disk (by BRS). This leads to a slight decrease in I/O cost for all methods in HOTEL. In HOUSE, however, due to its higher dimensionality (six instead of four) and different distribution, the inclusion of more records in the top-k result (and thus their exclusion from



(a) CPU time (HOTEL) (b) I/O time (HOTEL)

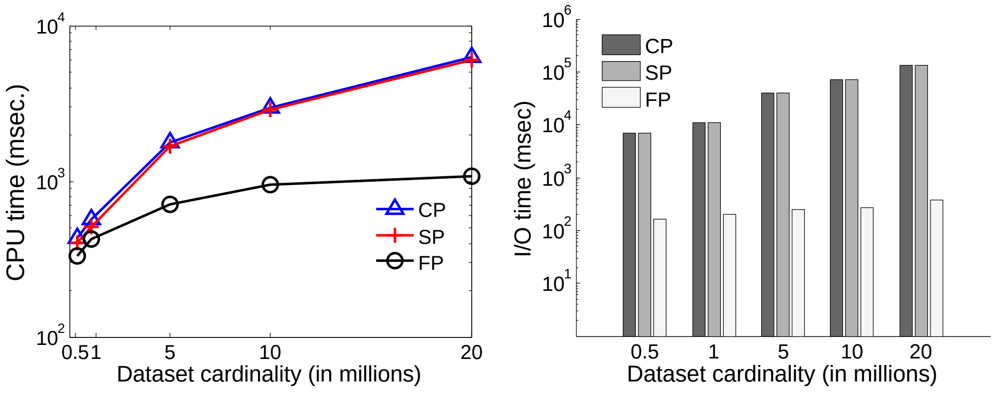
(c) CPU time (HOUSE) (d) I/O time (HOUSE)

Figure 4.21: Effect of k for real data

$D \backslash R$) deprives the skyline computation module (BBS) of records with high dominating/pruning power and "widens" the skyline, thus raising the I/O cost for SP and CP. In contrast, FP is independent of the skyline, and its I/O cost slightly decreases with k in this dataset as well.

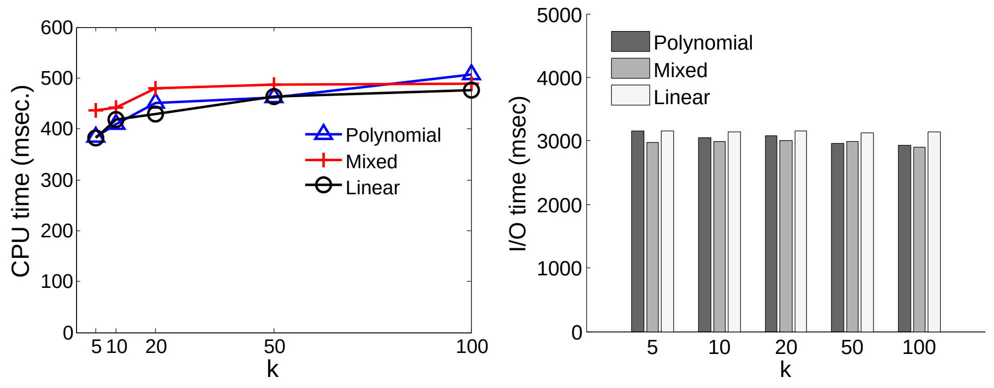
In Figure 4.22 we evaluate our algorithms for the computation of *order-insensitive* GIR. We set all parameters to their defaults and vary n (for IND data). The trends are similar to Figure 4.20, however, the cost of all methods increases. The reason is that, as explained in Section 4.7.1, multiple result records need to be considered against the non-results (as opposed to just considering p_k against them).

Returning to the default, order-sensitive GIR, in Figure 4.23 we consider *non-linear* scoring functions. Using HOTEL and varying k, we investigate the performance of SP for (monotone) functions $\mathcal{S}(p,q) = w_1x_1^4 + w_2x_2^3 + w_3x_3^2 +$



(a) CPU time (b) I/O time

Figure 4.22: Order-insensitive GIR, effect of n (IND)



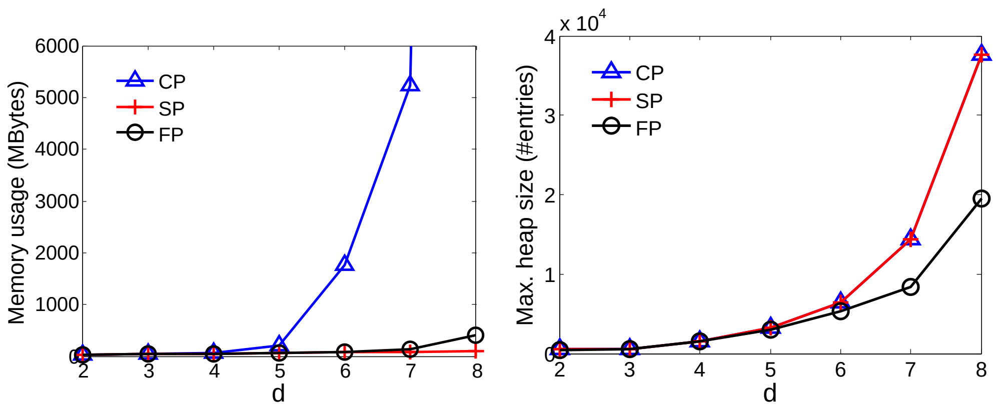
(a) CPU time (b) I/O time

Figure 4.23: Non-linear scoring functions, effect of k (HOTEL)

$w_4x_4^1$ and $\mathcal{S}(p,q) = w_1x_1^2 + w_2e^{x_2} + w_3\log x_3 + w_4\sqrt{x_4}$ (recall that HOTEL is 4-dimensional). We label the functions as "Polynomial" and "Mixed", respectively. "Linear" is included for the sake of comparison.

SP performance is similar for all functions. That is because skyline computation by BBS is independent of the function type (thus the comparable I/O cost), which in turn leads to a similar number of half-spaces to intersect for GIR derivation (thus the comparable CPU time). Results with other monotone functions are similar and omitted in order not to clutter the charts.

In Figure 4.24 we investigate the memory usage of our algorithms. We set all parameters to their default and vary d (for IND data). As d increases, the memory requirement of SP remains the smallest, while the requirement of CP



(a) Memory usage (IND) (b) Max. heap size (IND)

Figure 4.24: Effect of dimensionality d on memory usage

jumps significantly, as shown in Figure 4.24(a). This is due to the excessive number of facets, in the order of $O(n^{d/2})$, maintained by CP. In contrast, FP only stores $O(n^{d/2-1})$ facets, an order of magnitude smaller than that of CP. In terms of heap size, as shown in Figure 4.24(b), FP maintains the smallest search heap compared to SP and CP, because the facets maintained by FP can prune more index/data nodes.

4.9 Summary

In this chapter we study the problem of global immutable region (GIR) computation, which can provide the users with insightful and valuable information for top-k recommendation support. Assuming a top-k query with a linear scoring function, the GIR indicates all the possible weight settings that produce exactly the same result as the original query. The GIR can be used as a guide for query weight refinement, as a sensitivity measure, and as a means for result caching. We propose a suite of scalable algorithms that exploit the geometric properties of the problem to achieve efficient GIR computation.

Chapter 5

Bucketization for Group Recommendation

In this chapter, we consider the heterogeneous group formation problem in group recommendation, which essentially is a bucketization problem. This problem is related to the number partitioning problem in Artificial Intelligence, or the balanced multi-way number partitioning (BMNP) problem in particular. Specifically, BMNP seeks to split a collection of numbers into subsets, or groups, with (roughly) the same cardinality and subset sum.

The BMNP problem is NP-hard, and there are several exact and approximate algorithms for it. However, existing exact algorithms solve only the simpler, balanced two-way number partitioning variant, whereas the most effective approximate algorithm, BLDM, may produce widely varying subset sums. In this chapter, we introduce LRM algorithm that lowers the expected spread in subset sums to one-third that of BLDM for uniformly distributed numbers and odd subset cardinalities. We also propose Meld, a novel strategy for skewed number distributions. A combination of LRM and Meld leads to a heuristic technique that consistently achieves a narrower spread of subset sums than BLDM.

5.1 Preliminaries

Number partitioning is a category of NP-complete problems that has been studied extensively. One of its variants is balanced multi-way number partitioning (BMNP). The input of BMNP is a set S of n numbers and a positive integer k; the output is a partition of S into subsets. The *subset sum* is the sum of numbers in a subset, and the *subset cardinality* indicates how many numbers it contains. The objective in BMNP is to partition S into k subsets such that (i) the cardinality of each subset is either $\lfloor \frac{n}{k} \rfloor$ or $\lceil \frac{n}{k} \rceil$ numbers, and (ii) the *spread* (i.e., the difference) between the maximum and minimum subset sum is minimized. BMNP has a wide range of applications, including multiprocessor scheduling [28] and VLSI manufacturing [74].

BMNP is NP-hard [28]. To deal with its hardness, most existing approaches focus on suboptimal solutions (where the spread is higher than the minimum possible). Among them, BLDM is currently the most effective; originally proposed for balanced 2-way partitioning in [81], it was subsequently generalized to arbitrary k in [51]. BLDM first divides S into k-tuples (i.e., batches of k numbers). Then it selects and *folds* two of them, i.e., it couples their numbers and places the k produced pair sums into a new k-tuple, aiming to offset the variation within the original tuples. Folding continues iteratively until a single k-tuple remains; each of its elements corresponds to one of the k returned subsets.

As BLDM is currently the most effective approximate algorithm for BMNP, we examine it in detail. Given a set of numbers S, BLDM performs balanced k-way partitioning as follows. First, S is padded with zero-value numbers, so that $n = |S| = bk$ for some integer b. The numbers in S are then sorted in descending order. The sorted sequence, denoted by $(v_1, v_2, ..., v_n)$, is split into b disjoint k-tuples $p_i = (v_{(i-1)k+1}, v_{(i-1)k+2}, ..., v_{ik})$, for $1 \leq i \leq b$. The spread in each p_i is $\delta(p_i) = v_{(i-1)k+1} - v_{ik}$. Next, BLDM repeatedly replaces the two k-tuples p_i and p_j with the largest spreads with a new k-tuple p'; p' is the result of *folding* p_i with p_j, by adding the first (largest) value in p_i with the last (smallest) value in p_j, the second (largest) value in p_i with the second last (smallest) in p_j, and so on. This process continues until a single k-tuple remains; each of the k elements in this tuple corresponds to one subset of the produced partitioning. The pseudocode of BLDM and the fold operation is given in Algorithms 5.1 and 5.2, respectively.

Algorithm 5.1: BLDM($P_1, P_2, \ldots, P_b$)

```
1 Set $\mathbf{P} = \{P_1, P_2, \ldots, P_b\}$;
2 while $|\mathbf{P}| > 1$ do
3     Let $P_\alpha$ be the tuple in $\mathbf{P}$ with highest spread $\delta(P_\alpha)$;
4     $\mathbf{P} = \mathbf{P} \backslash \{P_\alpha\}$;
5     Let $P_\beta$ be the tuple in $\mathbf{P}$ with highest spread $\delta(P_\beta)$;
6     $\mathbf{P} = \mathbf{P} \backslash \{P_\beta\}$;
7     $P_\gamma$ = Fold($P_\alpha$, $P_\beta$);
8     $\mathbf{P} = \mathbf{P} \cup \{P_\gamma\}$;
```

Algorithm 5.2: Fold(P_α, P_β)

```
  Input: $m$-tuple $P_\alpha = [v_{\alpha,1}, v_{\alpha,2}, \ldots, v_{\alpha,m}]$ and $P_\beta = [v_{\beta,1}, v_{\beta,2}, \ldots, v_{\beta,m}]$
  Output: $m$-tuple $P_\gamma = [v_{\gamma,1}, v_{\gamma,2}, \ldots, v_{\gamma,m}]$
1 for $i = 1$ to $m$ do
2     $v_{\gamma,i} = v_{\alpha,i} + v_{\beta,m-i+1}$;
3 Sort the frequencies in $P_\gamma$ in ascending order;
```

While efficient, BLDM often produces partitions very far from optimal. We demonstrate that its spread is particularly high when the numbers in S follow a roughly uniform or a skewed distribution, and identify the reasons behind this. Motivated by these weaknesses, we propose heuristic algorithms LRM and Meld, each tailored to uniform and skewed data respectively. We prove analytically and empirically in the sections that follow, that LRM reduces the expected spread to one third that of BLDM for uniform data. Meld, on the other hand, is shown to be significantly more effective than BLDM for non-uniform distributions (e.g., Zipf, normal, etc). Finally, we incorporate LRM, Meld and BLDM into a Hybrid algorithm that dynamically adapts to different data characteristics.

5.2 Limitations of BLDM

Although BLDM is efficient, it performs poorly in two general scenarios. We explain each scenario with the aid of an example.

Example 3 *Consider a balanced* 4*-way partitioning on set* $S = \{12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1\}$*. BLDM divides* S *into three* 4*-tuples* $p_1 = (12, 11, 10, 9)$*,* $p_2 = (8, 7, 6, 5)$ *and* $p_3 = (4, 3, 2, 1)$*. Then, the two* 4*-tuples with largest spread, say* p_1 *and* p_2 *(since all the* 4*-tuples have the same spread of* 3*), are folded to form a new* 4*-tuple* $p' = (17, 17, 17, 17)$*. Next,* p' *is folded with* p_3*, yielding the final* 4*-tuple* $(21, 20, 19, 18)$*. Tracing back the numbers that contributed to the final tuple, we derive the* 4*-way partitioning* $(\{12, 5, 4\},\{11, 6, 3\},\{10, 7, 2\},\{9, 8, 1\})$*, with a spread in subset sums of* $21-18 = 3$*. This spread is as high as that of the original* 4*-tuples. In comparison, an optimal solution is* $(\{12, 5, 2\},\{11, 8, 1\},\{10, 7, 3\}, \{9, 6, 4\})$*, with a spread of just 1.*

The scenario in Example 3 occurs when the numbers in S follow a uniform (or roughly uniform) distribution and b is odd. In folding pairs of k-tuples, BLDM essentially offsets their spreads against each other. Since all the initial k-tuples have nearly the same spread (for uniform data), when b is odd BLDM will succeed in canceling out pairs of k-tuples as it is designed to do, leaving nothing to compensate for the last remaining k-tuple. Consequently, the reported partitioning inherits the spread of this last k-tuple. Figure 5.1 illustrates this situation in a uniform data scenario where $k = 4$ and $b = 3$. Each number is represented as a bar with height equal to its value. Folding p_1 and p_2 leaves no space for spread offsetting when p_3 is appended to derive the final k-tuple.

Example 4 *Consider a balanced* 4*-way partitioning on set* $S = \{60, 52, 40, 26, 19, 16, 14, 12, 10, 9, 6, 5\}$*. After dividing* S *into three* 4*-tuples* $p_1 = (60, 52, 40, 26)$*,* $p_2 = (19, 16, 14, 12)$ *and* $p_3 = (10, 9, 6, 5)$*, BLDM folds* p_1 *and* p_2 *first, since they have the largest spreads of* 34 *and* 7*. The resulting* 4*-tuple* $p' = (72, 66, 56, 45)$

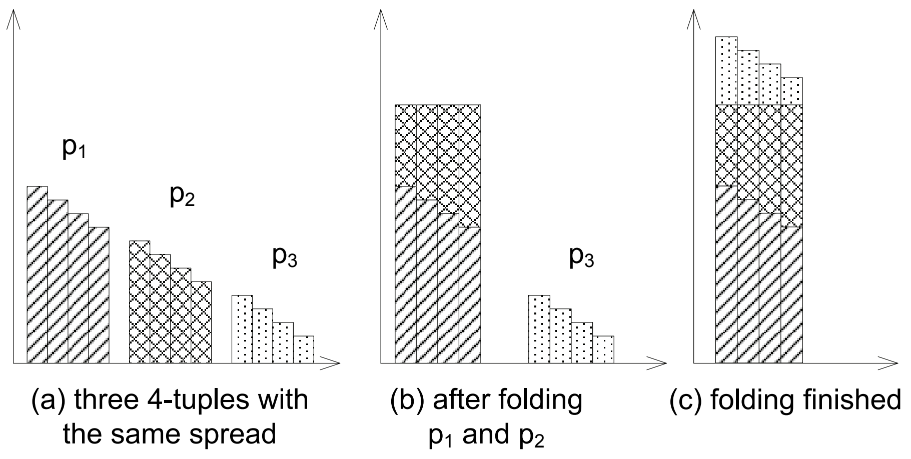

p1
p2
p3
(a) three 4-tuples with the same spread
p3
(b) after folding p1 and p2
(c) folding finished

Figure 5.1: Folding process of BLDM on three 4-tuples

is then folded with p_3*, yielding the final* 4*-tuple* $(77, 72, 65, 55)$*. Thus, the reported partitioning is* $(\{60, 12, 5\}, \{52, 14, 6\}, \{40, 16, 9\}, \{26, 19, 10\})$*, with spread* $77-55 = 22$*. In contrast, the optimal solution* $(\{60, 6, 5\}, \{52, 10, 9\}, \{40, 14, 12\}, \{26, 19, 16\})$ *has a spread of only 10.*

The scenario illustrated in Example 4 occurs when the numbers in S follow a skewed distribution. The poor performance of BLDM stems from the k-tuple boundaries imposed right at the beginning, causing each subset sum in the final solution to be derived from exactly one number in each initial k-tuple. If the spread δ of some k-tuple is so large as to dominate the total spread, Σ, of all the other k-tuples, then BLDM is unable to obtain a final k-tuple with a spread narrower than $\delta - \Sigma$.

The shortcomings of BLDM highlighted above need to be addressed, owing to the abundance of both uniformly distributed and of highly skewed data in real applications.

5.3 Algorithms for Balanced Bucketization

5.3.1 The LRM algorithm

In this section, we introduce an algorithm motivated by the first scenario where BLDM works poorly, i.e., uniformly distributed numbers with odd subset cardinality b. We begin with the case of $b = 3$, before extending to larger odd values of b.

The rationale of LRM is as follows. Let there be three k-tuples p_1, p_2 and p_3 with means of μ_1, μ_2 and μ_3, respectively. If folding the initial k-tuples could achieve a perfect balanced partitioning, each subset sum in the final k-tuple would be equal to $\mu_1+\mu_2+\mu_3$. Targeted at this ideal subset sum, we design our algorithm to fold the three tuples simultaneously (instead of performing two consecutive pair-wise folds). Specifically, we optimistically form each subset from the leftmost[1] (L) number v_L in one tuple, the rightmost (R) number v_R in another tuple, and a compensating number somewhere in the middle (M) of the remaining tuple that is closest to $\sum_{i=1}^{i=3} \mu_i - v_L - v_R$. This gives rise to LRM.

In forming a subset sum, LRM always performs the M operation (to pop a compensating number) in the input tuple that currently has the smallest spread[2]. The L and R operations are carried out on the tuples presently having the largest and second largest spreads, respectively. This is meant to reduce the chance of picking a compensating number that might be more useful for a subsequent subset sum; since the compensating number comes from the tuple with the smallest spread, the impact of a suboptimal choice (of compensating number) is small because the tuple is likely to hold other numbers with a similar value. Note that the strategy of picking the leftmost and rightmost numbers from the tuples currently having the highest spreads is consistent with the largest-first differencing strategy in KK [44] and BLDM [51]. Algorithm 5.3 describes the LRM method.

To illustrate LRM, we refer again to Example 1 where we have $p_1 = (12, 11, 10, 9)$, $p_2 = (8, 7, 6, 5)$, $p_3 = (4, 3, 2, 1)$, and $\mu_1 + \mu_2 + \mu_3 = 19.5$.

[1]Since each k-tuple is sorted in descending order, its leftmost number has the maximum positive offset from the mean, whereas the rightmost number has the maximum negative offset.

[2]We say "currently" because as numbers are removed from the tuples, their spread is updated to reflect their remaining contents.

At first, LRM arbitrarily picks p_1 and p_2 for the L and R operations, as all three tuples have the same spread of 3. Popping 12 from p_1 and 5 from p_2, it chooses the compensating number from p_3 to be 2, because it has the closest value to $19.5 - 12 - 5 = 2.5$. This produces the first subset $\{12, 5, 2\}$. Now the spread in the remainders $p_1 = (11, 10, 9)$, $p_2 = (8, 7, 6)$ and $p_3 = (4, 3, 1)$ becomes $2, 2$ and 3, respectively, causing LRM to pick p_3 and p_1 for the L and R operations in the second round. With numbers 4 from p_3 and 9 from p_1, 6 is chosen as the compensating number in p_2 that is nearest to $19.5 - 4 - 9 = 6.5$, leading to the second subset $\{4, 9, 6\}$. Repeating this process, we get the third and fourth subsets, $\{3, 10, 7\}$ and $\{1, 11, 8\}$. The final partitioning of $(\{12, 5, 2\}, \{4, 9, 6\}, \{3, 10, 7\}, \{1, 11, 8\})$, with a spread of only 1, matches the optimal solution and is a major improvement over BLDM's spread of 3.

Proposition 1 *Consider three k-tuples p_1, p_2 and p_3, comprising numbers that follow a uniform distribution. By combining p_1, p_2 and p_3 simultaneously, LRM generates a k-tuple p' with an expected spread $\delta(p')$ that is one third of that generated by BLDM.*

Proof. Given that the numbers in S are uniformly distributed, the three input k-tuples have roughly the same spread of, say, δ. Moreover, the expected difference between successive numbers in p_1, p_2 and p_3 is $C = \frac{\delta}{k-1}$. In each round of LRM, let p_L, p_R and p_M denote the k-tuples that produce the leftmost (v_L), rightmost (v_R) and compensating (v_M) numbers. Let μ_L, μ_R and μ_M denote the respective mean of the tuples. The subset sum generated is:

$$v_L + v_R + v_M = \sum_{i=1}^{3} \mu_i + (v_L - \mu_L) + (v_R - \mu_R) + (v_M - \mu_M) \tag{5.1}$$

The LRM strategy removes numbers from the k-tuples according to a rotating pattern:

Round	p_1	p_2	p_3
1	leftmost	rightmost	middle
2	rightmost	middle	leftmost
3	middle	leftmost	rightmost
4	2^{nd} leftmost	2^{nd} rightmost	middle
5	2^{nd} rightmost	middle	2^{nd} leftmost

...

Thus, the leftmost number of a tuple is always matched with the rightmost number of another tuple, the 2^{nd} leftmost number is matched with the 2^{nd} rightmost number, etc. So, we have $(v_L - \mu_L) \approx (\mu_R - v_R)$, and Formula (5.1) simplifies to:

$$v_L + v_R + v_M = \sum_{i=1}^{3} \mu_i + (v_M - \mu_M) \tag{5.2}$$

$v_M - \mu_M$ is small in the early rounds, but grows gradually as the numbers in the middle of the tuples are used up. In round $i \in [1, k]$, $|v_M - \mu_M| = (\lfloor \frac{i-1}{6} \rfloor + 0.5)C$ for even k, whereas $|v_M - \mu_M| = (\lceil \frac{i+3}{6} \rceil - 1)C$ for odd k. Therefore, the last two rounds produce subset sums that, respectively, fall below and above $\sum_{i=1}^{3} \mu_i$ by the widest margin, and the difference between those two subset sums determines the final spread of the partitioning solution. In fact, the final spread is twice the value $|v_M - \mu_M|$ of the final round k, so:

$$spread = \begin{cases} 2(\lfloor \frac{k-1}{6} \rfloor + 0.5)\frac{\delta}{k-1} & \text{for even } k \\ 2(\lceil \frac{k+3}{6} \rceil - 1)\frac{\delta}{k-1} & \text{for odd } k \end{cases} \tag{5.3}$$

Recall that BLDM yields a final spread of δ, so the spread ratio of LRM w.r.t. BLDM converges to 1:3 for large k. □

LRM extends easily to odd values of b larger than 3. Specifically, the three k-tuples with the largest spreads are combined through LRM into an interim k-tuple with a small expected spread. The interim and the remaining k-tuples are iteratively folded pairwise, in the manner of BLDM, to cancel out their spreads until we are left with a single tuple. LRM has a time complexity of

$O(n \log n)$, as we need to keep the numbers in each k-tuple sorted so as to find compensating numbers in logarithmic time (see line 7 in Algorithm 5.3).

Algorithm 5.3: LRM

```
  Input: $k$-tuples $p_1$, $p_2$ and $p_3$ with means $\mu_1$, $\mu_2$ and $\mu_3$
  Output: a final $k$-tuple
1 $Sum = \mu_1 + \mu_2 + \mu_3$;
2 $p' = \emptyset$;
3 while $|p'| < k$ do
4     let $p_L, p_R, p_M$ be the input $k$-tuple with the largest, second largest,
      and smallest spread, respectively;
5     $v_L$ = the leftmost number removed from $p_L$;
6     $v_R$ = the rightmost number removed from $p_R$;
7     $v_M$ = the compensating number removed from $p_M$ that is closest to
      $(Sum - v_L - v_R)$;
8     $p' = p' \cup \{v_L + v_R + v_M\}$;
9 return $p'$;
```

5.3.2 The Meld Algorithm

Our second algorithm is designed to handle skewed data, the second scenario in which BLDM falls short. In case of skewed data, some k-tuples (e.g., tuple p_1 in Example 4) have a particularly large spread that cannot be effectively canceled out by folding with the remaining, smaller-spread k-tuples.

For ease of presentation, we consider the case where we have three input k-tuples (i.e., $b = 3$) before generalizing to arbitrary b. Let the tuples be p_1, p_2 and p_3 and suppose that the spread in p_1 is larger than the spread of the other two combined, i.e., $\delta(p_1) > \delta(p_2) + \delta(p_3)$. Assuming that $\delta(p_2) > \delta(p_3)$, BLDM would fold p_1 with p_2, and then the interim tuple with p_3; this achieves a final spread that is no less than $\delta(p_1) - \delta(p_2) - \delta(p_3) > 0$.

To avoid the pitfall, we deviate from BLDM's principle of eliminating the spread *whenever* a pair of k-tuples is folded, because in certain occasions a large interim spread may be needed to counterbalance the excessive spread of another k-tuple. Specifically, we combine p_2 and p_3 into an interim tuple p' with a spread $\delta(p')$ that is larger than $\delta(p_2) + \delta(p_3)$ and as close to $\delta(p_1)$

as possible, so that the subsequent folding of p' with p_1 can cancel out their respective spreads.

Our Meld algorithm achieves this by "melding" p_2 and p_3, that is, by uniting $p_2 = (v_{2,1}, \ldots, v_{2,k})$ and $p_3 = (v_{3,1}, \ldots, v_{3,k})$ into a common sorted sequence $p_\cup = (v_1, v_2, \ldots,\ v_{2k-1}, v_{2k})$, and then pairing its numbers so that the produced k-tuple p' has values that are (almost) uniformly spaced in $\left[\mu_2 + \mu_3 - \frac{\delta(p_1)}{2}, \mu_2 + \mu_3 + \frac{\delta(p_1)}{2}\right]$. We first identify in $p_\cup$ two number pairs $\mathcal{A} = \{v_i, v_j\}$ and $\mathcal{B} = \{v_l, v_m\}$ that add up to give $v_i + v_j$ and $v_l + v_m$ at the two extreme ends of p', such that $(v_i + v_j) - (v_l + v_m) \approx \delta(p_1)$. The next two number pairs $\mathcal{A}' = \{v'_i, v'_j\}$ and $\mathcal{B}' = \{v'_l, v'_m\}$, intended for the second position from the left and right of p', are chosen to meet condition $(v'_i + v'_j) - (v'_l + v'_m) \approx \delta(p_1) - \delta^-$; $\delta^- = \frac{2\delta(p_1)}{k-1}$ is the desired rate of decline among the numbers in p'. This process is repeated to complete p'.

In pairing the numbers in $p_\cup$ as explained above, one may suggest using an exhaustive search. However, with $O(k^2)$ possible pairs, it takes $O(k^4)$ time to compute the difference between any two of them. Recall that $n = bk$, so the time complexity is $O(n^4)$, making exhaustive search impractical for large n. Instead, we adopt a heuristic search strategy.

As shown in Algorithm 5.4 (lines 4 and 5), in the first iteration we place the largest (v_1) and smallest (v_{2k}) numbers of $p_\cup$ into pairs $\mathcal{A}$ and $\mathcal{B}$, respectively. For the second number in $\mathcal{A}$, we scan from v_{2k-1} back to v_2. Simultaneously, we scan from v_2 to v_{2k-1} to complete $\mathcal{B}$. The scan stops as soon as we encounter v_i and v_{2k-i+1} such that $(v_1 + v_{2k-i+1}) - (v_{2k} + v_i) \geq \delta(p_1)$ for some $2 \leq i \leq 2k-1$; these numbers complete the pairs, i.e., $\mathcal{A} = \{v_1, v_{2k-i+1}\}$ and $\mathcal{B} = \{v_i, v_{2k}\}$. The four chosen numbers are removed from $p_\cup$, and we proceed to find the next two pairs $\mathcal{A}$ and $\mathcal{B}$ (see lines 4 to 14). The process is repeated until $p_\cup$ becomes empty or it is left with two numbers only (see lines 6 to 8). The complexity of Meld is $O(k^2)$, since in the worst case it takes $O(k^2)$ steps to process all the numbers in $p_\cup$.

Algorithm 5.4: Meld

```
Input: $k$-tuple $p_1, p_2$ and $p_3$ with $\delta(p_1) > \delta(p_2) + \delta(p_3)$
Output: a final $k$-tuple $p'$
1  $p_\cup = p_2 \cup p_3$, sort $p_\cup$ in descending order;
2  $\delta^- = \frac{2\delta(p_1)}{k-1}$; $p' = \emptyset$;
3  while $|p_\cup| > 0$ do
4      let $v_1$ be the largest number in $p_\cup$;
5      let $v_{|p_\cup|}$ be the smallest number in $p_\cup$;
6      if $|p_\cup| = 2$ then
7          $p' = p' \cup \{v_1 + v_{|p_\cup|}\}$;
8          break out of the while loop;
9      for $i = 2$ to $|p_\cup| - 1$ do
10         if $(v_1 + v_{|p_\cup|-i+1}) - (v_{|p_\cup|} + v_i) \geq \delta(p_1)$ then
11             break out of the for loop;
12     $p' = p' \cup \{v_1 + v_{|p_\cup|-i+1}\} \cup \{v_{|p_\cup|} + v_i\}$;
13     Remove $v_1, v_{|p_\cup|}, v_i, v_{|p_\cup|-i+1}$ from $p_\cup$;
14     $\delta(p_1) = \delta(p_1) - \delta^-$;
15 fold $p_1$ with $p'$ and store the result in $p'$;
16 return $p'$;
```

To illustrate the algorithm, we apply Meld to Example 4, where $S = \{60, 52, 40, 26, 19, 16, 14, 12, 10, 9, 6, 5\}$. Here $p_\cup = p_2 \cup p_3 = \{19, 16, 14, 12, 10, 9, 6, 5\}$, $\delta(p_1) = 60 - 26 = 34$ and $\delta^- = 34 * 2/3 = 22.7$. After initializing $\mathcal{A} = \{19, \_\}$ and $\mathcal{B} = \{5, \_\}$, Meld scans from 6 towards 16 to complete $\mathcal{A}$ and from 16 to 6 for $\mathcal{B}$. The scan produces numbers 16 and 6, without satisfying the condition in line 10 of Algorithm 5.4. The iteration is completed by adding $19 + 16$ and $6 + 5$ to p', and removing these four numbers from $p_\cup$. The next iteration starts with 14 in $\mathcal{A}$ and 9 in $\mathcal{B}$. The scan in lines 9 to 11 yields $\mathcal{A} = \{14, 12\}$ and $\mathcal{B} = \{10, 9\}$, again without passing the test in line 10. Now we have $p' = (19 + 16, 6 + 5, 14 + 12, 9 + 10)$=$(35, 26, 19, 11)$. Folding p_1 with p' produces the final partitioning $(60+11, 52+19, 40+26, 26+35) = (71, 71, 66, 61)$, with an optimal spread of 10, and achieving a 50% improvement over BLDM's spread of 22.

The following proposition shows formally that melding p_2 and p_3 indeed achieves the objective of a wider spread than folding.

Proposition 2 *Melding k-tuples p_2 and p_3 produces a k-tuple p' in which each* *element is the sum of two numbers in $p_2 \cup p_3$, such that $\delta(p') \geq \delta(p_2) + \delta(p_3)$.*

Proof. Since the numbers in every k-tuple are sorted in descending order by construction, we have:

$$\delta(p_2) + \delta(p_3) = (v_{2,1} - v_{2,k}) + (v_{3,1} - v_{3,k}) = (v_{2,1} + v_{3,1}) - (v_{2,k} + v_{3,k}) \tag{5.4}$$

Now consider the number pairs $\mathcal{A}$ and $\mathcal{B}$ found in the first iteration of Algorithm 5.4. If the scan in lines 9 to 11 completes without passing the test in line 10, $\mathcal{A}$ holds the two largest numbers v_1 and v_2 in $p_\cup$, and $v_1 + v_2 \geq v_{2,1} + v_{3,1}$; moreover, $\mathcal{B}$ has the two smallest numbers v_{2k-1} and v_{2k} in $p_\cup$, and $v_{2k-1} + v_{2k} \leq v_{2,k} + v_{3,k}$. Thus, $\delta(p') \geq \delta(p_2) + \delta(p_3)$. If the scan terminates early, then $\mathcal{A} = \{v_1, v_{2k-i+1}\}$ and $\mathcal{B} = \{v_i, v_{2k}\}$ lead to $\delta(p') = (v_1 + v_{2k-i+1}) - (v_i + v_{2k})$ that is just above $\delta(p_1)$, so $\delta(p') > \delta(p_2) + \delta(p_3)$. □

Meld extends to situations where S is split into more than three k-tuples. We repeatedly test whether the three k-tuples p_i, p_j and p_l with the largest spreads satisfy the condition $\delta(p_i) > \delta(p_j) + \delta(p_l)$. If so, we perform melding to combine them into a new k-tuple; if not, we resort to a folding operation on p_i and p_j. Through this combination of melding and folding operations, we eventually obtain a single k-tuple that represents the final partitioning.

5.3.3 The Hybrid Algorithm

Where the numbers to be partitioned are known (or have been tested) to follow a uniform or skewed distribution, the LRM and Meld algorithms introduced in the earlier sections can be applied to produce high-quality partitioning solutions. To cope with arbitrary input data without a priori knowledge of their distribution, we incorporate LRM, Meld and the folding operation of BLDM into a Hybrid algorithm. Specifically, whenever the three k-tuples p_i, p_j and p_l with the largest spreads satisfy condition $\delta(p_i) > \delta(p_j) + \delta(p_l)$, Meld is executed; otherwise, we invoke LRM or folding, depending on whether the number of remaining k-tuples is odd or even. This process is repeated until we are left with a single k-tuple. Hybrid simply switches between Meld and LRM, according to the comparison results of at most $O(n)$ conditions $\delta(p_i) > \delta(p_j) + \delta(p_l)$. Since the time complexities of Meld and LRM are $O(n^2)$ and $O(n \log n)$, respectively, the time complexity of Hybrid is also $O(n^2)$.

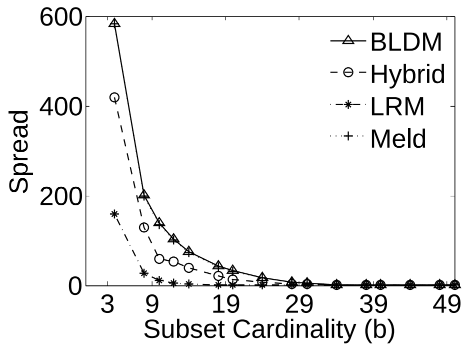
5.4 Empirical Validation

To investigate how LRM, Meld and Hybrid perform relative to BLDM, we implemented them in C++ and tested them on a PC equipped with an Intel Core Duo 2GHz CPU and 2GB memory. We employ three kinds of datasets: (a) The first contains numbers that are normally distributed with a mean of 1000 and variance from $0.1 \times mean$ to $1.0 \times mean$; (b) The second dataset is uniformly distributed, with numbers drawn from $[0, 2000]$; (c) The last dataset follows a Zipf distribution $f(x) = \frac{1}{x^\theta}$ in $[0, 10000]$, with θ from 0.1 to 2. Each reported measurement is the average over 100 trials.

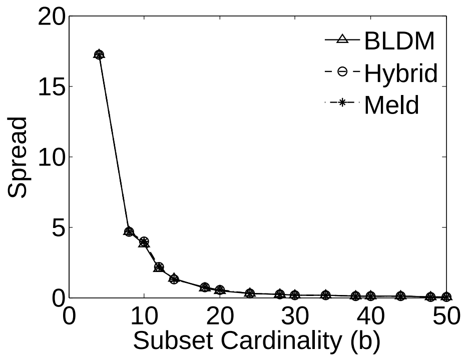
5.4.1 Impact of Subset Cardinality (b)

In the first experiment, we measure the spread as we vary the subset cardinality b from 3 to 50 (because LRM and Meld are designed for $b \geq 3$), while fixing the number of subsets k to 500; this setting corresponds to a balanced 500-way partitioning. In tandem with the varying b, the dataset size n increases from 3×500 to 50×500.

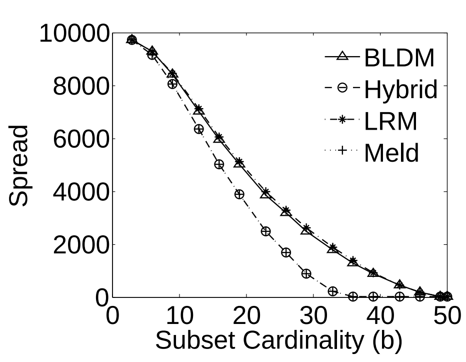
For the uniform dataset, we report the results separately for odd and even b, because LRM is designed specifically for odd b settings. Figure 5.2(a) shows the spreads for odd b values, with $b = 3, 5, 7..., 49$ on the x-axis. The results confirm that LRM achieves $\frac{1}{3}$ the spread of BLDM, as proven in Proposition 1. Meld performs as poorly as BLDM, which is not surprising; since the condition for melding is never met for k-tuples with nearly equal spreads, Meld resorts to folding operations instead. Hybrid is slightly inferior to LRM, because the



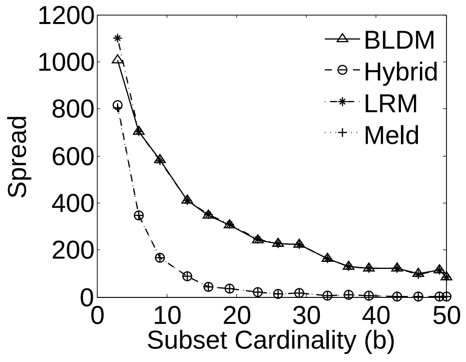
(a) Uniform Dist. (b is odd)



(b) Uniform Dist. (b is even)



(c) Zipf Dist.



(d) Normal Dist.

Figure 5.2: Performance comparison with k fixed at 500

former may invoke LRM multiple times (instead of just once), which is not ideal for uniformly distributed data.

Figure 5.2(b) illustrates the spread for uniformly distributed data and even b values, with $b = 4, 6, 8..., 50$ on the x-axis (for even b LRM degenerates to BLDM and is thus omitted from the chart). BLDM achieves partitioning solutions with very small spreads. This graceful performance is possible only because in this experiment there is an even number of k-tuples with similar spreads that successfully offset each other. Since the execution conditions for LRM and melding are not met, Hybrid relies exclusively on folding operations, leading to the same partitioning solutions as BLDM. Likewise for Meld.

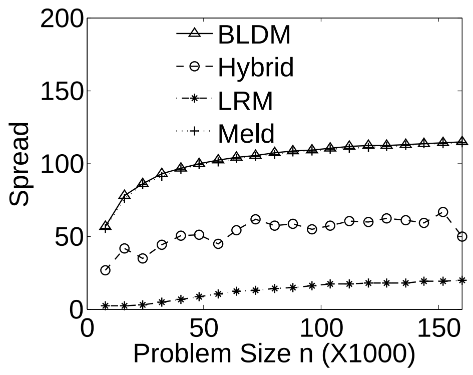
Next, we turn to the results obtained with the Zipf dataset, depicted in Figure 5.2(c). Due to space limitation, the figure only includes results for $\theta = 1.2$ (where about 95% of the numbers fall in 20% of the data space). Clearly, BLDM does not handle skewed data well, for the reasons discussed in

Section 5.2. The pure LRM exhibits similar performance as BLDM because, even when b is odd, the data skew in the Zipf distribution is too high to be offset by applying LRM only once on the first three k-tuples. In comparison, Meld handles skewed data successfully, especially for $b > 10$. We note that Meld works even better in (omitted) experiments with a higher θ. Hybrid behaves similarly to Meld, since it invokes the melding operation extensively.

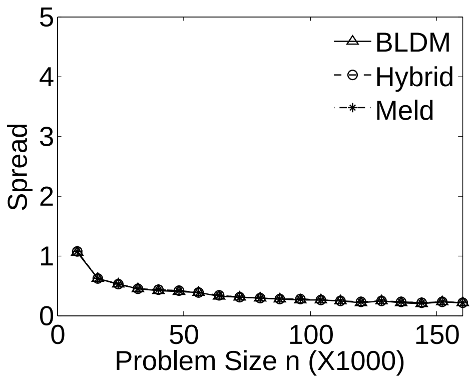
Turning to normally distributed data, we plot results for a variance of $0.6 \times mean$ in Figure 5.2(d) (the relative performance of the algorithms is similar across all variances tested). Meld and Hybrid outperform BLDM vastly, producing partitioning solutions with spreads less than $\frac{1}{10}$ of those generated by BLDM when b ranges from 18 to 50. The reason for this interesting result lies in the property of the normal distribution, which contains relatively few large and small numbers, while the majority of the values cluster around the mean. Thus, when the dataset S is carved into k-tuples, the first few have steep spreads, followed by many tuples with smaller and smaller spreads, before the pattern reverses. Meld and Hybrid are able to offset those k-tuples with large spreads in the early iterations, before switching to folding operations until termination. BLDM and LRM, however, are not adept at handling such data.

5.4.2 Impact of the Problem Size (n)

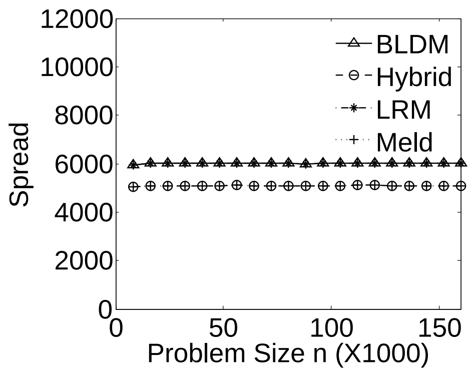
The next experiment studies the effect of the dataset size, i.e., n, on the various algorithms. We increase n progressively from $8,000$ to $160,000$ while fixing the subset cardinality b. We report only results for $b = 16$ (as well as $b = 15$ for uniform data); results for other b settings follow a similar trend to those shown here. Note that k increases in tandem with n, from 500 to $10,000$. For $b = 15$ and uniform data (Figure 5.3(a)) LRM and Hybrid are consistently better than BLDM. In Figure 5.3(b), for $b = 16$ and uniform data, as n increases, the spreads produced by all the algorithms drop and then remain close to zero, verifying that even cardinalities are generally easier to handle. For Zipf



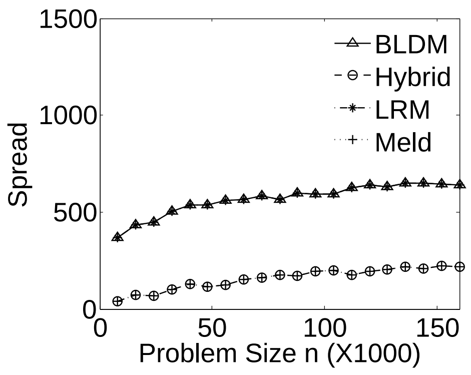
(a) Uniform Dist. (b=15)



(b) Uniform Dist. (b=16)



(c) Zipf Dist. (b=16)



(d) Normal Dist. (b=16)

Figure 5.3: Performance comparison with fixed b

numbers, Figure 5.3(c) shows that Meld and Hybrid outperform BLDM consistently by about 17% across different n settings. For normally distributed data (Figure 5.3(d)), Meld and Hybrid achieve spreads that are about 10% to 34% those of BLDM. Across all experiments in Figure 5.3, the relative performance of LRM, Meld, Hybrid and BLDM is stable with respect to n, because it is the data distribution that has a larger effect on their effectiveness.

5.4.3 Computation Overhead

Finally, we consider the CPU overhead of the various algorithms. At $n = 25,000$ and $k = 500$, BLDM executes in under 80 ms in all of our experiments. Despite having the same time complexity $O(n \log n)$ as BLDM, the actual CPU time incurred by LRM is higher, at about 100 ms. In contrast, Meld and Hybrid are more computationally intensive, owing to the melding operation; their CPU times are less than 600 ms for the normal distribution, 780 ms for the Zipf data, and 200 ms for the uniform dataset. Nevertheless, LRM, Meld and Hybrid are all practical, having very reasonable execution times. It is worth mentioning that even for one million numbers (with, say, $k = 500$ and normal distribution) they all complete in less than 4 seconds.

5.5 Summary

In this chapter, we study the bucketizatoin problem for heterogeneous group formation in group recommendation. The bucketization problem is equivalent to a balanced k-way number partitioning problem. With the goal of designing practical solutions for large problem settings, we introduce two heuristic methods. The first, called LRM, utilizes a predictable pattern of adding numbers across three batches of k, to produce k partial sums that are similar in magnitude. The second method, Meld, employs a (seemingly counterintuitive) strategy of melding two batches of k numbers into a batch of k widely varying partial sums, before offsetting them against another high-variance batch. Furthermore, we combine LRM and Meld into a Hybrid algorithm that dynamically adapts to different data characteristics. Extensive experiments confirm the effectiveness of LRM and Meld for uniform and skewed data, respectively, while Hybrid consistently produces high-quality partitioning solutions within short execution times.

Chapter 6

Conclusion

In this chapter, we summarize the dissertation by reviewing the problem of recommendation support for multi-attribute databases and the work we have accomplished in the dissertation. Finally, we discuss some promising future work.

6.1 Dissertation Summary

In this dissertation we focus on three recommendation tasks, namely preference-overlap recommendation, top-k recommendation, and group recommendation. We show how each of the tasks may be augmented with additional useful information for effective recommendation support.

In Chapter 3, we propose the concept of direct neighbor (DN) for preference-overlap recommendation. Given a query object q, an object p is a DN of q if there exists some query window that exclusively retrieves p and q. DN offers useful information that could be submitted to the users along with the recommendation result. For instance, the DNs of a query object q collectively define an exclusive retrieval region (see Figure 3.1(b) in Chapter 3), which is a polygon that only covers q and no other objects. This region could help a user to adjust her preferences q in searching for more alternatives, or to monitor whether any new items overlap with her region of interest. To compute

DNs, we analyze the nature of the problem and propose efficient, I/O-optimal algorithms based on skyline processing and segment trees. We also introduce practical variants of DN query, i.e., the k-DN query and the All-DN query, and we provide novel and efficient solutions for each variant.

One limitation of our work in Chapter 3 is that different from NN query and top-k search, DN query itself cannot tell the relative importance among DNs according to their distance to query source q, because the DN concept is not based on distance, rather, it is defined by query windows. On the other hand, as dimensionality explodes, the number of DNs increases exponentially, hindering the usefulness of DN query. Also, similar to other queries such as NN query, the performance of our techniques for DN query will degenerate, due to the inability of spatial index structures to handle high-dimensional data.

In Chapter 4 we study the problem of global immutable region (GIR) computation for top-k recommendation. Given a user's top-k query q, the GIR is the maximal locus containing all query vectors q', such that the top-k result with respect to q' is the same as for q. In other words, GIR is the maximal polyhedron, within which q can be adjusted freely without altering the top-k result.

GIR conveys insightful information to the users. For example, GIR can tell the users how aggressive they should be when adjusting their preferences in order to get new recommendations, or how robust the recommendation result is if there are some perturbations in their preferences. Hence, GIR can be used in applications such as sensitivity analysis, top-k result caching, etc.

In Chapter 4 we answer several questions relating to the problem of GIR computation.

1. What does a GIR look like geometrically?

2. Can we efficiently compute the GIR, when the database is very large?

3. Can we compute the GIR for top-k recommender systems with non-linear

scoring functions?

For the first question, we formally define the concept of GIR based on properties of top-k query and half-space intersection [27], which reveal the shape and nature of GIR. With that understanding, we are able to identify the relationship between GIR and other sensitivity measures, i.e., STB and LIR, proposed in [70] and [53], respectively. To derive GIR, we devise three algorithms, namely skyline pruning (SP) and convex hull pruning (CP), as well as an advanced technique called facet pruning (FP). We show both analytically and empirically that FP is superior to SP and CP, because it computes a small proportion of facets of a convex hull, and subsequently only maintains this set of facets. The second problem is thus answered by our FP method completely. We give an affirmative answer to the third question, by showing that our SP method is capable of dealing with any monotone scoring functions, while CP, due to the nature of convex hull, can support certain types of concave functions. We also consider a variant of GIR, i.e., order-insentitive GIR, that omits the constraint on relative ordering among the top-k recommendations and only enforces the constraint of preserving the composition of the recommendation list.

Our work of GIR computation has several limitations. Firstly, GIR computation is based on the assumption that each attribute of the data is from a total-ordered (numerical) domain. If some attributes are from partial-ordered domains, then our techniques cannot be applied, because the top-k computation, half-space intersections, and convex hull construction require that any two tuples are comparable with respect to each of the attributes. Secondly, as we have shown in section 4.7.2, the proposed CP and FP methods may not be able to deal with more general function types, due to the nature of convex hull that the two methods rely on.

We address the group formation problem in group recommendation in Chapter 5. The task of group recommendation is to recommend items to a group of users with similar quantitative features, e.g., preferences, skill set, knowledge level, etc [3, 54]. Besides recommending items to groups, the problem of group formation is also an important issue in group recommendation [3]. We focus on heterogeneous group formation, where a heterogeneous group consists of users with diversified quantitative measures [71]. Heterogeneous groups can be very useful in applications such as education and learning [17], where they promote peer learning [41].

We treat the heterogeneous group formation problem as a bucketization problem which is equivalent to the balanced multi-way number partitioning in Artificial Intelligence. The goal of our bucketization problem is to find a grouping plan which partitions a collection of objects with different quantitative measures into a number of groups, such that each group holds the same number of objects, and for any two groups G_1 and G_2, the accumulated value of quantities of all objects in G_1 is as close as possible to that of G_2.

To solve the bucketization problem, we propose three algorithms, namely LRM, Meld, and Hybrid. LRM and Meld perform better on datasets with uniform and skewed distribution, respectively, while Hybrid is superior when the data distribution is not known.

The limitation of our work in Chapter 5 is that our techniques, i.e., LRM and Meld, are tailored for specific data distributions, thus none of them can consistently achieve the best performance on all data distributions. Even the Hybrid method, that combines the merits of LRM and Meld, still does not perform well on data with uniform distribution when subset cardinality is odd.

To summarize, in this dissertation we consider three different recommendation tasks, and study how to provide recommendation support for the users with useful and additional information. We have also designed effective and scalable techniques to compute this additional information.

6.2 Future Work

We conclude this dissertation by outlining several interesting and promising research problems for further work.

A direction for future work is DN search for containment queries that are not axis-parallel windows but arbitrary regions. Another challenging direction regards the exploding number of DNs with dimensionality – it would be useful to devise techniques that prioritize among the DNs and possibly choose/compute only a subset of them.

Another direction of improvement regards our global immutable region (GIR) for top-k recommendation. Our techniques in Chapter 4 are for exact GIR computation. It would be meaningful to design techniques that can give an approximate GIR under some user constraints, such as error bound or limited response time. This extension will be useful in time-critical applications such as online business analysis, real-time product recommendation, online stock market monitoring, etc.

Bibliography

[1] Apartments - floor preference? Website. http://www.indianrealestateboard.com/forums/showthread.php/7938-Apartments-Floor-Preference.

[2] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):734–749, 2005.

[3] Sihem Amer-Yahia, Senjuti Basu Roy, Ashish Chawlat, Gautam Das, and Cong Yu. Group recommendation: Semantics and efficiency. *Proceedings of the VLDB Endowment, PVLDB*, 2(1):754–765, 2009.

[4] Lars Arge, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica*, 47(1):1–25, 2007.

[5] Brian Babcock and Chris Olston. Distributed top-k monitoring. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 28–39, 2003.

[6] Jonathan Backer and J Mark Keil. The bichromatic rectangle problem in high dimensions. In *Proceedings of the 21st Annual Canadian Conference on Computational Geometry, CCCG*, pages 157–160, 2009.

[7] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, 1996.

[8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 322–331, 1990.

[9] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 322–331, 1990.

[10] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the average number of maxima in a set of vectors and applications. *Journal of the ACM*, 25(4):536–543, 1978.

[11] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Computing Survey*, 11(4):397–409, 1979.

[12] Jon Louis Bentley and Derick Wood. An optimal worse case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, 100(7):571–577, 1980.

[13] Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. *Computational geometry: algorithms and applications*. Springer, 2008.

[14] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is 'nearest neighbor' meaningful? In *Proceedings of the 7th International Conference on Database Theory, ICDT*, pages 217–235, 1999.

[15] Christian Böhm and Hans-Peter Kriegel. Determining the convex hull in large multidimensional databases. In *Proceedings of the 3rd International*

Conference on Data Warehousing and Knowledge Discovery, DaWaK, pages 294–306, 2001.

[16] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proceedings of the 17th IEEE International Conference on Data Engineering, ICDE*, pages 421–430, 2001.

[17] David Boud, Ruth Cohen, and Jane Sampson. *Peer Learning in Higher Education: Learning from & with Each Other*. Psychology Press, 2001.

[18] Timothy M Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry*, 16(4):361–368, 1996.

[19] D. R. Chand and S. S. Kapur. An algorithm for convex polytopes. *Journal of the ACM*, 17(1):78–86, 1970.

[20] Yuan-Chi Chang, Lawrence Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R. Smith. The onion technique: Indexing for linear optimization queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 391–402, 2000.

[21] Bernard Chazelle. An optimal convex hull algorithm and new results on cuttings. In *FOCS*, pages 29–38, 1991.

[22] Lei Chen and Xiang Lian. Efficient processing of metric skyline queries. *IEEE Transactions on Knowledge and Data Engineering*, 21(3):351–365, 2009.

[23] H. Christopher Frey and Sumeet R. Patil. Identification and Review of Sensitivity Analysis Methods. *Risk Analysis*, 22(3):553–578, 2002.

[24] Kenneth Clarkson, Kurt Mehlhorn, and Raimund Seidel. Four results on randomized incremental constructions. *Computational Geometry*, 3(4):185–212, 1993.

[25] Douglas Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2), 1979.

[26] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. Answering top-k queries using views. In *Proceedings of 32nd International Conference on Very Large Data Bases, VLDB*, pages 451–462, 2006.

[27] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithm and Applications.* Springer-Verlag, 2008.

[28] M. Dell'Amico and S. Martello. Bounds for the cardinality constrainted $p||c_{max}$ problem. *Journal of Scheduling*, 4:123–138, 2001.

[29] Evangelos Dellis and Bernhard Seeger. Efficient computation of reverse skyline queries. In *Proceedings of 33rd International Conference on Very Large Data Bases, VLDB*, pages 291–302, 2007.

[30] Ke Deng, Xiaofang Zhou, and Heng Tao Shen. Multi-source skyline query processing in road networks. In *Proceedings of the 23rd IEEE International Conference on Data Engineering, ICDE*, pages 796–805, 2007.

[31] Jonathan Eckstein, Peter L Hammer, Ying Liu, Mikhail Nediak, and Bruno Simeone. The maximum box problem and its application to data analysis. *Computational Optimization and Applications*, 23(3):285–298, 2002.

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

[33] Craig S. Fleisher and Babette E. Bensoussan. *Business and Competitive Analysis: Effective Application of New and Classic Methods.* FT Press, 2007.

[34] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, New York, NY, 1979.

[35] Ronald L Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972.

[36] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 47–57, 1984.

[37] D. M. Hamby. A review of techniques for parameter sensitivity analysis of environment models. *Environmental Monitoring and Assessment*, 32(2):135–154, 1994.

[38] Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining: concepts and techniques.* Morgan Kaufmann, 2006.

[39] Gísli R. Hjaltason and Hanan Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, 1999.

[40] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4), 2008.

[41] Clement Kirabo Jackson and Elias Bruegmann. Teaching students and teaching each other: the importance of peer learning for teachers. *American Economic Journal: Applied Economics*, 1(4):85–108, 2009.

[42] C. Jin, K. Yi, L. Chen, J. X. Yu, and X. Lin. Sliding-window top-k queries on uncertain streams. In *Proceedings of the VLDB Endowment, PVLDB*, pages 301–312, 2008.

[43] Ibrahim Kamel and Christos Faloutsos. On packing r-trees. In *Proceedings of the 2nd ACM International Conference on Information and Knowledge Management, CIKM*, pages 490–499, 1993.

[44] N. Karmarkar and R. Karp. The differencing method of set partitioning. Technical Report UCB/CSD 82/113, University of California, Berkeley, CA, 1982.

[45] R. Korf. A complete anytime algorithm for number partitioning. *Artificial Intelligence*, 106(2):181–203, 1998.

[46] Flip Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 201–212, 2000.

[47] Ken C.K. Lee, Wang Chien Lee, and Hong Va Leong. Nearest surrounder queries. In *Proceedings of the 22nd IEEE International Conference on Data Engineering, ICDE*, pages 85–94, 2006.

[48] Scott T. Leutenegger, J. M. Edgington, and Mario A. Lopez. Str: A simple and efficient algorithm for r-tree packing. In *Proceedings of the 13th IEEE International Conference on Data Engineering, ICDE*, pages 497–506, 1997.

[49] Michael Levandowsky and David Winter. Distance between sets. *Nature*, 234:34–35, 1971.

[50] J Matousek and O. Schwarzkopf. Linear optimization queries. In *Proceedings of the ACM Symposium on Computational Geometry*, pages 16–25, 1992.

[51] Wil Michiels, Jan H. M. Korst, Emile H. L. Aarts, and Jan van Leeuwen. Performance ratios for the differencing method applied to the balanced number partitioning problem. In *Symposium on Theoretical Aspects of Computer Science, STACS*, pages 583–595, 2003.

[52] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 635–646, 2006.

[53] Kyriakos Mouratidis and HweeHwa Pang. Computing immutable regions for subspace top-k queries. In *Proceedings of the VLDB Endowment, PVLDB*, pages 73–84, 2013.

[54] I. Ntoutsi, K. Stefanidis, K. Norvag, and H. Kriegel. gRecs: A group recommendation system base on user user clustering. In *Proceedings of the 17th International Conference on Database Systems for Advanced Applications, DASFAA*, pages 299–303, 2012.

[55] Sarana Nutanong, Rui Zhang, Egement Tanin, and Lars Kulik. The v*-diagram: a query-dependent approach to moving knn queries. In *Proceedings of the VLDB Endowment, PVLDB*, pages 1095–1106, 2008.

[56] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proceedings of the 12th ACM Symposium on Principles of Database Systems, PODS*, pages 214–221, 1993.

[57] Dimitris Papadias and Yufei Tao. Reverse nearest neighbor query. In *Encyclopedia of Database Systems*, pages 2434–2438. 2009.

[58] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Transactions on Database Systems*, 30(1):41–82, 2005.

[59] Resnick Paul, Iacovou Neophytos, Suchak Mitesh, Bergstrom Peter, and Riedl John. GroupLens:an open architecture for collaborative filtering of netnews. In *Proceedings of ACM Conference on Computer Supported Cooperative Work, CSCW*, pages 175–186, 1994.

[60] Sunil Prabhakar, Yuni Xia, Dmitri V. Kalashnikov, Walid G. Aref, and Susanne E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 51(10):1124–1140, 2002.

[61] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. Osborne/McGraw-Hill, 2000.

[62] Paul Resnick and Hal R Varian. Recommender systems. *Communications of the ACM*, 40(3):56–58, 1997.

[63] John T Robinson. The KDB-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 10–18, 1981.

[64] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 71–79, 1995.

[65] Andrea Saltelli, Karen Chan, E Marian Scott, et al. *Sensitivity analysis*. Wiley New York, 2000.

[66] Andrea Saltelli, Marco Ratto, Terry Andres, Francesca Campolongo, Jessica Cariboni, Debora Gatelli, Michaela Saisana, and Stefano Tarantola. *Global sensitivity analysis: the primer*. John Wiley & Sons, 2008.

[67] Andrea Saltelli, Stefano Tarantola, and K.P.-S Chan. A quantitative model-independent method for global sensitivity analysis of model output. *Technometrics*, 41(1):39–56, 1999.

[68] Beomjoo Seo and Roger Zimmermann. Edge indexing in a grid for highly dynamic virtual environments. In *Proceedings of the 14th ACM International Conference on Multimedia*, pages 402–411, 2006.

[69] Mehdi Sharifzadeh, Cyrus Shahabi, and Leyla Kazemi. Processing spatial skyline queries in both vector spaces and spatial network databases. *ACM Transactions on Database Systems*, 34(3):14, 2009.

[70] Mohamed A. Soliman, Ihab F. Ilyas, Davide Martinenghi, and Marco Tagliasacchi. Ranking with uncertain scoring functions: semantics and sensitivity measures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 805–816, 2011.

[71] Rafael Sotelo, Yolanda Blanco-Fernandez, Martin Lopez-Nores, Alberto Gil-Solla, and Jose J Pazos-Arias. Tv program recommendation for groups based on muldimensional tv-anytime classfications. *IEEE Transactions on Consumer Electronics*, 55(1):248–256, 2009.

[72] Kian-Lee Tan, Pin-Kwang Eng, and Beng Chin Ooi. Efficient progressive skyline computation. In *Proceedings of 27th International Conference on Very Large Data Bases, VLDB*, pages 301–310, 2001.

[73] Yufei Tao, Vagelis Hristidis, Dimitris Papadias, and Yannis Papakonstantinou. Branch-and-bound processing of ranked queries. *Information Systems*, 32(3):424–445, 2007.

[74] L. Tasi. The modified differencing method for the set partitioning problem with cardinality constraints. *Discrete Applied Mathematics*, 63:175–180, 1995.

[75] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava. Ranked join indices. In *Proceedings of the 19th IEEE International Conference on Data Engineering, ICDE*, pages 277–288, 2003.

[76] Bala R. Vatti. A generic solution to polygon clipping. *Communications of the ACM*, 35(7):56–63, 1992.

[77] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Nørvåg. Reverse top-k queries. In *Proceedings of the 26th IEEE International Conference on Data Engineering, ICDE*, pages 365–376, 2010.

[78] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Nørvåg. Monochromatic and bichromatic reverse top-k queries. *IEEE Transactions on Knowledge and Data Engineering*, 23(8):1215–1229, 2011.

[79] Akrivi Vlachou, Christos Doulkeridis, Kjetil Norvag, and Yannis Kotidis. Branch-and-bound algorithm for reverse top-k queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 481–492, 2013.

[80] Min Xie, Laks V. S. Lakshmanan, and Peter T. Wood. Efficient top-k query answering using cached views. In *Proceedings of the ACM International Conference on Extending Database Technology, EDBT*, pages 489–500, 2013.

[81] B. Yakir. The differencing algorithm LDM for partitioning: A proof of a conjecture of Karmarkar and Karp. *Mathematics of Operations Research*, 21(1):85–99, 1996.

[82] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *Proceedings of the 19th IEEE International Conference on Data Engineering, ICDE*, pages 189–200, 2003.

[83] Zhenjie Zhang, Yin Yang, Ruichu Cai, Dimitris Papadias, and Anthony K. H. Tung. Kernel-based skyline cardinality estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD*, pages 509–522, 2009.