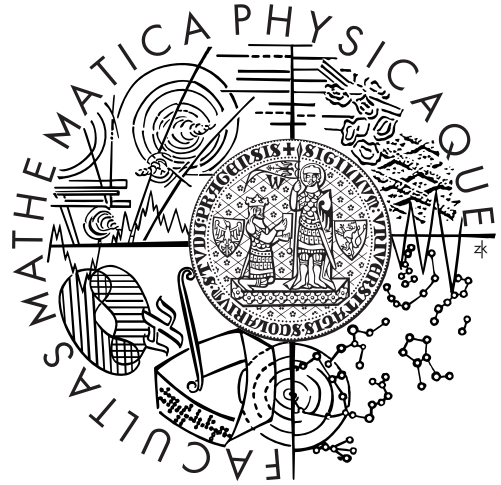
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



RNDr. Matúš Ondreička

Preference Top-k Search Based on Multidimensional B-tree

Department of Software Engineering
Supervisor: Prof. RNDr. Jaroslav Pokorný, CSc.

Study Programme: Computer Science
Specialization: Software Systems

March 2013

I would like to thank all those who supported me during my doctoral studies and I very much appreciate the help and advice received from my supervisor Prof. Jaroslav Pokorný. Furthermore, I would like to thank my nearest family for their support. Finally, I thank Aneta Mudrochová for her help with proofreading this thesis.

My thanks also go to the institutions that provided financial support for my research work. During my doctoral studies, my work was partially supported by the Grant Agency of Charles University (GAUK) number 9209, the grant SVV-2011-263312 and the grant SVV-2012-265312.

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, March 19, 2013

RNDr. Matúš Ondreička

Názov: Preferničné top-k vyhľadávanie založené na viacrozmernom B-strome
Autor: RNDr. Matúš Ondreička
Katedra: Katedra softwarového inženýrství
Matematicko-fyzikální fakulta
Univerzita Karlova v Praze
Školiteľ: Prof. RNDr. Jaroslav Pokorný, CSc.
Email autora: ondreicka@ksi.mff.cuni.cz
Email školiteľa: pokorny@ksi.mff.cuni.cz

Abstrakt: V tejto práci sa zameriavame na top-k vyhľadávanie podľa používateľských preferencií s použitím B^+-stromov a viacrozmerného B-stromu (MDB-strom). Používame model používateľských preferencií založený na fuzzy funkciách, ktorý nám umožňuje vyhľadávať podľa nemonotónnej ohodnocovacej funkcie. Navrhujeme model zotriedeného zoznamu založený na B^+-strome, ktorý umožní faginovym algoritmom vyhľadávať k najlepších objektov podľa nemonotónnej ohodnocovanej funkcie. Tento model používame v prostredí internetu s dátami na rôznych vzdialených serveroch. Okrem toho sme navrhli nové dynamické stromové štruktúry, konkrétne MDB-strom zložený z B^+-stromov, MDB-strom so zoznamami, MDB-strom so skupinami B^+-stromov a viacnásobne zoradený MDB-strom. Súčasne sme vyvinuli nové top-k algoritmy, konkrétne MD algoritmus, MXT algoritmus a ich varianty, ktoré dokážu vyhľadávať k najlepších objektov podľa nemonotónnej ohodnocovacej funkcie. Tieto top-k algoritmy sú efektívne, pretože dokážu nájsť k najlepších objektov bez potreby získania väčšiny objektov, ktoré sú uložené v navrhovaných stromových dátových štruktúrach.

Kľúcové slová: top-k vyhľadávanie, viacrozmerný B-strom, používateľské preferencie, nemonotónne ohodnocovanie, MD algoritmus, MXT algoritmus.

Title: Preference Top-k Search Based on Multidimensional B-Tree
Author: RNDr. Matúš Ondreička
Department: Department of Software Engineering
Faculty of Mathematics and Physics
Charles University in Prague
Supervisor: Prof. RNDr. Jaroslav Pokorný, CSc.
Author's e-mail address: ondreicka@ksi.mff.cuni.cz
Supervisor's e-mail address: pokorny@ksi.mff.cuni.cz

Abstract: In this thesis, we focus on the top-k search according to user preferences by using B^+-trees and the multidimensional B-tree (MDB-tree). We use model of user preferences based on fuzzy functions, which enable us to search according to a non-monotone ranking function. We propose model of sorted list based on the B^+-tree, which enables Fagin's algorithms to search for the top-k objects according to a non-monotone ranking function. We apply this model in the Internet environment with data on different remote servers. Furthermore, we designed novel dynamic tree-based data structures, namely, MDB-tree composed of B^+-trees, MDB-tree with lists, MDB-tree with groups of B^+-trees and multiple-ordered MDB-tree. Concurrently, we have developed novel top-k algorithms, namely, the MD algorithm, the MXT algorithm and their variants which are able search for the top-k objects according to a non-monotone ranking function. These top-k algorithms are efficient because they are able to find the top-k objects without the need to obtain the majority of the objects stored in the designed tree-based data structures.

Keywords: top-k search, multidimensional B-tree, user preferences, non-monotone ranking, MD algorithm, MXT algorithm.

Contents

1 Introduction **1**
- 1.1 Contribution . . . 5
- 1.2 Thesis organization . . . 6

2 Related Work **9**
- 2.1 Top-k search . . . 9
  - 2.1.1 Top-k search applications . . . 10
  - 2.1.2 Problem specification . . . 11
- 2.2 Research directions of top-k search . . . 12
  - 2.2.1 Top-k query processing in relational databases . . . 12
  - 2.2.2 Top-k search based on sorted lists . . . 15
  - 2.2.3 Top-k search in information retrieval . . . 18
  - 2.2.4 Top-k search over distributed data sources . . . 19
  - 2.2.5 Top-k search based on multidimensional indexes . . . 19
- 2.3 Ranking function in top-k search . . . 20
  - 2.3.1 Monotone ranking function . . . 21
  - 2.3.2 Non-monotone ranking function . . . 21
  - 2.3.3 No ranking function . . . 22

3 Model of User Preferences **23**
- 3.1 Preference modeling . . . 23
  - 3.1.1 Notation . . . 24
- 3.2 Ranking function . . . 24
- 3.3 Local preferences . . . 25
  - 3.3.1 Fuzzy logic perspective . . . 26
  - 3.3.2 Various attribute types . . . 27
  - 3.3.3 Approximation of local preferences . . . 28
- 3.4 Global preference . . . 29
  - 3.4.1 Substitution by local preferences . . . 30
  - 3.4.2 Monotone aggregate function . . . 31
  - 3.4.3 Non-monotone ranking function . . . 31

3.5 Utilization of the model . . . 33
3.5.1 Projection of user preferences . . . 33
3.5.2 Nearest neighbours and user preferences . . . 35
3.5.3 Range queries and user preferences . . . 37

4 Fagin's Algorithms 39
4.1 Prerequisites of Fagin's algorithms . . . 39
4.1.1 Sorted lists . . . 40
4.1.2 Sorted access and random access . . . 40
4.1.3 Monotone aggregate function . . . 41
4.2 Threshold algorithm . . . 42
4.3 No random access algorithm . . . 45
4.4 Combined algorithm . . . 49
4.5 Utilization of Fagin's algorithms . . . 51
4.5.1 Problem with support of local preferences . . . 51

5 Data Structures Based on B-trees 53
5.1 B-tree . . . 53
5.1.1 Redundant B-tree . . . 56
5.1.2 B^+-tree . . . 58
5.1.3 External associated data . . . 60
5.1.4 Object array . . . 60
5.2 Multidimensional indexing by reduction of dimensionality . . . 62
5.2.1 Multiple attribute tree . . . 62
5.2.2 Filial set . . . 64
5.3 Multidimensional B-tree . . . 64
5.3.1 Dynamic updates . . . 66
5.3.2 Associative queries . . . 68
5.4 Multidimensional B-tree composed of B^+-trees . . . 70
5.4.1 Degenerated multidimensional B-tree . . . 73
5.5 Multidimensional B-tree with lists . . . 73
5.5.1 Multidimensional B-tree with groups of B^+-trees . . . 75

6 Support of Local User Preferences Based on B^+-tree 77
6.1 Sorted list based on B^+-tree . . . 77
6.1.1 Abilities of B^+-tree . . . 78
6.1.2 Monotone local preference . . . 79
6.1.3 Arbitrary local preference . . . 80
6.2 Usage of B^+-trees in Fagin's algorithms . . . 85
6.3 Lists based on B^+-tree on remote servers . . . 89
6.3.1 Client-server architecture . . . 89

6.3.2 Stateless server . . . 90
6.3.3 Client application . . . 93
6.4 Experiments . . . 94
6.5 Summary . . . 96

7 Top-*k* Search Based on MDB-tree 97
7.1 Abilities of MDB-tree . . . 98
7.2 Top-*k* search in MDB-tree . . . 100
7.2.1 Depth-first search . . . 100
7.2.2 Advantages of TA algorithm . . . 102
7.2.3 Advantages of NRA algorithm . . . 103
7.2.4 Utilization of monotonicity . . . 106
7.3 MD algorithm . . . 107
7.3.1 Use of local preferences . . . 111
7.4 Experiments . . . 118
7.4.1 Distribution of attribute values . . . 118
7.4.2 Different domain sizes . . . 121
7.4.3 Various user preferences . . . 122
7.5 Multiple-ordered MDB-tree . . . 123
7.5.1 Ordering of attributes in MDB-tree . . . 124
7.5.2 Double-ordered MDB-tree . . . 124
7.5.3 Integration of all permutations of MDB-tree . . . 125
7.5.4 Use of multiple-ordered MDB-tree . . . 127
7.6 Summary . . . 128

8 Top-*k* Search Based on MDB-tree with Lists 129
8.1 Motivation . . . 130
8.2 Abilities of MDB-tree with lists . . . 130
8.3 Top-*k* search in MDB-tree with lists . . . 131
8.3.1 Use of MD algorithm . . . 131
8.3.2 Use of TA algorithm . . . 133
8.3.3 Integration of algorithms MD and TA . . . 134
8.4 MXT algorithm . . . 134
8.4.1 Use of local preferences . . . 139
8.5 Experiments . . . 142
8.5.1 Distribution of attribute values . . . 143
8.5.2 Real data set . . . 144
8.5.3 Various user preferences . . . 145
8.6 Summary . . . 147

9 Conclusion **149**
9.1 Contribution 150
9.2 Future work 151

Bibliography **153**

List of Algorithms

1 TA algorithm (Fagin's threshold algorithm) 43
2 NRA algorithm (Fagin's no random accesses algorithm) 48
3 CA algorithm (Fagin's combined algorithm) 50
4 BtreeList algorithm (list with support of local preference) . 83
5 NRAlp algorithm (NRA algorithm with local preferences) . . 87
6 TAlp algorithm (TA algorithm with local preferences) 88
7 BtreeListServer algorithm (B^+-tree as list on server) . . . 92
8 searchMDBtree (depth-first search of the MDB-tree) . . . 101
9 selectTopK (choosing of the top-k objects) 102
10 MD algorithm (the top-k search in the MDB-tree) 109
11 MDlp algorithm (MD algorithm with local preferences) . . . 114
12 MXT algorithm (the top-k search in MDB-tree with lists) . . . 136
13 MXTlp algorithm (MXT algorithm with local preferences) . . 141

Chapter 1

Introduction

In today's world, information systems of different types involve massive amounts of multidimensional data. A very frequent requirement is to develop effective techniques for search and retrieval of specific data stored in these systems. Oftentimes, for the users of such a system, it is interesting to find only a few of the best data which they are looking for.

Significant representatives of such information systems are full-text search engines, which search for a few of the most relevant web pages or documents with regard to some keywords or an expression. Another example is similarity search in multimedia databases or nearest neighbors search in metric spaces, where the task is to attempt to find a few of the most similar objects for a given object.

In this thesis, we perceive a multidimensional data set as a set of objects with more attributes. The search for the best k results, the top-k objects, in a multidimensional data set based on the ranking of the objects is often indicated as *top-k search* [Fag02, CG99] or in the context of relational database management systems it is indicated as *top-k query processing* [IBS08].

For the purpose of the top-k search, there is the need to use a suitable model of user preferences, which enables the user to express his/her preferences and makes it posssible to determine the top-k objects for the user. In this work, we use a model of user preferences based on composition of *local preferences* and *global preferences* [EPV07, GVP08, OP08]. By using this we accomplish the extension of commonly used preference models in top-k search, which are often based only on a monotone ranking function. The main advantage of our model of user preferences is that it enables us to use a *non-monotone ranking function* for the top-k search although the monotone aggregate function is required. We can obtain the non-monotone ranking function by the composition of local and global preferences because we do not suppose any restriction for the course of local preference functions.

In this thesis, we focus on the family of threshold algorithms, the most influential and significant top-k search algorithms developed by Fagin et al. [FLN03, Fag02, Fag99, Fag96], therefore we call them Fagin's algorithms. Fagin's algorithms are based on the principle that data are stored in sorted lists, which can be accessed by different access modes, i.e., the random access or the sorted access. The main advantage of Fagin's algorithms is that they are able to find the top-k objects according to a monotone aggregate function without accessing all objects. The main disadvantage of Fagin's algorithms is that they do not support local user preferences directly.

Therefore, we focus on tree-based data structures related to the B-tree [BM72], which can be utilized for efficient processing of the top-k search. First, we focus on the B-tree and its variants, which are the fundamental building blocks of our research, namely, the B^+-tree [Com79].

We introduce a new *model of a sorted list based on the B^+-tree*, by which it is possible to obtain the objects sequentially in descending order according to an arbitrary fuzzy function representing a user local preference for an attribute of the objects [OP08, HOP11]. This model enables Fagin's algorithms to access a set of objects with regard to user's local preference for an attribute. The most important advantage of this model is that the proposed B^+-trees as data sources used by Fagin's algorithms are independent of any user preferences and common for all users. We introduce the TAlp algorithm and the NRAlp algorithm as an extension of the TA algorithm and the NRA algorithm, which use the B^+-trees instead of sorted lists and allows the user to find the top-k objects according to his/her both local and global preferences, i.e., according to a non-monotone ranking function.

Moreover, we focus on the use of this new model of sorted list and Fagin's algorithms in Web environment, where the values of each attribute are stored on different remote servers on the Internet. We developed a system [HOP11], which is capable of accessing all these servers from a single location and finding the top-k objects with the support of local user preferences based on fuzzy functions efficiently.

Then we focus on multidimensional indexing based on reduction of dimensionality [GM80]. We present the *multidimensional B-tree*, which was developed by Scheuerman and Ouksel in [SO82]. The multidimensional B-tree, which we denote as MDB-tree, is composed of B-trees and enables a set of objects to be indexed according to more attributes simultaneously. In contrast to other approaches, we introduce a variant of the *MDB-tree composed of the B^+-trees* proposed by us in [OP08], which constitutes the base of our research. Moreover, we introduce the *MDB-tree with lists* as a new data structure, which we developed in [OP09, OP10]. In the MDB-tree with lists, some attribute values are stored like in the multidimensional B-tree and

others are stored like in Fagin's sorted lists. These newly proposed multidimensional tree-based data structures provide a dynamic environment for the top-k search and enable us to use some advanced models of user preferences based on user local preferences.

Therefore, we focus on the use of the MDB-tree, for the purposes of the top-k search. At the beginning of our research, there was an idea to apply Fagin's algorithms on the MDB-tree. It is not possible to apply them directly for this purpose because there are many obstacles which cannot be overcome. It was the main reason why we started developing new top-k algorithms using the MDB-tree, which would utilize useful features of Fagin's algorithms. We introduce the depth-first search of the MDB-tree, by which we can obtain all the objects stored in the MDB-tree. Then we focus on some useful advantages of Fagin's algorithms, which can be adopted and applied for the top-k search in the MDB-tree, and the utilization of monotonicity of the aggregate function. We developed a sequence of some statements with consequences and proofs. Finally, we provide the most important statement, by which it possible to skip some parts of the MDB-tree during the execution of the top-k search in the MDB-tree.

We developed the *MD algorithm* [OP08], which can search for the top-k objects in the MDB-tree according to a monotone aggregate function. The MD algorithm is an efficient top-k algorithm because it is searching only in some parts of the MDB-tree and thus without obtaining the greater part of the objects stored in the MDB-tree. The MD algorithm is efficient because it only searches in some parts of the MDB-tree. Thus, the MD algorithm does not need to obtain the majority of the objects stored in the MDB-tree.

Moreover, we focus on the top-k search in the MDB-tree according to the model of user preferences based on the local preferences, which enable us to use a non-monotone ranking function. Based on this assumption, we propose to use the variant of the MDB-tree, which is composed of the B^+-trees. Thus, we can apply the model of sorted list based on B^+-tree. Then it is possible to apply the local preferences directly during the computation of the top-k search in the MDB-tree. We developed the MDLP algorithm as an extension of the MD algorithm, which is able to search for the top-k objects also by using the local preferences.

If we suppose a set of objects indexed in the MDB-tree and the objects have more attributes with big-sized actual domains, the MDB-tree can become degenerated [SO82], i.e., the MDB-tree is very irregular. In such a case, the use of the MDB-tree is not the best choice because the use of the MD algorithm for the top-k search is not efficient. The problem of the degenerated MDB-tree can be partially solved by using the MDB-tree with lists because it is just designed for a set of objects having more attributes with

big-sized actual attribute domains. Especially, in the case of real data sets, the occurrence of more such attributes is very common.

Therefore, we focus on the top-k search by using the MDB-tree with lists. We developed the *MXT algorithm* [OP09, OP10], which can search for the top-k objects in the MDB-tree with lists according to a monotone aggregate function. The MXT algorithm is based on an integration of the MD algorithm and the TA algorithm because it uses the MDB-tree with lists composed of the MDB-tree and Fagin's sorted lists. Briefly, the base of the MXT algorithm is constituted by the MD algorithm and each group of lists in the MDB-tree with lists is searched by a new instance of the TA algorithm.

Afterwards, we focus on the top-k search in the MDB-tree with lists according to the model of user preferences based on the local preferences, which enable us to use a non-monotone ranking function. Therefore, instead of the MDB-tree with lists, we propose to use the *MDB-tree with groups of B^+-trees*, which is composed solely of B^+-trees. In such a case, we can apply the model of sorted list based on B^+-tree, and thus it is possible to use the local preferences directly during the computation of the top-k search in the MDB-tree with groups of B^+-trees. We developed the MXTLP algorithm as an extension of the MXT algorithm, which is able to search for the top-k objects efficiently by using the local preferences.

Algorithms MD, MDLP, MXT and MXTLP are efficient because they do not need to obtain the greater part of the objects stored in corresponding data structure. Moreover, these algorithms use multidimensional tree-based data structures, which are independent of user preferences, and it is possible to update these data structures easily and quickly.

All the proposed top-k algorithms in this thesis were experimentally evaluated. We tested the influence of various data sets used and various settings of user preferences used on the efficiency of the top-k algorithms. We used real and artificial data sets. We showed that the top-k algorithms proposed by us achieve results comparable with the results of the other top-k algorithms proposed in related works. Moreover, in the majority of cases, our top-k algorithms have very good results, especially for a small number of top-k objects. This ability is convenient for the purpose of the top-k search.

Based on the experimental results, we found out that according to the properties of a set of objects and user preferences it is possible to choose the best variant of the MDB-tree or the MDB-tree with lists, and thus to use the most efficient top-k algorithm proposed in this thesis. Moreover, we propose the multiple-ordered MDB-tree as a new multidimensional index structure derived from the MDB-tree, which guarantees the most efficient execution of the MD algorithm according to the importance of particular attributes in the aggregate function.

1.1 Contribution

In this section, we briefly describe the contribution of this thesis, which was published in [OP08, OP09, Ond09, OP10, HOP11, On11a, On11b]. It can be split into three main contributions as follows.

- **Support of local user preferences based on B$^+$-tree**
  The first contribution was that we introduced the model of a sorted list based on the B$^+$-tree in detail, which enables Fagin's algorithms to search for the top-k objects according to the non-monotone ranking function. Initially, we adopted the concept of similarity search from Fagin et. al [FKS03] and the approach from [EPV07, GVP08]. Subsequently, in this thesis and in [OP08, OP09, OP10, HOP11], we redefined the theoretical background of the model. Thus, we were able to apply Fagin's NRA algorithm using the model in Web environment [HOP11], where the values of each attribute are stored on different remote servers on the Internet and, moreover, each server can be stateless, i.e. independent of any previous request.

- **Top-k search based on multidimensional B-tree**
  The second contribution was that we applied the multidimensional B-tree in the top-k search. We developed the MD algorithm as a new efficient top-k algorithm, which is able to find the top-k objects in the MDB-tree according to a monotone ranking function [OP08]. Moreover, we developed the MDLP algorithm as a variant of the MD algorithm which is able to find the top-k objects in the MDB-tree composed of B$^+$-trees according to a non-monotone ranking function [OP08, OP10]. In addition, we propose the multiple-ordered MDB-tree as a new multidimensional index structure, which guarantees the most efficient execution of the MD algorithm and the MDLP algorithm.

- **Top-k search based on multidimensional B-tree with lists**
  The third contribution was that we developed the MDB-tree with lists as a new multidimensional tree-based data structure and the MXT algorithm as a new efficient top-k algorithm, which is able to find the top-k objects in the MDB-tree with lists according to a monotone ranking function [OP09, OP10]. Moreover, we developed the MDB-tree with groups of B$^+$-trees as a new multidimensional tree-based data structure. In addition, developed the MXTLP algorithm as a variant of the MD algorithm which is able to find the top-k objects in the MDB-tree with groups of B$^+$-trees according to a non-monotone ranking function [OP09, OP10].

1.2 Thesis organization

The rest of this thesis is organized as follows.

Chapter 2 is dedicated to the related work and to a detailed explanation of the top-k search. First, Section 2.1 introduces the top-k search. We discuss top-k search applications and we specify the top-k search in the context of this thesis. Section 2.2 is devoted to introduce the main directions of research in the area of top-k search. In Section 2.2.1, we discuss top-k query processing in relational databases and we focus on various query models of the top-k search. Section 2.2.2 briefly explain top-k search based on sorted lists and we present the most significant top-k algorithms. In Section 2.2.3 and Section 2.2.4, we mention top-k search in information retrieval and top-k search over distributed data sources. In Section 2.3 we focus on top-k search based on multidimensional indexes. Finally, in Section 2.3 we discuss the use of the ranking function in the top-k search.

Chapter 3 deals with the model of user preferences used in this thesis. In Section 3.1 we focus on the problem of preference modeling. We introduce the ranking function in Section 3.2. Section 3.3 introduces the local user preferences, the fuzzy functions, classification of attribute types and approximation of local preferences, respectively. Section 3.4 presents the global user preference, the substitution by local preferences, monotone aggregate function and non-monotone aggregate function, respectively. Finally, we discuss the utilization of our preference model in Section 3.5.

Chapter 4 clarifies Fagin's algorithms as the fundamental top-k algorithms used for our research. Section 4.1 focuses on all the prerequisites of Fagin's algorithms, namely, the random access mode and the sorted access mode. According to utilization of the access modes, Fagin et al. [FLN03] define more variants of top-k search algorithms, namely, threshold algorithm (the TA algorithm), no random access algorithm (the NRA algorithm) and combined algorithm (the CA algorithm), described in Section 4.2, Section 4.3 and Section 4.4, respectively. In Section 4.5, we discuss the utilization of Fagin's algorithms with respect to different access modes. In Section 4.5.1, we also provide a discussion about disadvantages of Fagin's algorithms in the context of the issues discussed in this thesis.

Chapter 5 presents data structures utilized and developed in this thesis. In Section 5.1, we provide a review of the B-tree, the redundant B-tree, and the B^+-tree. In Section 5.2, we focus on multidimensional indexing based on reduction of the dimensionality. Section 5.3 focuses on the multidimensional B-tree in detail. In Section 5.4, we provide the multidimensional B-tree composed of the B^+-trees. Finally, Section 5.5 introduces the multidimensional B-tree with lists, which partially solves the problem of placement of

the indexed attributes in the multidimensional B-tree.

Chapter 6 introduces the support of local user preferences based on the B^+-tree in detail. In Section 6.1, we discuss the abilities of a B^+-tree and explain the principle of the sequential access into the B^+-tree according to a fuzzy function. In Section 6.2, we discuss the application of the new model of the sorted list in Fagin's algorithms, which makes it possible to apply the local user preferences directly during the computation of Fagin's algorithms. In Section 6.3, we focus on the use of this new model of the sorted list and Fagin's algorithms in Web environment, where the values of each attribute are stored on different remote servers on the Internet. Section 6.4 shows experimental results. In Section 6.5 provides a summary.

Chapter 7 is devoted to explain the top-k search based on the use of the MDB-Tree. First, Section 7.1 focuses on the abilities of the MDB-tree, which helps us to develop the MD algorithm. In Section 7.2, we focus on the top-k search in the MDB-tree. We introduce the depth-first search of the MDB-tree, by which we can obtain all the objects stored in the MDB-tree. Then, we focus on some useful advantages of Fagin's algorithms, which can be adopted and applied for the top-k search in the MDB-tree. We present a sequence of some statements with consequences and proofs. In Section 7.2.4, we focus on utilization of the monotonicity of the aggregate function and we provide the most important statement, by which it possible to skip some parts of the MDB-tree during the execution of the top-k search in the MDB-tree. Section 7.3 presents the MD algorithm and the MDLP algorithm using the local preferences. In Section 7.4 we show experimental results of presented algorithms, which achieve better results in the number of accessed objects than Fagin's algorithms. In Section 7.5, we present *the multiple-ordered MDB-tree* as an advantageous index structure derived from the MDB-tree. Finally, in Section 7.6, we provide a summary and an analysis of the efficiency of the MD algorithm.

Chapter 8 is devoted to explain the top-k search based on use of the MDB-Tree with list. First, in Section 8.1, we explain why we use the MDB-tree with lists for the top-k search. In Section 8.2, we focus on the abilities the MDB-tree with lists, which help us to develop the MXT algorithm. In Section 8.3, we discuss how it is possible to use the MD algorithm, the TA algorithm and their mutual integration during the execution of the top-k search in the MDB-tree. Section 8.4 presents the MXT algorithm and the MXTLP algorithm using the local preferences. Finally, Section 8.5 shows experimental results of presented algorithms. In Section 8.6, we provide a summary and an analysis of the efficiency of the MXT algorithm.

Chapter 9 concludes this thesis, Section 9.1 summarizes the main contributions of this thesis. Finally, Section 9.2 provides some suggestions for the

future research and further improvements based on ideas presented in this thesis.

Chapter 2

Related Work

In this chapter, we discuss the related work in the area of top-k search. Several intresting approaches have been proposed in the last decade. Nowadays, the top-k search constitutes the fundamental building block in many applications. First, we attempt to specify the top-k search in the context of this thesis. We provide a brief survey of the main directions of research in the area of top-k search. Concurrently, we provide an overview of the most significant top-k algorithms. We also discuss the top-k search in the context of multidimensional indexes. Finally, we discuss the use of non-monotone ranking functions in the top-k search.

In Section 2.1, we introduce the top-k search. We discuss top-k search applications and we specify the top-k search in context of this thesis. Section 2.2 is introducing the main directions of research in the area of top-k search. In Section 2.2.1, we discuss top-k query processing in relational databases and we focus on various query models of the top-k search. In Section 2.2.2, we briefly explain the top-k search based on sorted lists and we present the most significant top-k algorithms. In Section 2.2.3 and Section 2.2.4, we mention the top-k search in information retrieval and top-k search over distributed data sources. In Section 2.3 we focus on the top-k search based on multidimensional indexes. Finally, in Section 2.3 we discuss the use of the ranking function in the top-k search.

2.1 Top-k search

Because of a rising amount of multidimensional data, effective search for the few best results in the huge amount of multidimensional data is an important research question. This is why the *top-k search* [Fag02, CG99], i.e., top-k query processing [IBS08], has become a special research topic.

In the last decade, there has been a great amount of discoveries in the area of top-k search. Many top-k algorithms using various data sources have been proposed. The research in the area of top-k search is in progress in various domains such as relational databases [IBS08], XML [MAK05], multimedia search [CGM04], the Web [BGM02], or distributed systems [MTW05].

2.1.1 Top-k search applications

The top-k search on multidimensional data sets is perceived as the search for the most k relevant or interesting results according to some requirement. Nowadays, the top-k search constitutes the fundamental building block in many applications, e.g.,

- Full-text search engines, which search for the few most relevant web pages or documents with regard to some keywords or an expression.
- Rank queries in relational database management systems.
- Multimedia databases, where for a given image, music or video, users attempt to find a few of the few most similar objects.
- Ranking in information retrieval, where users search for the few most relevant documents in document databases, semi-structured databases or XML.
- Product catalogues, where users search for the k most relevant objects with regard to their preferences.

Regardless of particular realization of the top-k search, the most important factor has always been the speed of computation of the top-k results. This factor depends on many aspects or characteristics of a particular solution and their mutual relation. The most crucial aspect is the impact of the data source used and the cost or time of the access method to the particular data items. We can use as the data source a relational database management system, sorted lists (inverted files), a multidimensional index or data can be distributed on many remote locations. Moreover, this data can be accessed in different modes, which can differ in their access cost.

Generally speaking, there are two means of speeding up the top-k search. One is using solutions based on a suitable model of data source and a suitable top-k algorithm, which can realize the top-k search in this data source efficiently. Many of these solutions have a very sophisticated theoretical

background and constitute the foundation of top-k search. The most important variants of these solutions of top-k search will be discussed more in Section 2.2.

The other possibility of speeding up the top-k search is using various improvements and optimizations of well-known top-k algorithms from the programmer's perspective. These enhancements are highly dependent on a particular variant of top-k algorithm and therefore we do not need to discuss them in this chapter.

2.1.2 Problem specification

In this thesis, we focus on some novel solutions to search for the top-k objects efficiently. Therefore, in this section, we attempt to specify the top-k search. In the rest of this thesis we perceive the top-k search as it is specified in the following.

We suppose that a multidimensional data set is a set of objects with m attributes, which includes n different objects. Each object has m values of these m attributes or, in terminology of some related works, m local scores. Incidentally, there are some solutions, which deal with uncertain data (see Section 2.2.2), in which all the attribute values do not have to be known. In the rest of the text we presume that each object has all the local scores known, if not explicitly stated otherwise.

In general, it is possible to find the top-k objects, if a ranking of the objects from the best to the worst exists. Each object from multidimensional data set is determined by its values of attributes in the multidimensional data space. For the purposes of top-k search, it is common to use a ranking function (aggregate function), which combines the values from multiple attributes into the score or the ranking. Thereafter each object has its score and it is possible to rank the objects from the best to the worst. Then the top-k objects can be clearly determined.

A trivial solution to the problem of finding the top-k objects provides an exhaustive search. The exhaustive search needs to load all relevant objects together with the values of their attributes from the multidimensional data set, to evaluate every object's score according to a ranking function, and finally to select k objects with the highest score.

The exhaustive search is not efficient because all the objects with their attribute values have to be obtained from the data source. This can be difficult, especially for a large data set of objects or in the case of distributed data sources. Therefore, the meaningful top-k algorithms are able to find top-k results without searching the whole data set, i.e., without loading all objects with their attribute values from data sources.

In the case of the exhaustive search, it is necessary to load n objects from the data source. If we suppose that object attribute values are separated in the data source, obtaining one attribute value of one object can be perceived as *one access* into the data source. In these terms, the exhaustive search has to perform $n \cdot m$ accesses into the data source. In this sense, we perceive a top-k algorithm as *efficient*, if it is able to find top-k objects by performing far fewer accesses into the data source than $n \cdot m$ accesses.

2.2 Research directions of top-k search

The most comprehensive survey of the top-k search has been carried out in [IBS08]. The main disadvantage of this survey is that the authors focus mainly on the top-k search, i.e. top-k query processing, in context of the relational database management systems.

In general, we can distinguish two main research directions in the area of top-k search. The first direction is an extension or adaptation of top-k query processing methods in the relational database management system (RDBMS), which enables to execute the top-k queries efficiently. The second direction is to use some suitable data sources and top-k algorithms, which are not limited by the construction characteristic of the RDBMS.

At first sight, these two directions are very different, but they are significantly interconnected. The last ten years have many times shown that ideas from the first direction were adopted by researchers of the second direction and vice versa.

2.2.1 Top-k query processing in relational databases

At the end of 1990's, research groups around the world were trying to solve a new research problem in RDBMS, which was how to find the best k answers for a query despite the result for the classical queries based on the selection of the results which exactly satisfy the query. This approach was not sufficient for the top-k queries because the result can be an empty-set or a too-large-set of objects. Nowdays, a query has begun to be assumed as a preference of a user, according to which the user wants to find the top-k answers for him/her. For the purpose of top-k search, it is necessary to use a ranking function, which provides the user preference into the query.

For example, the following SQL query can be assumed as a top-k selection query, where ranking function $R(A_1, ..., A_m)$ expresses a user preference. The query returns the top-k objects according to the ranking function with all their attribute values and their scores.

```
SELECT A_1, ..., A_m, R(A_1, ..., A_m) as ranking
FROM set_of_objects
ORDER BY ranking DESC
LIMIT k
```

The execution of such a query is not naturally optimized in classical RDBMS. In general, the execution is based on the exhaustive search of all the records and the evaluation of the score for each record. Finally, top-k answers are produced only by the limitation to the first k of the query results.

The main problem of searching for the top-k answers in RDBMS is the computation complexity, especially for large data sets and some complicated query models. In [IBS08], the authors classify the top-k search algorithms in RDBMS according to an assumed query model. There are three basic query models, i.e., top-k selection, top-k join and top-k aggregate query model.

Top-k selection query model

In the top-k selection query model, all input sources, i.e., a set of relations in RDBMS, include information about the same set of objects. The most common case is that each input source, i.e., each relation in RDBMS, contains the same set of objects, but with the values of one attribute. In other words, we perceive all the input sources together as a set of objects, i.e., a relation in RDBMS, where each object has all its attribute values. In this case, a top-k algorithm searches for the top-k objects with the highest k scores from all the input sources with regard to a ranking function.

Most of the proposed approaches use top-k selection query model, i.e., Fagin's algorithms [FLN03, Fag02, Fag99, Fag96] or the top-k algorithm proposed in [NR99, GBK00]. In the context of RDBMS, one of the earliest approaches using this model was proposed by Chaudhuri and Gravano in [CG99] and further extended in [BCG02, CGM04]. In this thesis, we use the top-k selection query model.

Top-k join query model

The top-k join query model is also known as the *rank-join* query model. In this model, each input source can contain a different set of objects. In this case, the top-k algorithm uses a join condition among the objects in different input sources and joins them into a result, which contains the top-k items with the highest k scores from all the input sources with regard to a ranking function.

The earliest top-k algorithms dealing with top-k join query model were the J* algorithm developed by Natsev et al. in [NCS01] and the NRA-RJ

algorithm developed by Ilyas et al. in [IAE02], derived from Fagin's NRA algorithm. Later, Ilyas et al. developed the Rank-Join algorithm [IAE04] and the Hash Rank Join Operator (HRJN) based on symmetrical hash join as two-way hash join implementation of the Rank-Join algorithm. Moreover, Ilyas et al. developed the RankSQL system [LCI05] as the first algebraic framework for the integration of rank-aggregation operators in RDBMS, which supports efficient evaluations of the top-k queries in RDBMS.

Top-k aggregate query model

In the top-k aggregate query model, scores are computed for groups of records, rather than individual records. A top-k aggregate query reports the top-k groups with the highest k scores. The scores of groups are computed using a group aggregate function such as sum. Naturally, the processing of a top-k aggregate query is more complex than the processing of a top-k join query. For example, an approach to compute top-k aggregate queries efficiently is introduced in [LCI06], where the authors present a framework for efficient processing of ad-hoc top-k (ranking) aggregate queries, which is able to find the k groups with the highest k values of a ranking function as results.

Materialized views

Analogously to the traditional query processing in RDBMS, the use of materialized views can improve the efficiency of the top-k search approaches. The top-k query processing can also benefit from the existence of materialized results of previously executed top-k queries.

Das et al. proposed the LPTA algorithm [DGK06], i.e., linear programming based on Fagin's TA algorithm, which deals with materialized views. The LPTA algorithm reads the records sequentially from materialized views, which were built using the answers of previous queries, and evaluates the scores of obtained records according to a linear (monotone) ranking function. The idea of the LPTA algorithm is to choose an optimal subset among all available materialized views to minimize the number of obtained records.

Another approach is the PREFER system [HKP01]. The PREFER system utilizes a number of materialized views, which are preprocessed and stored, based on different weight assignments of the attribute values. For a given monotone ranking function, the PREFER system searches for the top-k objects by using a few of these preprocessed materialized views. The disadvantage of this approach is the rising number of materialized views with regard to the number of attributes, i.e., dimensions.

2.2.2 Top-k search based on sorted lists

The fundamental family of top-k algorithms is a family of threshold algorithms. The main representatives are *threshold algorithm* (TA), *no random access algorithm* (NRA) and *combined algorithm* (CA), which were developed by Ronald Fagin et. al [FLN03, Fag02, Fag99, Fag96] at the IBM Almaden Research Center. Therefore, we call them Fagin's algorithms. These algorithms have also been independently proposed by other research groups, i.e. Nepal et al. [NR99] and Güntzer et al. [GBK00]. Moreover, according to [The06], a variant of these algorithms has already been proposed by Buckley [BL85] and Pfeifer et al. [PF95]. In addition to the other authors, Fagin et al. for the first time provided the most comprehensive study and demonstrated that these algorithms are "instance optimal" in class of databases based on sorted lists [Fag02].

Fagin's algorithms are the most influential and significant top-k search algorithms. Since Fagin's algorithms constitute the base of our research, Chapter 4 of this thesis is devoted to explain them in detail because we will refer to their advantages in further chapters. In this section, we briefly sketch the main principles of Fagin's algorithms and we discuss them in the context of related works.

Data model based on sorted lists

The family of threshold algorithms uses top-k selection query model, and the data, i.e. a set of objects, is stored in sorted lists. If we suppose that the objects have m attributes, they are stored in m sorted lists. Each sorted list includes all the objects with values of one attribute as object-value pairs. The object-value pairs in each list are sorted in descending order according to the attribute values included.

The principle of the family of threshold algorithms is naturally logical. They suppose that the objects with the higher attribute values have the biggest potential to be the top-k objects. Fagin's algorithms primarily focus on such objects situated on the top of the sorted lists. Therefore these algorithms attempt to access these objects first and to investigate whether they can or cannot be the ones of the top-k objects as soon as possible.

The main advantage of the family of threshold algorithms is that they are capable of finding top-k objects much sooner than they read the whole lists, i.e., obtain all the objects. During their computation, they evaluate a threshold value or a crucial condition, the use of which enables the algorithms to stop as soon as possible (see Chapter 4).

Access modes to data

In general, if we assume a data set composed of data items, we distinguish two fundamental modes how to access data items. The first mode is when data items are stored in some sort of a list which can be accessed sequentially, and data items can be obtained one after another without the need of any extra maintenance. We call this access mode the *sorted access*. The second access mode is when according to a key value, it is necessary to access a random data item. We call this access mode the *random access*.

Naturally, the execution cost of random access is usually greater than the execution cost of sorted access because there is the need for some extra maintenance such as searching of an additional index. A typical example occurs when the costs correspond to disk access. Moreover, the use of one of the access modes can be restricted or absolutely impossible in some real applications.

Use of the access modes

The top-k algorithms can be classified according to the access modes which they use. Fagin et al. [FLN03, Fag02, Fag99, Fag96] define three basic variants of top-k search algorithms according to utilization of the sorted access and the random access (see Chapter 4). Fagin's TA algorithm uses both these access modes. Fagin's NRA algorithm only uses the sorted access. Fagin's CA algorithm is a merger between the NRA algorithm and the TA algorithm which uses both these access modes. The authors in [FLN03] consider that there is the need to distinguish between the cost of the random access and the cost of the sorted access (see Section 4.5). Therefore, additionally to algorithms TA an NRA, the CA algorithm deals with different execution cost of access modes and tries to minimize the overall cost of computation of the top-k objects. Fagin [Fag02] defines the cost ratio, which drives the execution of random accesses in the CA algorithm.

When the use of the random access is too expensive or even impossible, the NRA algorithm or the *Stream-Combine* [GBK00] can be used. These algorithms are based on nearly the same concept. In contrast to the TA algorithm, these algorithms cannot obtain all attribute values of an object immediately because they do not use the random access. Therefore they cannot evaluate the score of objects obtained from a list (see Section 4.3). Therefore these algorithms have to use some score estimation methods. While the NRA algorithm uses upper and lower score estimations, the Stream-Combine algorithm only uses the upper score estimation. Thus, the NRA algorithm is able to stop much earlier than the Stream-Combine algorithm.

Also the sorted access can be too expensive or even impossible, and thus data sources can be accessed only by the random access. There are some approaches [CH02, BGM02], which consider unsorted data sources and attempt to integrate these sources into the TA algorithm.

Access scheduling

Fagin's algorithms TA and NRA perform the sorted access into all the sorted lists in each step of their computation. If we can perceive the step as the sorted access in each list separately, the algorithms perform the sorted access into the sorted lists in round-robin style, i.e., in one round the sorted access is performed in each sorted list (see Section 4.1.2). Round-robin style is one of the simplest scheduling methods. But we can perform the sorted access by a more sophisticated scheduling method because in each step it is possible to decide in which list the next sorted access will be performed.

Many related works propose improvements denoted as *access scheduling*, which can significantly speed up the run time of Fagin's algorithms. Namely, Güntzer et al. [GBK00] propose the *Quick-Combine* algorithm, which is based on nearly the same concept as the TA algorithm. The *Quick-Combine* algorithm observes the effectiveness of the reading from each of the lists with regard to speed of the algorithm and selects the most suitable list for the execution of the next sorted access.

In [GLV05, GV08], the authors propose some heuristics as an extension of access scheduling of the TA algorithm and the NRA algorithm. The use of this heuristics can decrease the number of sorted accesses performed to find the top-k objects.

Moreover, it is possible to schedule when to perform the random access. Bast et al. [BMT06] focus on the prioritization of different sorted lists in the sorted accesses, and the decision when to perform random accesses and for which candidates, i.e., some temporally kept objects which still have a chance to become top-k objects. The authors propose that both access modes involve highly specialized probabilistic cost models, according to which it is possible to drive the prioritization of the sorted lists. Thus, the performance of the top-k search can be improved with no loss of result quality.

Uncertainty of results

The family of threshold algorithms [FLN01] provides the objects with the highest k rankings as an exact result of the top-k search. Some top-k algorithms are based on the uncertainty involved in the results, which can speed up the search for the top-k objects.

One research direction is constituted by approximation algorithms, which have the ability to stop early and produce a set of results that are close to the exact top-k result. For example, Fagin et al. [FLN03] proposed that Fagin's TA algorithm can be easily modified to be an approximation algorithm. The authors provided in [CDN01] an approximate top-k search solution, which recognizes the importance of taking into account the variance in the data distribution in RDBMS.

Another research direction is constituted by probabilistic algorithms. The support of approximate results also requires building probabilistic models to fit the score distributions in data source. The main representative of the probabilistic solution to the top-k search in the context of information retrieval provided by Theobald et al. [TSW04], which was used as an improvement of the TopX system [The06, TSW05, TBM08].

Another solution is provided by the authors in [CDH04], where they adapt and apply principles of probabilistic models from information retrieval for structured data. Moreover, there are some approaches [ARS03, RDS07], which deal with the uncertainty involved in the data. These approaches are also based on the probabilistic models.

In this thesis, we focus on the top-k algorithms which provide the exact result of the top-k search. The application of these approximate and probabilistic approaches should be the next interesting direction of our future research, which can speed up our top-k algorithms and save the number of accesses into the data sources performed by them.

2.2.3 Top-k search in information retrieval

Information Retrieval (IR) focuses on finding relevant documents that satisfy user requirements. In many application domains, such as full-text search engines, the number of relevant documents for a term-based query can be huge, but the users find the top-k documents. The top-k search in IR deals with a set of terms, which can occur in a set of documents. For example, the biggest full-text search engines can include several billions of documents and they can distinguish several thousands of terms.

In the context of our specification of the top-k search (see Section 2.1.2), we can perceive a set of documents as the set of objects and a set of terms as the attributes. The most common model in IR is that the value of a term for a document is perceived as frequency of occurrence of the term in the document, e.g. PageRank [BP98].

Using these assumptions, it is possible to use the top-k selection query model and thus the model of sorted lists. In this case, there is one sorted list for each term, which includes all the objects with the frequency of occurrence

of the term. For example, if we choose the query as a set of several terms, then we can use a top-k algorithm from the family of threshold algorithms which find the top-k documents in the corresponding lists.

Based on the context, it is clear that IR provides many opportunities to improve or to apply the existing top-k algorithms. One of the most interesting contributions in the area of top-k search is provided by the TopX system developed by Theobald et al. [The06, TSW05, TBM08]. The TopX system is an efficient and versatile top-k query engine for text and semi-structured data like XML. The TopX system includes several significant improvements of Fagin's CA algorithm, namely, efficient access scheduling [BMT06] or efficient probabilistic approaches [TSW04].

2.2.4 Top-k search over distributed data sources

The top-k search over distributed data sources is an important problem especially in the context of information retrieval of document databases distributed over P2P networks. To the best of our knowledge, there are not too many approaches which deal with the top-k search in the environment of distributed data sources.

The authors of [MTW05] developed the KLEE framework, which is the most significant contribution in this area. The KLEE framework provides efficient processing of top-k queries in wide-area distributed data repositories. The framework tries to minimize the computational cost, which includes network latency and local peer work. The KLEE framework uses approximate algorithms [TSW04] and searches for the best k documents but only according to a monotone aggregation function.

Therefore, we focus on the top-k search according to a non-monotone ranking function in Section 6.3. Briefly, we concentrate on the use of Fagin's algorithms using a non-monotone ranking function in Web environment, where the values of each attribute are stored on different remote servers.

2.2.5 Top-k search based on multidimensional indexes

The most commonly used multidimensional index for searching in the multidimensional data sets is based on an R-tree [Gut84]. The R-tree is designed especially for multidimensional range queries and the similarity search.

The k-nearest neighbours search, also known as the similarity search, is a search for the k closest points in metric spaces. We percieve the k-nearest neighbours search as a special case of the top-k search, where instead of the ranking function a similarity measure between the query object and other objects is used.

Many algorithms in the literature [ARS03, BBK01, CP00, HS99, HS03] include nearest neighbour search methods which use a multidimensional index based on the R-tree.

Many algorithms in the literature [ARS03, BBK01, CP00, HS99, HS03] include nearest neighbour search methods which use a multidimensional index based on the R-tree. The idea of using the R-tree for the top-k search was for the first time presented in [TPK03], where the authors compare the top-k search over the R-tree and over their new index called ranked join index. The authors only consider a simple weighted summation as the ranking function.

Another example is using the M-tree (metric tree) [CPZ97] and the NM-tree (non-metric tree) [SL08] for the similarity search. Skopal et. al [SL08, Sko06] proposed methods for the nearest neighbour search based on the M-tree and the NM-tree for approximate similarity search in metric and non-metric spaces.

Another interesting approach based on arbitrary similarity measures, was introduced in [DPK08], where the author describe an on-line top-k algorithm. Their algorithm is based on the TA algorithm and uses a AL-tree [DPK08] as an multidimensional index. represent the data with appropriate attribute values. The similarities between various values of each attribute are precomputed. Their on-line top-k algorithm is able to return the top-k objects similar to a query object using the AL-tree.

The most closely related works for us are [SG11] and [SG12] because the authors use the model of user preferences based on non-monotone ranking functions [EPV07, GVP08], which we also use for the top-k search (see Section 3.4). The authors in [SG11] introduce the top-k search over the R-tree with using user preferences. For searching top-k objects they adapted algorithm incremental nearest neighbour [HS99] commonly used in the k-nearest neighbours search. The authors in [SG12] introduce the top-k search over the grid file [NHS84]. The grid file does not require computation with all the attribute values together in the case of the R-tree nor the M-tree. The grid file can be used directly for indexing any type of attributes with natural ordering. The authors showed that the grid file offer much better performance of the top-k search than separated B^+-trees or the exhaustive search.

2.3 Ranking function in top-k search

Top-k algorithms can be classified based on the restrictions of the ranking function, i.e., the aggregate function, according to which they search for the top-k objects.

2.3.1 Monotone ranking function

The mainstream of top-k algorithms [FLN03, CG99, CGM04, NR99, HKP01] have a crucial restriction that they are able to search for top-k objects only according to a monotone aggregate function (see Section 3.4.2). The monotonicity of the ranking function is a reasonable property expected of an aggregate function, but ranking functions do not necessarily have to be monotone.

2.3.2 Non-monotone ranking function

One of the challenges in the top-k search is to use a generic ranking function, i.e. the arbitrary ranking function which can be non-monotone (see Section 3.4.3). Therefore, in this thesis, we focus on the top-k search using a model of user preferences based on the composition of local preferences and global preferences [EPV07, GVP08], which enables us to search for the top-k objects according to a non-monotone ranking function.

Fagin et al. focused on the top-k search according to a non-monotone ranking function in [FKS03], where they propose an approach to performing efficient similarity search in high dimensional data by using the sorted lists. The authors described the L2TA algorithm, which is an application of the TA algorithm to the problem of computing Euclidean k-nearest neighbours. This algorithm can be used in place of the naive k-nearest neighbours algorithm, and is often much faster. Compared to this approach, our approach based on local user preferences is more versatile (see Section 6.2).

The most closely related works for us are [SG11, SG12] because the authors use the same model of user preferences, i.e., a non-monotone ranking function. Compared to our approach based on use of the multidimensional B-tree, their approach focuses on the use of the R-tree and the grid file, respectively, for the top-k search (see Section 2.2.5). Note that the model of user preferences, consisting of fuzzy functions, one for each attribute, and a monotone aggregate function, thus approaches based on the model have a higher expressive power.

Moreover, there are interesting approaches [ZHC06, XHC07] which are able to find top-k objects according to a non-monotone ranking function in relational database systems. Both these solutions utilize structures based on B-tree during their computation. Thus, they provide a motivation for us to focus just on the data structures based on B-tree (see Chapter 5).

The authors of [ZHC06] present the OPT* algorithm, which is based on idea, that searching for the top-k objects is perceived as an optimization problem. The problem of the top-k search is transformed into a shortest path problem, where the optimization goal function consists of a boolean

expression. This solution assumes that objects are indexed in B-trees, where for each attribute there is one B-tree in which all the objects are indexed by values of corresponding attribute. The optimization function filters objects according to required attribute values, and it is able to search for the top-k object according to a generic ranking function.

Xin et al. in [XHC07] deal with the analysis of the aggregation function by using mathematical methods. The authors presented an index-merge framework that performs a progressive search over a space of states composed by joining index nodes. The main idea of this approach is is to utilize the existing B-tree and R-tree indexes of ranking predicates to construct a search space of possible top-k results. A state in this space is composed by joining multiple index nodes. According to the authors of this approaches, it is possible to find the top-k objects in an effective way, if the analysis of the given ranking function over any domain sub-region can find the maximum and possibly recognize monotonicity of the ranking function. This analysis can be rather difficult for an arbitrary function.

2.3.3 No ranking function

In some cases, it is not straightforward to define a ranking function. Moreover, there are also some approaches which do not use a ranking function, but they are able to find the top-k objects as the most relevant k objects with regard to a query.

Typical representatives of the top-k search with no ranking function are skyline queries [BKS01]. The concept of skyline query is different from the concept of ranking function. The skyline is a set of dominating objects. The skyline can be perceived as a horizontal silhouette of a city, where the taller buildings shadow (dominate) the smaller ones behind them. Thus the result of the skyline is a set of objects, which dominate the others and cannot be ranked from the best to be worst. Only each pair of objects is comparable. In the context of the ranking function, when an object is in all attributes better or equal to another object, then it should be ranked higher than the other object.

Moreover, the authors in [YM07] propose the top-k dominating query, which returns the top-k objects dominating the highest number of objects in a dataset. This query is an important tool for the decision support since it provides data analysis an intuitive way of finding the top-k objects. In addition, it combines the advantages of the top-k search and skyline queries without sharing their disadvantages as, for example, the output size can be controlled, no ranking functions need to be specified by users, and the result is independent of the scales at different dimensions.

Chapter 3

Model of User Preferences

Nowadays, users of different systems are trying to find various objects, such as laptops, jobs, holidays, etc. In most cases, these objects are of the same type and have multiple attributes. According to the values of these attributes, users are searching for objects that best suit their preferences and each user prefers some objects with different attribute values. Often, a user only looks for the top-k objects, i.e., a few objects, which are the best according to his/her preferences.

For purpose of the top-k search, there is a need to use a suitable model of user preferences, which enables the user to express his/her preferences and makes it posssible to determine the top-k objects for him/her. In this work, we use a model of user preferences based on composition of local preferences and global preferences [EPV07, GVP08, OP08]. By using this we accomplish the extending of commonly used preference models in top-k search, which are often based only on a monotone ranking function.

In Section 3.1 we focus on the problem of preference modeling. We introduce the ranking function in Section 3.2. Thereafter we provide a detailed explanation of the local preferences and the global preferences, which are described in Section 3.3 and Section 3.4, respectively. Finally, we discuss the utilization of our preference model in Section 3.5.

3.1 Preference modeling

In many information systems, such as product catalogs, the user can only restrict the values of the object attributes and objects that satisfy restrictions are selected. This approach is not always sufficient, because the result can be an empty-set or a too-large-set of objects. Therefore we introduce a model of user preferences, which enables finding of the most k preferred objects.

The main stream of top-k research considers the model of user preferences based on a ranking function (scoring function or aggregate function) [Fag02, CG99], which assigns a ranking for each object from a data set, i.e., a set of objects. Thus, it is possible to rank all objects from the best to the worst and to determine the best k objects. We suppose that each user can choose a particular form of the ranking function and thus express his/her preferences. Afterwards, it is possible to determine the best k objects according to the user's preferences, i.e., to find top-k objects for the user.

We can perceive a user preference as a *measure of suitability* of objects for a user dependent on the attribute values of the objects. In this work, we suppose a model of user preferences, in which we differentiate between local preferences and global preferences of the user. Thus, the user can express his/her preference for each particular attribute by local preferences and mutual relations between the attributes by global preferences.

3.1.1 Notation

For the purposes of this work, the following terminology is used. We suppose a set of objects X with m attributes $A_1, ..., A_m$. Each attribute A_i has its own *attribute domain*, which we denote D_{A_i}. The attribute domain D_{A_i} is a set of relevant values allowed for the attribute A_i. Every object $x \in X$ has m attribute values $A_1(x), ..., A_m(x)$ from domains $D_{A_1}, ..., D_{A_m}$ of attributes $A_1, ..., A_m$, respectively, i.e., $A_i(x) \in D_{A_i}$ for $i = 1, ..., m$. We differentiate between the attribute domain and an *actual attribute domain*. The actual attribute domain $D_{A_i}^{act}$ is the set of values $A_i(x)$ of all objects $x \in X$. Consequently, the relationship $D_{A_m}^{act} \subseteq D_{A_m}$ holds.

3.2 Ranking function

In the top-k search, the user's preferences are modeled by a *ranking function*, which assigns a *ranking* for each object from a data set. In some of the related works the ranking is denoted as a *score*. According to the ranking function, it is possible to sort objects from the data set and to find the best k objects according to the user preferences.

More formally, ranking function R evaluates the ranking (overall score) of each object $x \in X$ according to all its attribute values $A_1(x), ..., A_m(x)$ and the value of ranking is $R(A_1(x), ..., A_m(x))$. The ranking function R maps every object $x \in X$ into the real numbers, i.e.,

$$R(a_1, ..., a_m) : D_{A_1} \times ... \times D_{A_m} \rightarrow \mathbb{R}.$$

For better clarity, we can restrict the range of values of the ranking function on the unit interval $[0, 1]$, where 0 means no preference and 1 means the highest preference. In the top-k search, this restriction is a very common convention. In short notation, we denote a ranking of the object x as a function $R(x) = R(A_1(x), ..., A_m(x))$ with one variable, which we understand as a mapping

$$R(x) : X \to [0, 1].$$

Comparing $R(x_1)$ and $R(x_2)$ of any two objects $x_1, x_2 \in X$, it is possible to decide which one of them is more suitable for user U. When $R(x_1) > R(x_2)$, then x_1 is more suitable. When $R(x_1) = R(x_2)$, then x_1 and x_2 are equally suitable. According to object rankings, we can determine the best k objects. If there are more objects with the same ranking as the ranking of the best kth object, we can generally suppose that a random object with this ranking is chosen.

3.3 Local preferences

By *local preferences* we express how objects are preferred according to only particular attributes. Therefore, in this section we suppose such a *local preference function*, which assigns local ranking (local score) to objects dependent only on one attribute. We express a local preference for ith attribute A_i, as a function f_i. The function f_i is understood as a mapping

$$f_i(a_i) : D_{A_i}^{act} \to [0, 1],$$

which maps every value $a_i \in D_{A_i}^{act}$ into the unit interval $[0, 1]$.

Local preferences of user U for the attributes A_1, ..., A_m are represented by a user's functions denoted as f_1^U, ..., f_m^U, respectively [EPV07, OP08]. Then each user's function $f_i^U(a_i)$, maps every object $x \in X$ according to the value of its ith attribute $a_i = A_i(x)$ into interval $[0, 1]$.

In short notation, we denote the local preference of user U for an object $x \in X$ according to ith attribute A_i as a function $f_i^U(x)$, for which $f_i^U(x) = f_i^U(A_i(x))$ holds. Then f_i^U is understood as a mapping

$$f_i^U(x) : X \to [0, 1],$$

which maps every object $x \in X$ into interval [0, 1], i.e., the user's local preference $f_i^U(x)$ evaluates a local ranking from [0, 1] for each object $x \in X$ and thereby it is possible to sort all objects $x \in X$ according to their local scores $f_i^U(x)$.

3.3.1 Fuzzy logic perspective

Our model of user preferences is motivated by fuzzy logic [Voj06]. In classical two-valued logic it would be possible to represent only whether the object $x \in X$ is suitable (true value, ranking is 1) or not (false value, ranking is 0). In that case, the range of the ranking function is set $\{0, 1\}$.

Multi-valued logic, namely fuzzy logic, may have more than two truth values. This may be expressed as existence of various degrees of truth. In fuzzy logic, the set of truth values is the unit interval of real numbers [0, 1]. In context of user preferences, the degree of truth, which attains values in the interval [0, 1], is interpreted as the preference, where 1 means the most preferred and 0 the least preferred.

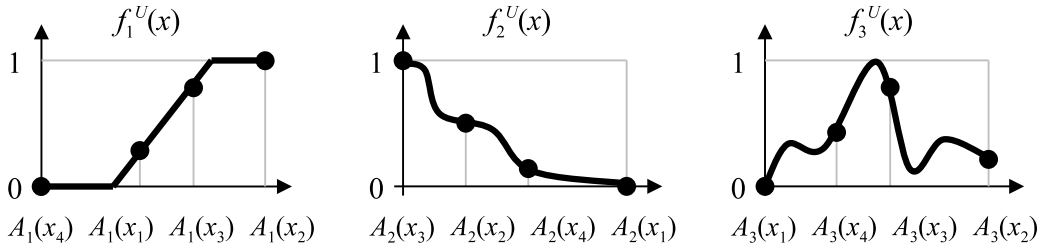
In our model, we express a local preference for ith attribute A_i as a fuzzy function f_i, which can be defined in the same way as the local preference function, i.e.,

$$f_i(A_i(x)) : D_{A_i}^{act} \rightarrow [0, 1].$$

When we perceive a local preference as a fuzzy function, then it is possible to interpret the user's fuzzy function in two different ways. We distinguish between evaluating interpretation and ordering interpretation of the fuzzy function.

If the focus lies only on evaluating the object according to user fuzzy function $f_i^U(x)$, the evaluating interpretation of a fuzzy function is used. From the value of the ith attribute of an object $x \in X$, it is possible to evaluate an object's local ranking (local score) directly by using local user preference $f_i^U(x)$. If local scores of all objects are evaluated, then the set of objects X is sortable according to $f_i^U(x)$ from the best object to the worst.

In ordering interpretation, a fuzzy function is perceived as a geometrical ordering of the ith attribute domain A_i. From a geometrical point of view, specific attribute values are irrelevant and the set of objects X can be sorted solely by course of the fuzzy function $f_i^U(x)$ (see Figure 3.1).



(a) nondecresing function (b) nonincreasing function (c) non-monotone function

Figure 3.1: Three different user's local preferences as fuzzy functions.

Example 3.3 Figure 3.1 shows an example of three different user's fuzzy functions $f_1^U(x)$, $f_2^U(x)$ and $f_3^U(x)$, where $f_1^U(x)$ is a non-decreasing function, $f_2^U(x)$ is a non-increasing function and $f_3^U(x)$ is a non-monotone function. By using the ordering-based interpretation of a fuzzy function, we can sort objects in descending order according to the course of particular fuzzy functions dependent on attribute values of the object. Namely, in Figure 3.1 for $f_1^U(x)$ we obtain the order of the objects x_2, x_3, x_1, x_4, for $f_2^U(x)$ the order of the objects x_3, x_2, x_4, x_1, and for $f_3^U(x)$ the order of the objects x_3, x_4, x_2, x_1.

3.3.2 Various attribute types

In general, we differentiate several attribute types according to the character of their attribute domain. We introduce a *categorical* attribute type and a *numerical* attribute type, for which it is necessary to adapt the application of local preferences.

Categorical attributes

Categorical attributes have a finite range of possible values. For example, a typical categorical attribute is a brand or a kind of some product, city name or personal identification number. A categorical attribute can be either *ordinal* or *nominal*, depending on whether an ordering relation is defined within the set of its possible values or not.

Thus values of an ordinal attribute are sortable naturally. If the domain of an ordinal attribute is too small, or the order of values is meaningless (identification number), then we can perceive the ordinal attribute as a nominal attribute. In the rest of the cases, we consider the ordinal attribute as a numerical attribute. In the case of a nominal attribute, we can represent possible values as a set of strings, for which a lexical ordering can be applied. It is unsuitable to use a local preference function modeled as a fuzzy function in this case.

Fast food restaurant	Preference
Burger King	0.85
KFC	0.70
McDonald's	0.75
Pizza Hut	0.60
Starbucks	0.45
Subway	0.95

Table 3.1: User's local preference for fast food restaurants.

For expressing a user's local preference for a nominal attribute, the user has to set a preference degree for each attribute value. The user has to rank each possible value of this categorical attribute. For example, Table 3.1 shows a setting of local user preferences for fast food restaurants.

Numerical attributes

We can specify a numerical attribute in the following way. A domain of a numerical attribute is a subset of a continuous interval and the ordering relation is defined within the set of its possible values. Typical examples are integer numbers, real numbers or also time values.

In the case of numerical attributes, it is suitable to use a local preference function modeled as a fuzzy function. The local preference of user U for a numerical attribute A_i is represented by fuzzy function f_i^U and the user can express his/her local preference by its course.

3.3.3 Approximation of local preferences

For common users of information systems or the Internet who do not have an education in mathematical analysis, it can be difficult to express the course of the local preference function by an analytical representation. Moreover, handling the analytical representation of a function can also be ambiguous or problematic to implement. Therefore, it is needed to establish the local preference function in a more suitable manner.

We introduce an approximation of a local preference function by *sequence of points* and *sequence of linear functions*. For this purpose, we understand a point d as an attribute value with its ranking $f^U(d)$ according to supposed user's function f^U.

We can set up $n + 1$ points d_0, ..., d_n, where $n \geq 1$. The more points we set up, the better approximation we achieve. As the first point d_0, we set up a minimal value of the attribute domain, and its ranking $f^U(d_0)$. As the last point d_n, we set up a maximal value of the attribute domain, and its

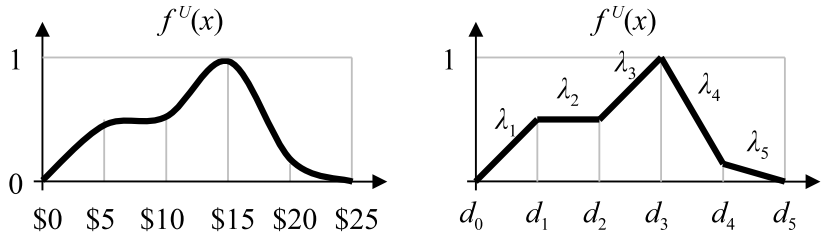


Figure 3.2: Approximation of user's local preference of price.

ranking $f^U(d_n)$. For example, a user's function f^U and its approximation are depicted in Figure 3.2, where f^U is approximated by points with attribute values d_0, ..., d_5.

From this sequence of points d_0, ..., d_n, it is possible to construct a sequence of linear functions λ_1, ..., λ_n. Each linear function λ_i is determined by points d_{i-1} and d_i, respectively. These two points form the beginning and the end of the line segment, which identifies the linear function. Thereafter, this sequence of linear functions composes an approximation of the user's function f^U.

For each attribute value a, it is possible to evaluate the local score $f^U(a)$ according to its corresponding linear function λ_i, where for index i, the relation $d_{i-1} \leq a \leq d_i$ holds. Specifically for an attribute value a, points d_{i-1} and d_i are determined by the relation $d_{i-1} \leq a \leq d_i$ and then the local score $f^U(a)$ is evaluated by the following expression

$$f^U(a) = f^U(d_{i-1}) + (a - d_{i-1}) \cdot \frac{f^U(d_i) - f^U(d_{i-1})}{d_i - d_{i-1}}.$$

With this solution, a user can express the course of his/her local preference function by setting a preference degree for selected attribute values, the same as in the case of a categorical attribute. For example, Table 3.1 shows a setting of local user preferences for the price of some product.

Price	Preference
\$0	0.00
\$5	0.50
\$10	0.50
\$15	1.00
\$20	0.15
\$25	0.00

Table 3.2: User's local preference of price.

In most cases, the course of user fuzzy functions is not too complicated, therefore the sequence of points contains only a few points. That is why we can use this representation in real applications, when an analytical representation of the local preference function is inappropriate.

3.4 Global preference

In this section we introduce *global user preferences*, by which a user can express mutual relations between the attributes $A_1, ..., A_m$. Each global user

preference will be modeled by a *global preference function*.

For this purpose, we use an *aggregate function*, which aggregates values (local scores) of attributes $A_1, ..., A_m$. Initially, the aggregate function was understood as a summation of local scores of objects. In this work, we understand the aggregate function as a combination function, which combines local rankings (local scores) of an object into one *global ranking* (global score) of the object.

More formally, we suppose that the range of each local score is interval [0, 1] and therefore we consider the global preference function g as a function with m variables $p_1, ..., p_m \in [0, 1]$ specified by scheme

$$g(p_1, ..., p_m) : [0, 1]^m \to [0, 1],$$

where p_i is a local score of an object for the corresponding attribute A_i.

3.4.1 Substitution by local preferences

In this work we suppose the model of user preferences, in which a user U can express global and local preferences by setting his/her global preference function g^U and local preference functions $f_1^U, ..., f_m^U$. In our model, it is possible to construct the user ranking function R^U by the composition of local and global preferences.

It is possible to provide a substitution of particular variables in the global preference function by local preference functions [OP08]. The user ranking function R^U originates by means of substitution $p_i = f_i^U$ in $g^U(p_1, ..., p_m)$. Then we obtain a user ranking function R^U which is defined by

$$R^U(A_1(x), ..., A_m(x)) = g^U(f_1^U(A_1(x)), ..., f_m^U(A_m(x))),$$

where $x \in X$ and $A_1(x), ..., A_m(x)$ are its respective values of attributes. As we defined before, we can use the short notation

$$R^U(x) = g^U(f_1^U(x), ..., f_m^U(x))$$

With $R^U(x)$ it is possible to evaluate the overall ranking of each object $x \in X$ according to local and global preferences of user U, to sort objects and to determine the best k objects for user U according to his/her preferences $f_1^U, ..., f_m^U$ and g^U.

For example, various users can express mutual relations between any attributes by the global preference function such as weighted average, where weights $w_1, ..., w_m$ of single attributes $A_1, ..., A_m$ determine how much the user prefers single attributes, i.e.

$$R^U(x) = \frac{w_1 \cdot f_1^U(x) + ... + w_m \cdot f_m^U(x)}{w_1 + ... + w_m}.$$

A particular user U can express his/her influence in the aggregate function by the setting of weights $w_1, ..., w_m$. When the user does not care about the ith attribute A_i, then he/she can set $w_i = 0$.

3.4.2 Monotone aggregate function

The majority of top-k algorithms have a crucial restriction. They can find top-k objects only according to a *monotone aggregate function* (monotone ranking function). Monotonicity is a reasonable property expected of an aggregate function.

We note that if a function is monotone according to all variables, it can be nonincreasing or nondecreasing. For the purposes of the top-k search, the nondecreasing aggregate function is mostly assumed as the monotone aggregate function, because it searches for objects with the highest value of aggregate function.

For example, let x_1 and x_2 be objects and the value of each attribute of x_2 is equal or higher than the value of the same attribute of x_1. When using a monotone aggregate function, we expect the value of object x_2 to be at least as high as the value of object x_1 according to the monotone aggregate function. More formally, we define a monotone function as follows.

Definition 3.4.2 (monotonicity) *A function g with m variables is monotone (nondecreasing) according to all variables when the following holds.*

$$\forall i \in \{1, ..., m\} \ : \ p_i \leq q_i \Rightarrow g(p_1, ..., p_m) \leq g(q_1, ..., q_m)$$

Typical examples of a monotone aggregate function on the unit interval $[0, 1]$ are summation, weighted summation, arithmetic average, multiplication (product), maximum and minimum.

Analogous to other top-k approaches, we restrict ourselves by using the monotone aggregate function as the global preference function in our model of user preferences. This is also a crucial restriction for the top-k algorithms introduced in this work (see Section 7).

3.4.3 Non-monotone ranking function

The main advantage of our model of user preferences is that it enables us to use a non-monotone ranking function for the top-k search, although the monotone aggregate function is required. We can obtain a non-monotone ranking function by the composition of local and global preferences, because we do not suppose any restriction for the course of the local preference functions. More formally, we prove it by the following statement.

Statement 3.4.3 (non-monotonicity) *Let f_1, ..., f_m be functions with one variable and at least one of them be non-monotone. Let g be a monotone function with m variables and R be a function with m variables constructed by such a composition of the f_1, ..., f_m and g that each ith variable of g is substituted by f_i. Then R does not have to be monotone.*

Proof 3.4.3 *Let g be nondecreasing and f_1, ..., f_m be nondecreasing. Then according to Definition 3.4.2 the following holds:*

$$\forall i \in \{1, ..., m\} \ : \ p_i \leq q_i \Rightarrow f_i(p_i) \leq f_i(s_i) \Rightarrow$$

$$\Rightarrow g(f_1(p_1), ..., f_m(p_m)) \leq g(f_1(q_1), ..., f_m(q_m))$$

and by using the relationship $R(p_1, ..., p_m) = g(f_1(p_1), ..., f_m(p_m))$ from Section 3.4.1 we achieve the following:

$$\forall i \in \{1, ..., m\} \ : \ p_i \leq q_i \Rightarrow R(p_1, ..., p_m) \leq R(q_1, ..., q_m).$$

It means that R is monotone according to Definition 3.4.2. Now we can suppose the situation from Statement 3.4.3. For example, let f_1 be non-monotone. Then f_1 has to be nondecreasing in a part of its domain and also f_1 has to be decreasing in another part of its domain. If f_1 is nondecreasing in a part of its domain, all the local preferences are nondecreasing and R is nondecreasing in the corresponding part of its domain. On the other hand, let I be the interval (a part of f_1 domain), on which f_1 is decreasing. Then according to Definition 3.4.2 for each p, q from I and for fixed variables $p_2, ..., p_m$ the following holds:

$$p < q \Rightarrow f_1(p) > f_1(s) \Rightarrow$$

$$\Rightarrow g(f_1(p), f_2(p_2), ..., f_m(p_m)) > g(f_1(q), f_2(q_2)..., f_m(q_m)).$$

Thus, by using the previous implications we achieve the following:

$$\forall p, q \in I \ : \ p < q \Rightarrow R(p, p_2, ..., p_m) > R(q, q_2, ..., q_m).$$

This result indicates that R is decreasing in its part of the domain corresponding to I, because it does not satisfy Definition 3.4.2. All these facts prove that R is non-monotone, when f_1 is non-monotone. □

Note that our model does not enable us to use an arbitrary ranking function, but only a ranking function composed from arbitrary local preference functions and a monotone aggregate function. In other words, it is such a ranking function, which can be decomposed to arbitrary local preference functions and a monotone aggregate function.

In this thesis, we will use this model of user preferences, which enables us to use a non-monotone ranking function in the top-k search. This way, we accomplish the extending of commonly used preference models based only on a monotone aggregate function. This is the main contribution of our preference model in comparison with the other top-k approaches.

3.5 Utilization of the model

In this section, we explain the application of our preference model based on composition of local and global preferences. When using the model, each particular user U specifies his/her local preference by f_1^U, ..., f_m^U and global preference by his/her global preference function g^U. Then a top-k algorithm is run on the data set and finds the best k objects according to the preferences of user U. For a different user with different preferences, the algorithm is run again on the same data set and finds other best k objects according to the new preference assignment.

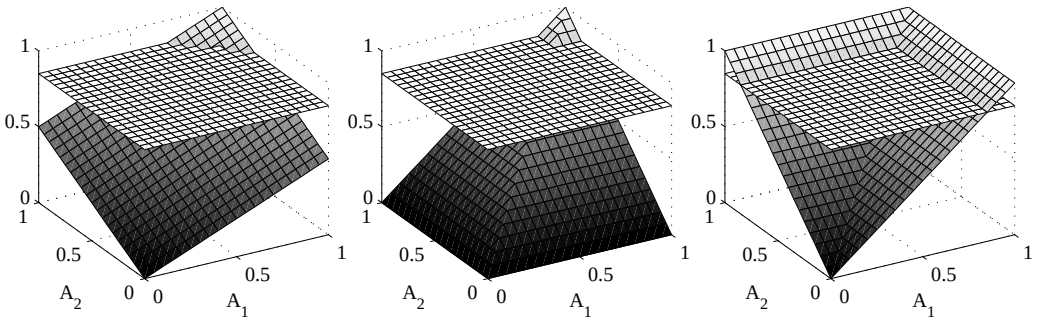
For a better understanding of our model, we show a projection of user preferences in multidimensional space and a relation between the top-k search and the nearest neighbour problem. Finally, we provide some suggestions for range query extension.

3.5.1 Projection of user preferences

We can perceive a data set X with m attributes as a set of points in multidimensional space with m dimensions. More formally, we can consider a Cartesian m-dimensional product space $D = D_1 \times ... \times D_m$, where D_1, ..., D_m are domains of attributes A_1, ..., A_m, respectively. Then we perceive an object $x \in X$ as a point of space D. In this sense, the data set X is a set of points D^X in m-dimensional space and $D^X \subseteq D$ holds.

The preference of user U represented by the user's ranking function R^U is a mapping from space D to the unit interval $[0, 1]$. When using our model, local preference functions f_1^U, ..., f_m^U map every object $x \in X$ into $[0, 1]$ according to particular attribute values. Global preference function g^U maps values of local preference functions into $[0, 1]$. In this sense, we perceive global preference function g^U as a mapping from m-dimensional space $[0, 1]^m$ to the $[0, 1]$ interval.

It is possible to visualize the course (or shape) of a user ranking function as an $m + 1$ dimensional graph, where m dimensions represent attribute domains and one additional dimension represents the values of the ranking function of the corresponding point of the m-dimensional space.



(a) sumation (b) minimum (c) maximum

Figure 3.3: Graphs of ranking functions for normalized sumation, minimum and maximum. The horizontal plane in each graph represents the threshold value of the ranking function determining the top-k objects.

For example, in Figue 3.3 the summation, minimum and maximum as a ranking function, i.e., global preference function, are depicted. We suppose a data set X with two attributes A_1 and A_2. The x-axis represents the first local score $f_1^U(x)$ of an object x and the y-axis represents the second local score $f_2^U(x)$. The z-axis in the graph represents the value of an object's ranking evaluated by the global preference function. Moreover, each graph contains the horizontal plane, which represents a threshold value of the ranking function determining the top-k objects.

Figue 3.3a shows the shape of the summation as a global preference function, which is normalized into interval $[0, 1]$. It is the same as the arithmetic average or weighted summation with weight $\frac{1}{2}$ of both attributes. The shape of this function is a plane.

There are two extreme cases of the shape of the monotone aggregate function, i.e., minimum and maximum, which are depicted in Figue 3.3b and Figue 3.3c, respectively. When we use the minimum as the aggregate function, we can observe that only a small amount of objects can have their global score almost equal to 1. They are the objects that have both local scores almost equal to 1. When we use the maximum as the aggregate function, lot of objects can have the global score almost the 1. They are objects that have one of the local scores almost equal to 1. For example, when we suppose the maximum function as the aggregate function and regular distribution of objects such as points of multidimensional space, then we have the biggest chance to obtain the best k object with global score almost equal to 1.

The shape of an aggregate function has a significant impact on the efficiency of top-k algorithms. The basic idea of top-k algorithms is to find, as

soon as it is possible, the best k objects in a data set according to a monotone aggregate function. Obviously, the top-k algorithms first search for the objects with the highest local score, which are the best candidates for the final best k objects. In this sense, the best case for the top-k algorithm is the maximum function as the aggregate function and the worst case is the minimum function.

3.5.2 Nearest neighbours and user preferences

Nearest neighbours search, also known as similarity search, is a search for the closest point in metric spaces. We show that the *k-nearest neighbours* (k-NN) search is a special case of the top-k search.

For this purpose, we can perceive a Cartesian m-dimensional product space D over domains of attributes A_1, ..., A_m as Euclidean space and use Euclidean distance as the metric on this space. Supposing the set of points D^X we can find the nearest (or the most similar) k objects in D^X to a given query point.

The Euclidean distance between points $p = (p_1, ..., p_m)$ and $q = (q_1, ..., q_m)$ is defined as the length of the line segment connecting these two points. The Euclidean distance $d(p, q)$ is given by

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + ... + (p_m - q_m)^2}.$$

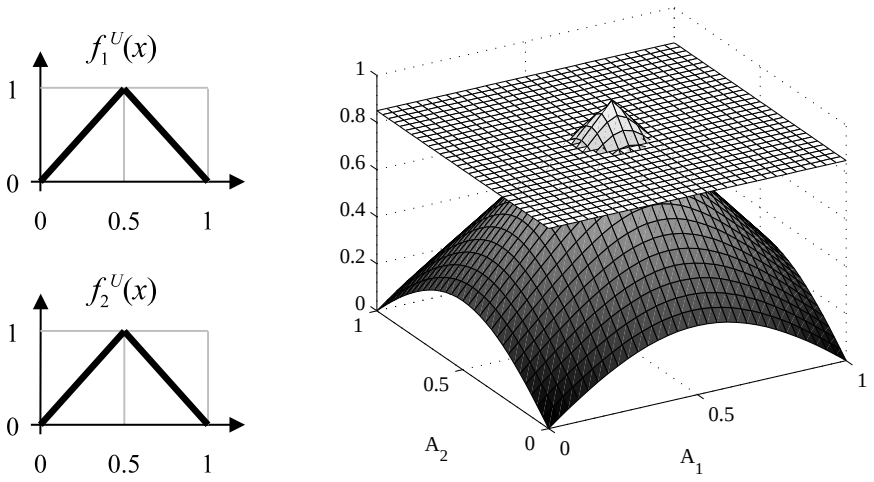
The Euclidean distance is also precieved as a measure of similarity between two points.

In the k-NN search, we are seaching for the k points with the smallest distance from a query point q. But when using the model, we are seaching for the best k objects with the highest value of a ranking function. This is possible to solve by a transformation of the Euclidean distance d from a given query object q into a ranking function R. This ranking function has to express for each object $x \in X$ its value of $R^U(x)$ dependent on the Euclidean distance of x (or similarity) from the given query object q, i.e. $d(x, q)$.

Moreover, when we compare $R(x)$ in the top-k search to $d(x, q)$ in the k-NN search, the best object x has $R(x)$ almost equal to 1 instead of $d(x, q)$ almost equal to 0 and the best object x has $R(x)$ almost equal to 0 instead of $d(x, q)$ almost equal to ∞. This problem can be solved by normalization of the distance into unit interval $[0, 1]$ and then by subtracting the normalized distance from 1.

Finally, when using this model, the transformation of the k-NN search to the top-k search is based on the composition of local and global preferences. We use local preference function

$$f_i^U(a_i) = 1 - |a_i - q_i|,$$



(a) local preferences (b) ranking function

Figure 3.4: Graph of ranking function for nearest neighbours search. The horizontal plane represents the threshold determining the top-k objects.

where q_i is the value of attribute A_i of given query object q and a_i is the value of attribute A_i of an object. We use global preference function

$$g^U(x) = 1 - \sqrt{(f_1^U(x)^2 + ... + f_m^U(x)^2)}.$$

Note that this aggregate function is monotone, but the user's ranking function does not have to be non-monotone (see Statement 3.4.3).

For example, in Figure 3.4 the ranking function for searching for the k-nearest neighbour of the point $(0.5, 0.5)$ is depicted. Both local user preferences $f_1^U(x)$ and $f_2^U(x)$ are represented by a function with the analytical representation $1 - |2(A_i(x) - 0.5)|$ with a maximum in attribute value 0.5 and two minimums in attribute values 0 and 1 (see Figure 3.4a). In terminology of the approximation (see Section 3.3.3), both local user functions are represented by a sequence of three points $(0, 0)$, $(0.5, 1)$ and $(1, 0)$, and thereby by two linear functions. Then we use a monotone aggregate function, which is based on Euclidean distance from point $(0.5, 0.5)$ and normalized into interval $[0, 1]$. Finally, we obtain the final user ranking function

$$R^U(x) = 1 - \sqrt{\frac{(1 - |2(A_1(x) - 0.5)|)^2 + (1 - |2(A_2(x) - 0.5)|)^2}{2}}.$$

The course of $R^U(x)$ is shown in Figure 3.4b. This ranking function is non-monotone, because it does not satisfy Definition 3.4.2. We can see that $R^U(x)$ is increasing in one and decreasing in the other part of its domain.

3.5.3 Range queries and user preferences

A range query retrieves all objects where some of their attribute values are between an upper and a lower boundary. The main disadvantage of the range query, it is not generally known in advance how many objects it will return. The result of the range query can be an empty-set or a too-large-set of objects. In our model of user preferences, we can apply a fuzzy logic for adjusting the range query. By doing this, it is possible to find exactly the best k preferred objects.

We propose the range query as a restriction of the attributes values. For each attribute A_i we propose a particular range from l_i to u_i, where l_i is the lower boundary and u_i is the upper boundary for values of attribute A_i. An object x is in the result of the range query, only if its attribute value $A_i(x)$ belongs to the interval $[l_i, u_i]$.

We can use the local preference function as a fuzzy function $f_i^U(x)$ instead of a range $[l_i, u_i]$ of allowable values in the result of the query. In contrast to the range query, which maps the attribute values into set $\{0, 1\}$, fuzzy function $f_i^U(x)$ maps the attribute values into the unit interval $[0, 1]$. Then the value of the fuzzy function should be almost equal to 1 for the attribute values from interval $[l_i, u_i]$. For the rest of the attribute values, i.e. outside of interval $[l_i, u_i]$, we can adjust a course of $f_i^U(x)$ according to a principle that the value of fuzzy function $f_i^U(x)$ is inversely proportional to the distance between attribute value $A_i(x)$ and interval $[l_i, u_i]$.

Figure 3.5 shows an adjusted range query for two attributes. The original range query had a range of $[\frac{1}{3}, \frac{2}{3}]$ for values of both attributes. We adjusted the course of $f_1^U(x)$ as a sequence of four points $(0, 0)$, $(\frac{1}{3}, 1)$ $(\frac{2}{3}, 1)$ and $(1, 0)$, and thereby by three linear functions. Then we adjusted the course of $f_2^U(x)$ by the function, which has the analytical representation $e^{-(3x-1.5)^8}$. These functions are depicted in Figue 3.5a. Then we use the product function as a monotone aggregate function, i.e.,

$$g^U = f_1^U(x) \cdot f_2^U(x)$$

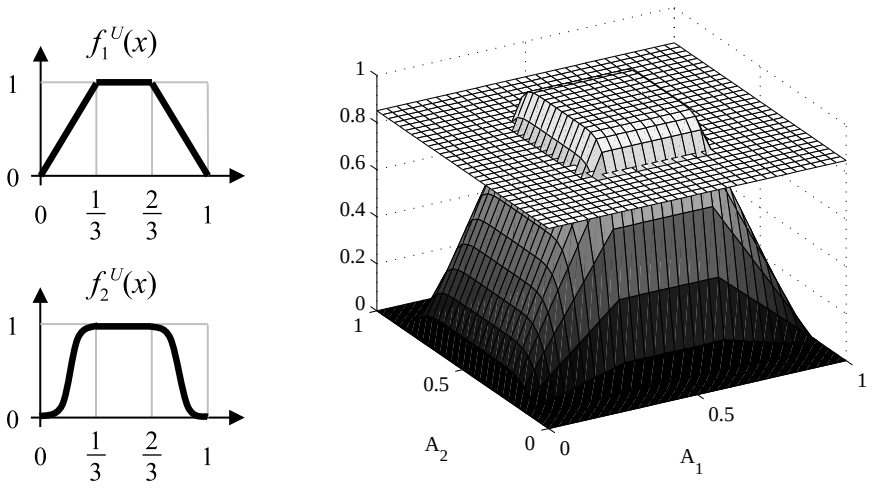
Finally, we obtain final user ranking function $R^U(x)$ defined as follows.

$$\forall x \in X, A_1(x) \in [0, \tfrac{1}{3}] : \; R^U(x) = 3 \cdot A_1(x) \cdot e^{-(3 \cdot A_2(x) - 1.5)^8},$$

$$\forall x \in X, A_1(x) \in [\tfrac{1}{3}, \tfrac{2}{3}] : \; R^U(x) = 1 \cdot e^{-(3 \cdot A_2(x) - 1.5)^8},$$

$$\forall x \in X, A_1(x) \in [\tfrac{2}{3}, 1] : \; R^U(x) = (3 - 3 \cdot A_1(x)) \cdot e^{-(3 \cdot A_2(x) - 1.5)^8}.$$

The course of $R^U(x)$ is shaped in Figure 3.5b. We can see that we can find the exact number of the top-k objects by using the ranking function as result



(a) local preferences (b) ranking function

Figure 3.5: Graph of ranking function for adjusted range query. The horizontal plane represents the threshold determining the top-k objects.

of the range query, which had its range $[\frac{1}{3}, \frac{2}{3}]$ for values of both attributes. Moreover, we can see that $R^U(x)$ is increasing in one and decreasing in other part of its domain. Therefore, this ranking function is non-monotone despite the fact that the aggregate function as the user global preference is monotone.

Chapter 4

Fagin's Algorithms

In this chapter, we focus on the family of threshold algorithms, the most influential and significant top-k search algorithms, developed by Fagin et al. [FLN03, Fag02, Fag99, Fag96], therefore we call them Fagin's algorithms. Fagin's algorithms are based on principle that data are stored in sorted lists, which can be accessed by different access modes, i.e., the random access or the sorted access. The main advantage of Fagin's algorithms is that they are able to find the top-k objects according to a monotone aggregate function (see Sectiion 3.4.2) without accessing all objects. Here it is necessary to provide more detailed description of these algorithms because we will refer to numerous details of them in further chapters.Fagin's algorithms search for the top-k objects according to a monotone aggregate function in the sorted lists, which can by accessed by the random access or by the sorted access.

In Section 4.1, we focus on all the prerequisites of Fagin's algorithms. According to utilization of the access modes, Fagin et al. [FLN03] define more variants of top-k search algorithms, namely, threshold algorithm,no random access algorithm and combined algorithm, described in Section 4.2, Section 4.3 and Section 4.4, respectively. In Section 4.5, we discuss utilization of Fagin's algorithms with respect to the different access modes. In Section 4.5.1, we also provide a discussion about disadvantages of Fagin's algorithms in context of the issues discussed in this thesis.

4.1 Prerequisites of Fagin's algorithms

As in other parts of this thesis, we assume a set of objects X, which includes n different objects, with m attributes $A_1, ..., A_m$. Each object $x \in X$ has m attribute values $A_1(x), ..., A_m(x)$ of attributes, respectively. We assume that each object has defined values of all the attributes.

The principle of Fagin's algorithms is naturally logical. They suppose that the biggest potential to be the top-k objects, i.e., objects with the k highest rankings, have some of the objects with high attribute values. Primarily, Fagin's algorithms focus on objects with high attribute values, therefore these algorithms attempt to access these objects first, and to investigate whether they can or cannot be the ones of the top-k objects as soon as possible.

4.1.1 Sorted lists

Fagin's algorithms assume that the set of object X with m attributes is stored in m sorted lists $L_1, ..., L_m$. For every attribute A_i there exists just one list L_i, which contains all the objects from set X with values of particular attribute A_i. These lists $L_1, ..., L_m$ include data items composed of an *object-value pair*, i.e., object x (or identifier of object x) and its value $A_i(x)$ of corresponding attribute A_i, which we denote $\{x, A_i(x)\}$. There is one record as pair $\{x, A_i(x)\}$ in each list L_i for each of the n objects.

These lists have to be sorted in descending order according to the values of attributes. Specifically, each list L_i is sorted in descending order by the $A_i(x)$ value. Thus, the object-value pairs $\{x, A_i(x)\}$, with the highest attribute value $A_i(x)$, are at the top of the list L_i. In other words, objects $x \in X$ are sorted in lists in descending order according to their attribute values $A_i(x)$.

4.1.2 Sorted access and random access

For understanding of the Fagin's algorithms, we have to explain two fundamental modes of access, by which they gain the pairs from the sorted lists.

The first mode of access is *sorted access* (SA). By the sorted access, the pairs in the sorted list are accessed sequentially from the top and pairs $\{x, A_i(x)\}$ are obtained gradually one after the other without the need of any extra maintenance. If an object x has the jth highest attribute value in the ith list L_i, then j sorted accesses to the list L_i are required to see this attribute value.

The second mode of access is *random access* (RA). By the random access, according to a key value, it is needed to access a data item randomly, i.e., the pairs in the sorted list are accessed directly. According to a key value (object x or identifier of object x), it is possible to obtain directly the corresponding pair $\{x, A_i(x)\}$ from the list L_i. Thus, random access allows for specific object x to get its attribute value $A_i(x)$ from the particular sorted list L_i regardless of the position of the pair $\{x, A_i(x)\}$ in the list L_i.

According to Fagin et al. [FLN03], Fagin's algorithms do sorted access in parallel to each of the m sorted lists $L_1, ..., L_m$. It means that these

algorithms access the top member of each of the sorted lists under sorted access, then they access the second member of each of the lists, and so on.

Generally, we can suppose that during the computation of Fagin's algorithms, they do the sorted access to one of the sorted lists in one step. In this sense, we can perceive the original Fagin's parallel sorted access as a sorted access into the all lists $L_1, ..., L_m$ in *round-robin style*. It means that in the first step a object-value pair is taken from L_1, in the second step the next pair is taken from L_2, in the mth step from L_m, in the $(m+1)$th step again from L_1 and so on.

It is possible to do the sorted access by more sophisticated styles as the basic realization in round-robin style proposed by Fagin et al. [Fag02]. Many related works propose such improvements denoted as *access scheduling*, which can significantly speed up the run time of Fagin's algorithms. Namely, Bast et al. [BMT06] focus on the prioritization of different sorted lists in the sorted accesses based on a Knapsack-related optimization. Also Güntzer et al. [GBK00] propose algorithm *Quick-Combine*, which observes effectiveness of reading from each of the lists with regard to speed of the algorithm and selects the most suitable list for execution of the next sorted access.

4.1.3 Monotone aggregate function

Fagin's algorithms can search for top-k objects according to a monotone aggregate function (see Section 3.4.2). In this chapter, we understand the aggregate function as a combination function, which combines (or aggregates) attribute values $A_1(x), ..., A_m(x)$ of an object x into its overall score, which we denote as $score(x) = g(A_1(x), ..., A_m(x))$. Using this aggregate function, it is possible to evaluate the score of each object and thus to determine the top-k objects, i.e. k objects with the highest value of score.

Fagin's algorithms find the best k objects correctly, only if the aggregate function g is monotone [FLN03]. We suppose that an aggregate function g with m variables is *monotone*, according to all its variables, when

$$\forall i \in \{1, ..., m\} \ : \ p_i \leq q_i \Rightarrow g(p_1, ..., p_m) \leq g(q_1, ..., q_m)$$

holds (see Section 3.4.2). An monotone aggregate function can be, according to all variables, nonincreasing or nondecreasing. Sorted lists have to be, with regard to the assumptions of Fagin's algorithms, sorted in descending order according to the attribute values of the objects. In this sense, for the correct functionality of Fagin's algorithms, it is needed to use nondecreasing aggregate function, because objects with the highest attribute values should be investigated first (see Figue 3.3 in Section 3.5.1).

4.2 Threshold algorithm

The TA algorithm (threshold algorithm) [Fag02] is the main representative of Fagin's algorithms. It is based on idea to obtain objects with high attribute values first and to investigate them immediately, whether they can or cannot be the ones of the top-k objects. The TA algorithm assumes the availability of both random access and sorted access, i.e., it needs to access the sorted lists sequentially by the sorted access and also by the random access.

By the sorted access, the TA algorithm is obtaining pairs $\{x, A_i(x)\}$ from sorted lists $L_1, ..., L_m$ sequentially in round-robin style. The TA algorithm goes through each sorted list L_i from the top to the bottom and gets pairs $\{x, A_i(x)\}$ in descending order according the $A_i(x)$. Each newly acquired object x by sorted access is obtained with a value $A_i(x)$ of only one attribute A_i depending on the list L_i, from which it was obtained.

Here, a certain problem occurs. For each new obtained object x, the TA algorithm needs to know immediately the value of its score, i.e., $score(x)$. To be able to instantly evaluate score of each new obtained object x, which has not been seen before, it is necessary to get all the other attribute values $A_1(x), ..., A_m(x)$ of that object, except $A_i(x)$ which we know from the obtained pair $\{x, A_i(x)\}$. Thereafter, we can evaluate $score(x)$ by aggregate function g, i.e., $score(x) = g(A_1(x), ..., A_m(x))$.

In order to do this, the TA algorithm has to perform the $m - 1$ random accesses into the lists, where it obtains the required attribute values $A_i(x)$ of object x, from corresponding lists $L_1, ..., L_m$ according to the object (object identifier), except the list L_i, from which the pair $\{x, A_i(x)\}$ was obtained.

In contrast to other variants of Fagin's algorithms, the TA algorithm requires to use only small amount of the memory. The TA algorithm suffices with remembering the best actual k objects, i.e., k objects the with highest evaluated rankings seen so far. For this purpose, the TA algorithm uses a temporal list *top-k*, in which it keeps the best current objects with their scores. Temporal list *top-k* is maintained in descending order according to ranking $score(x)$ of objects, which it contains. We denote *min-k* as the score of the kth best object in *top-k*, i.e. the smallest value of $score(x)$ included in the list *top-k*.

The main advantage of the TA algorithm is, that it is capable of finding top-k objects much sooner than it reads the whole lists, i.e., accesses all the objects. For this purpose, the TA algorithm is evaluating a *threshold* value while performing the sorted access in to the lists.

We define values $high_1$, ..., $high_m$ as all the last seen attribute values obtained by sorted access in each of the sorted lists $L_1, ..., L_m$. Because each list L_i is sorted in descending order according to attribute values, each $high_i$

Algorithm 1 TA algorithm (Fagin's threshold algorithm)

TA(sorted lists $L_1, ..., L_m$, aggergate function g, number k)

```
 1: top-k := ∅;
 2: for 1 ≤ i ≤ m do high_i := 1;
 3: threshold := 1;
 4: min-k := 0;
 5: repeat
 6:    for 1 ≤ i ≤ m do
 7:       {x, A_i(x)} := getNextPair(L_i); // sorted access
 8:       high_i := A_i(x);
 9:       threshold := g(high_1, ..., high_m);
10:       for 1 ≤ j ≤ m do
11:          if j ≠ i then A_j(x) := getValue(L_j, x); // random access
12:       end for
13:       score(x) := g(A_1(x), ..., A_m(x));
14:       if score(x) > min-k then
15:          if top-k.size() = k then top-k.removeKthObject();
16:          top-k.insertInTheRightPlace(x);
17:          min-k := top-k.getMin-k();
18:       end if
19:    end for
20: until threshold ≤ min-k
21: return top-k;
```

provides an estimation of the highest possible attribute value, which can be obtained by performing of the next sorted accesses in list L_i. In this way, it is possible to estimate attribute value $A_i(x)$ of each object x, which has not yet been seen as a pair $\{x, A_i(x)\}$ in list L_i.

Based on these estimations, the *threshold* value is evaluated by aggregate function g, which combines all the last seen attribute values in the lists, i.e., $threshold = g(high_1, ..., high_m)$. Since the aggregate function has to be monotone and the lists are sorted in the descending order, the TA algorithm can estimate score of each object, which has not yet been seen in any of the lists. It means that score of such that object can be only equal or lower than is the *threshold* value.

During the computation, the TA algorithm performs a simple test in which it compares the *threshold* value with *min-k* value, i.e., currently the kth best seen object. If the *threshold* value is equal or lower than the *min-k*, it can be safely assumed, that in further performing of sorted access into the lists no object with higher ranking than the *threshold* value exists [Fag02]. Therefore the objects, which have not been seen yet, cannot be in the final

top-k objects. This fact allows the algorithm to conclude the search and return the *top-k* currently seen objects with the highest ranking as result of top-k search.

The pseudo-code depicted in Algorithm 1 describes the TA algorithm. The GETNEXTPAIR(L_i) procedure represents the obtaining of the next pair $\{x, A_i(x)\}$ from list L_i by the sorted access. The GETVALUE(L_j, x) procedure represents the obtaining of attribute value $A_j(x)$ of object x from list L_j by the random access. The INSERTINTHERIGHTPLACE procedure and the REMOVEKTHOBJECT procedure represent operations, which maintain the *top-k* list sorted in descending order by scores of included objects.

L_1	L_2	L_3
$\{x_1, 1.0\}$	$\{x_3, 1.0\}$	$\{x_1, 1.0\}$
$\{x_2, 0.8\}$	$\{x_4, 0.8\}$	$\{x_4, 0.8\}$
$\{x_3, 0.6\}$	$\{x_6, 0.6\}$	$\{x_3, 0.6\}$
$\{x_4, 0.4\}$	$\{x_1, 0.4\}$	$\{x_5, 0.4\}$
$\{x_5, 0.2\}$	$\{x_2, 0.2\}$	$\{x_2, 0.2\}$
$\{x_6, 0.0\}$	$\{x_5, 0.0\}$	$\{x_6, 0.0\}$

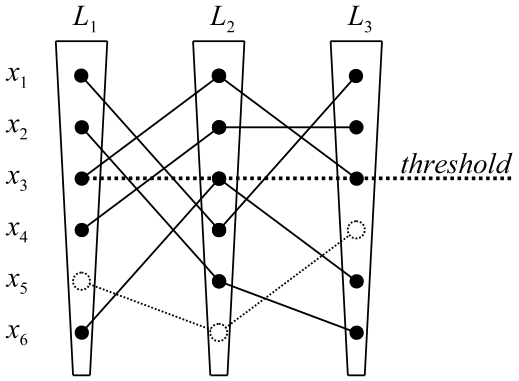


Figure 4.1: Set of six objects with values of three attributes stored in sorted lists L_1, L_2, L_3. On the right, the sorted lists are depicted. On the right, the moment of termination of the TA algorithm is depicted.

Example 4.2 Figure 4.1 depicts a set of objects, which contains six objects with values of three attributes stored in sorted lists L_1, L_2, L_3. In this example, the TA algorithm is searching for three objects with the highest score according to an aggregate function $g(p_1, p_2, p_3) = p_1 + p_2 + p_3$. This example demonstrates that the TA algorithm needs to obtain by sorted access only three pairs $\{x, A_i(x)\}$ from each of the lists. Let us assume the moment just after the execution of these nine sorted accesses. At that moment, *threshold* = $g(high_1, ..., high_m) = g(0.6, 0.6, 0.6) = 1.8$ and *top-k* includes three objects x_1, x_3 and x_4 with following scores: $score(x_1) = 2.4$, $score(x_3) = 2.2$, $score(x_4) = 2.0$. Because *min-k* = $score(x_4) = 2.0$ holds and *threshold* $\leq$ *min-k* holds, the TA algorithm is able to terminate. Moreover, the TA algorithm does not need to obtain object x_5 in any of the sorted lists, because it can estimate score of each object, which has not yet been seen, and it is lower or, at most, equal to the *threshold* = 1.8.

4.3 No random access algorithm

The NRA algorithm (no random access algorithm) was introduced by Fagin [Fag99] and then refined into the current form in [Fag02, FLN03].

The NRA algorithm supposes that set of objects with m attributes is stored in m sorted lists $L_1, ..., L_m$ and a monotone aggregate function g is given. The NRA algorithm assumes only the usage of sorted access and does not use the random access, when we compare the NRA algorithm to the TA algorithm. The NRA algorithm obtains pairs $\{x, A_i(x)\}$ by the sorted access from lists $L_1, ..., L_m$ sequentially in round-robin style, proceeds through the lists $L_1, ..., L_m$ from the top to the bottom and gets objects one by one as pairs $\{x, A_i(x)\}$.

Each new object x acquired by the sorted access is obtained with value $A_i(x)$ of only one attribute A_i depending on the list L_i, from which it was obtained. Because the NRA algorithm does not use the random acces, any other attribute values of the object x, except the value $A_i(x)$, can be obtained only by sorted access from the other sorted lists different from list L_i. It is not possible immediately, it can occur in one of the further sorted accesses performed in the other lists, i.e., if the NRA algorithm obtains a pair $\{x, A_j(x)\}$ from list L_j different from list L_i.

For an object x we consider its attribute value $A_i(x)$ as *known*, only if pair $\{x, A_i(x)\}$ has been seen in sorted list L_i. We consider attribute value $A_i(x)$ as *unknown*, if it has not been seen so far in the sorted list L_i. It is possible to evaluate $score(x)$ of an object x by aggregate function g, i.e., $score(x) = g(A_1(x), ..., A_m(x))$, only if all its attribute values $A_1(x)$, ..., $A_m(x)$ are known. In contrast to the TA algorithm, the NRA algorithm cannot obtain any of unknown attribute values immediately, because the NRA algorithm does not use the random access.

For this purpose, the NRA algorithm has to be able to estimate a score of objects, which have been seen during the obtaining of the pairs from the sorted lists. Therefore, the NRA algorithm is introducing an upper bound $Best(x)$ and a lower bound $Worst(x)$ of an object x as an estimations of its score. These estimations are based on the fact that aggregate function is monotone and the score of an object is evaluated by the aggregate function, where its variables are substituted by attribute values of the object, which are known.

Fagin et al. suppose in [Fag02] that attributes do not include negative values. Therefore, for the estimation of score lower bound of an object, it can be safely assumed, that an unknown attribute value of the object is at least equal to zero. This fact is formulated in the following statement.

Statement 4.3a (*Worst* score estimation) *Let an object x be obtained from at least one or more of sorted lists $L_1, ..., L_m$, by sorted access, and a monotone aggregate function g with m variables p_1, ..., p_m be given. Then the estimation of the lower bound $Worst(x)$ of score of the object x is defined by the aggregate function with a substitution as an expression $Worst(x) = g(p_1, ..., p_m)$, where each known attribute value $A_i(x)$ is assigned to p_i and for each unknown attribute value $p_i = 0$.*

For the estimation of score upper bound of an object, analogously to the TA algorithm, the NRA algorithm utilizes the last seen attribute values $high_1$, ..., $high_m$, the highest attribute values, which were for the last time obtained by sorted access from each of the sorted lists $L_1, ..., L_m$, respectively.

Because the sorted lists are sorted in descending order, the next obtained pair $\{x, A_i(x)\}$ from a sorted list L_i will have $A_i(x) \leq high_i$. Thus, the NRA algorithm can estimate the upper bound of an unknown attribute value. Because the aggregate function g is monotone, the NRA algorithm can estimate also the upper bound of score of objects. This fact is formulated in the following statement.

Statement 4.3b (*Best* score estimation) *Let an object x be obtained from at least one or more of sorted lists $L_1, ..., L_m$, by sorted access, and a monotone aggregate function g with m variables p_1, ..., p_m be given. Then the estimation of the upper bound $Best(x)$ of score of the object x is defined by the aggregate function with a substitution as an expression $Best(x) = g(p_1, ..., p_m)$, where each known attribute value $A_i(x)$ is assigned to p_i and for each unknown attribute value $p_i = high_i$.*

These two definitions describe how the NRA algorithm can estimate lower bound $Worst(x)$ and upper bound $Best(x)$ of score of an object x. Moreover, there exists an relationship between these two estimations and the real score of object x, which is evaluated by expression $score(x) = g(A_1(x), ..., A_m(x))$. Because aggregate function g is monotone, for an object x the following expression holds [Fag02].

$$Worst(x) \leq score(x) \leq Best(x)$$

This is the most influential relationship, by which the NRA algorithm is able to decide whether or not an object is *viable*, i.e., the object has a chance to become one of the top-k objects. In contrast to the TA algorithm, the NRA algorithm has to remember all such viable objects and update their score estimations continuously.

The NRA algorithm uses a list denoted as *top-k*, which includes actually the best k object. In contrast to the TA algorithm, the *top-k* list includes k

objects with the highest value of $Worst(x)$ seen so far, because the NRA algorithm deals with score estimations. The *top-k* list is maintained in descending order according to $Worst(x)$ of objects, which it includes. We denote *min-k* as $Worst(x)$ value of the kth best object in *top-k*, i.e. the smallest value of $Worst(x)$ included in the *top-k* list.

Additionally, the NRA algorithm uses a temporary set of objects denoted as *candidates*. The *candidates* set includes objects, beyond the *top-k*, which still have a chance to become one of the top-k objects. In this sense, it is each object x, which has been seen in at least one or more of the sorted lists, and its $Best(x)$ value is higher than *min-k* value. In other words, *candidates* is the set of objects which are candidates for moving into the *top-k* list.

The NRA algorithm maintains the *top-k* list and the *candidates* set, according to previously described rules, during the processing of the sorted lists by the sorted access. In the course of doing this, the NRA algorithm has to remember for each object, which is included in *top-k* or *candidates*, all its attribute values obtained from the sorted lists by sorted access so far.

After obtaining each new pair $\{x, A_i(x)\}$, score estimations of the object x have to be updated. After such an update, it is needed to inspect whether or not the object x belongs to the *top-k* list or the *candidates* set, and then to make necessary updates of them. If an object x has its $Best(x) \leq$ *min-k*, then object x is not viable, i.e., object x can never become one of the *top-k*, and object x is removed from *candidates* set (or *top-k* list).

Moreover, whenever one of last seen attribute values $high_1$, ..., $high_m$ in the sorted lists was decreased, then the whole *candidates* set has to be revised, because the $Best(x)$ value of some objects in *candidates* set would be changed. It means that the $Best(x)$ value of such an object x can be decreased. In such case, if $Best(x) \leq$ *min-k*, then object x has to be removed from the *candidates* set.

The NRA algorithm is able to stop, when the *top-k* contains k objects and the *candidates* set is empty, i.e., there is no viable object, which has its $Best(x)$ value higher than *min-k*. In other words, there is no object, which has a chance to become a member of the *top-k* list. At that moment the *top-k* list includes the final top-k objects. We note that some attribute values of these objects can be unknown, because the NRA algorithm deals only with score estimations.

The pseudo-code depicted in Algorithm 2 describes the NRA algorithm. The GETNEXTPAIR(L_i) procedure represents the obtaining of the next pair $\{x, A_i(x)\}$ from list L_i by the sorted access. The procedures INSERT, UPDATE, REMOVE and REMOVEKTHOBJECT represent operations with the objects included in the *top-k* list and the viable objects included in the *candidates* set.

Algorithm 2 NRA algorithm (Fagin's no random accesses algorithm)

NRA(sorted lists $L_1, ..., L_m$, aggergate function g, number k)

```
1: top-k := ∅;
2: candidates := ∅;
3: for 1 ≤ i ≤ m do high_i := 1;
4: min-k := 0;
5: repeat
6:    for 1 ≤ i ≤ m do
7:       // Obtaining the next pair from one of the lists
8:       {x, A_i(x)} := getNextPair(L_i); // sorted access
9:       high_i := A_i(x);
10:      for 1 ≤ j ≤ m do
11:         if A_j(x) is known then { worst_j := A_j(x); best_j := A_j(x); }
12:         if A_j(x) in unknown then { worst_j := 0; best_j := high_j; }
13:      end for
14:      Worst(x) := g(worst_1, ..., worst_m);
15:      Best(x) := g(best_1, ..., best_m);
16:      // Processing of the object from new pair
17:      if x ∈ top-k then
18:         top-k.update(x);
19:      else if Worst(x) > min-k then
20:         x' := top-k.removeKthObject();
21:         top-k.insert(x);
22:         if x ∈ candidates then candidates.remove(x);
23:         min-k := top-k.getMink();
24:         if Best(x') > min-k then candidates.insert(x');
25:      else if Best(x) > min-k then
26:         if x ∉ candidates then candidates.insert(x);
27:         candidates.update(x);
28:      else
29:         if x ∈ candidates then candidates.remove(x);
30:      end if
31:   end for
32:   // Removing of non-viable candidates
33:   if min-k was increased or high_1, ..., high_m were decrased then
34:      for all x ∈ candidates do
35:         candidates.update(x); //according to high_1, ..., high_m
36:         if Best(x) ≤ min-k then candidates.remove(x);
37:      end for
38:   end if
39: until candidates = ∅ and top-k.size() = k
40: return top-k;
```

4.4 Combined algorithm

The CA algorithm (combined algorithm) [Fag02] is a merger between the NRA algorithm and the TA algorithm. The CA algorithm deals with different execution cost of access modes and tries to minimize the overall cost of computation of the top-k objects. The CA algorithm is based on an extension of the NRA algorithm with the ability to perform the random accesses.

Analogously to the NRA algorithm, the CA algorithm obtains pairs, in form $\{x, A_i(x)\}$, by the sorted access from sorted lists $L_1, ..., L_m$ sequentially in round-robin style and uses *top-k* list with actually the best k objects. Additionally, the CA algorithm uses temporary list of objects denoted as *candidates* (the NRA algorithm uses *candidates* only as a set of objects), in which the included objects have to be sorted in descending order according to their score upper bound $Best(x)$.

We denote the object x from the *candidates* list, which has the highest value of $Best(x)$, as the *top candidate*. Consequently, the top candidate has the highest chance to move into the *top-k* list. This will happen, if the top candidate has its value of $Best(x)$ higher than *min-k*. In this case, the *min-k* value can be increased and afterwards a lot of objects can be removed from the *candidates* list, because their $Best(x)$ values will no longer not be higher than *min-k*.

Analogously to the NRA algorithm, the CA algorithm is able to stop as soon as the *candidates* list is empty. Therefore, the minimization of the number of objects in the *candidates* list is crucial. Additionally to the NRA algorithm, the CA algorithm uses the random access. The most efficient method to utilize the random access is, that the CA algorithm gets the unknown attribute values of the top candidate. Afterwards, the CA algorithm can evaluate the exact value of score of the top candidate immediately.

Fagin [Fag02] defines the cost ratio $\lfloor c_R/c_S \rfloor$ (see Section 4.5 for details) and a *step* of the CA algorithm. In the previously used terminology of this chapter, we define the step of the CA (or NRA) algorithm as m sorted accesses processed by one cycle of round-robin style, i.e., only one pair is obtained by sorted access from each of the m sorted lists.

Finally, the CA algorithm is an extension of the NRA algorithm, in which the cost ratio drives the execution of random accesses in the CA algorithm. For the optimality of the CA algorithm such rule holds, that the CA algorithm performs random accesses every $\lfloor c_R/c_S \rfloor$ steps. By performing these random accesses, the CA algorithm obtains all unknown attribute values of the top candidate, i.e., the object x with the highest value of $Best(x)$ in *candidates*.

The pseudo-code depicted in Algorithm 3 describes the CA algorithm.

Algorithm 3 CA algorithm (Fagin's combined algorithm)

CA (sorted lists $L_1, ..., L_m$, aggergate function g, number k)

```
 1: top-k := ∅;
 2: candidates := ∅;
 3: for 1 ≤ i ≤ m do high_i := 1;
 4: min-k := 0;
 5: step := 0;
 6: repeat
 7:    if step < ⌊c_R/c_S⌋ then
 8:       step := step + 1;
 9:       for 1 ≤ i ≤ m do
10:          // NRA: Obtaining the next pair from one of the lists // sorted access
11:          // NRA: Processing of the object from new pair
12:       end for
13:    else
14:       step := 0;
15:       x := candidates.getTopCandidate();
16:       for 1 ≤ j ≤ m do
17:          if A_j(x) is unknown then A_j(x) := getValue(L_j, x); //random access
18:       end for
19:       Worst(x) := g(A_1(x), ..., A_m(x));
20:       Best(x) := Worst(x);
21:       // NRA: Processing of the object from new pair
22:    end if
23:    // NRA: Removing of non-viable candidates
24: until candidates = ∅ and top-k.size() = k
25: return top-k;
```

The main part of the CA algorithm is identical to the NRA algorithm depicted in Algorithm 2. The parts of the NRA algorithm, which have a comment at its beginning, are referred in the CA algorithm by "NRA:", namely;

- "Obtaining the next pair from one of the lists" from line 7 to line 15,
- "Processing of the object from new pair" from line 16 to line 30 and
- "Removing of non-viable candidates" from line 32 to line 38.

In this pseudo-code, in addition to the NRA algorithm, we use procedure getTopCandidate, which obtains the top candidate from the *candidates* list. Performing the random accesses is driven by the cost ratio $\lfloor c_R/c_S \rfloor$ in

such manner that an additional variable *step* is incremented after each round of sorted accesses. If *step* is equal to $\lfloor c_R/c_S \rfloor$, variable *step* is decreased to zero and the CA algorithm performs random accesses obtaining the unknown attribute values of the top candidate.

4.5 Utilization of Fagin's algorithms

In [FLN03], the authors consider that there is a need to distinguish between the *cost of the random access* c_R and the *cost of the sorted access* c_S. Naturally, the execution cost of random access is usually greater than the execution cost of sorted access, because there is a need of some extra maintenance such as searching of an additional index. A typical example occurs when the costs correspond to disk access. Moreover, the usage of one of the access modes can be restricted or absolutely impossible in some real applications.

In sections 4.2, 4.3 and 4.4, we introduced three variants of Fagin's algorithms. When the cost of random access is insignificant or very low, the best choice is to use the TA algorithm, which assumes the availability of both random access and sorted access. On the other hand, when the random access is too expensive or simply impossible, then the only possibility is to use the NRA algorithm, which assumes only the usage of sorted access.

When both access modes are available, but with c_R higher than c_S, the usage of the CA algorithm is the most suitable. In such case, the CA algorithm searches for the top-k objects similarly as the NRA algorithm, but sometimes it performs the random access. The cost ratio $\lfloor c_R/c_S \rfloor$ drives the optimal performing of random accesses and sorted accesses during the CA algorithm's computation. The CA algorithm performs at most $m - 1$ random accesses after performing $m \cdot \lfloor c_R/c_S \rfloor$ sorted accesses. It is interesting to notice that when the cost of random access c_R is very cheap, i.e., it is approximately equal to the cost of sorted access c_S, then $\lfloor c_R/c_S \rfloor = 1$ holds. Then the computation of the CA algorithm is nearly equivalent to the computation of the TA algorithm.

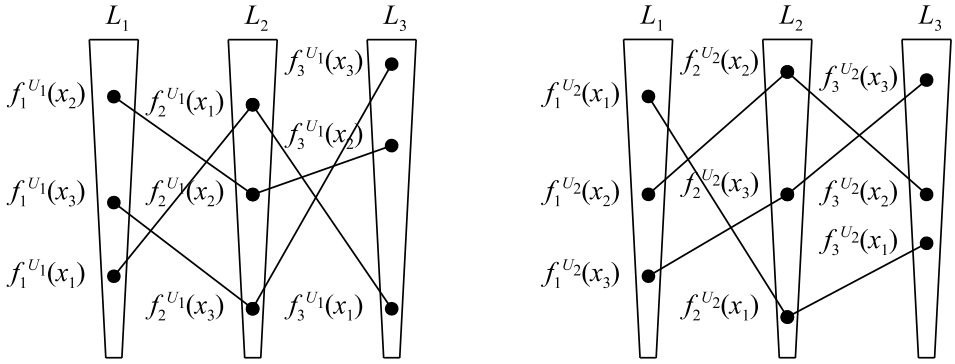
4.5.1 Problem with support of local preferences

In terms of the model of user preferences introduced in Chapter 3, original Fagin's algorithms offer the possibility of ranking objects only globally with a monotone aggregate function g. They allow a user to find the best k object only according to his global preference. A main disadvantage of Fagin's algorithms is that they do not support local user preferences directly.

Let us assume that one of Fagin's algorithms uses the local preferences.

A user U_1 can prefer objects $x \in X$ with his/her local preferences determined by fuzzy functions $f_1^{U_1}$, ..., $f_m^{U_1}$. If we want to apply his/her local preferences in the algorithm, it is necessary that every ith sorted list L_i contains pairs $\{x, f_i^{U_1}(x)\}$, instead of pairs $\{x, A_i(x)\}$. Moreover, each sorted list L_i has to be sorted in descending order according to values of $f_i^{U_1}(x)$.

For finding top-k objects for another user U_2 by the same algorithm, it is necessary to create new sorted lists, before the start of the algorithm. For a user U_2 with different local preferences $f_1^{U_2}$, ..., $f_m^{U_2}$, the sorted lists $L_1, ..., L_m$ have to be created again according to local preferences $f_1^{U_2}(x), ..., f_m^{U_2}(x)$. This situation is depicted in Figure 4.2. Because each user has different local preferences, the same objects occur at different positions in the sorted lists.



(a) local preferences of user U_1 (b) local preferences of user U_2

Figure 4.2: Different positions of three objects in three sorted lists according to local preferences $f_1^{U_1}$, $f_2^{U_1}$, $f_3^{U_1}$ of user U_1 and $f_1^{U_2}$, $f_2^{U_2}$, $f_3^{U_2}$ of user U_2.

If we want to change the order of objects in sorted list L_i according to local preference f_i^U, then we have to obtain all pairs from list L_i and sort them according to f_i^U. Thus, the whole list L_i has to be accessed. In this sense, the application of Fagin's algorithms is meaningless, because they attempt to minimize the number of pairs obtained from the sorted lists.

The application of the local preferences in Fagin's algorithms causes this problem of sorted list preprocessing. This problem can be solved by application of a new model of the sorted list. We focus on this issue in Chapter 6.

Chapter 5

Data Structures Based on B-trees

This chapter focuses on tree-based data structures, which can be utilized for efficient processing of the top-k search. We focus on data structures related to the B-tree [BM72]. First, we focus on the B-tree and its variants, which are the fundamental building blocks of our research, namely, the B$^+$-tree [Com79]. Then we present the multidimensional B-tree, which was developed in [SO82]. The multidimensional B-tree is composed of B-trees and enables a set of objects to be indexed according to more attributes simultaneously. Moreover, we introduce a variant of the multidimensional B-tree composed of the B$^+$-trees proposed by us in [OP08], which constitutes the base of our research. Finally, we introduce the multidimensional B-tree with lists as a new data structure, which we developed in [OP09, OP10]. These data structures provide a dynamic environment for the top-k search and enable us to use some advanced models of user preferences based on the local preferences. This chapter provides a coherent survey of data structures based on the B-trees used in this thesis.

In Section 5.1, we provide a review of the B-tree, the redundant B-tree, and the B$^+$-tree. In Section 5.2, we focus on multidimensional indexing based on reduction of the dimensionality. In Section 5.3, we focus carefully on the multidimensional B-tree. In Section 5.4, we provide the multidimensional B-tree composed of the B$^+$-trees. Finally, in Section 5.5, we introduce the multidimensional B-tree with lists, which partially solves the problem of the placement of the indexed attributes in the multidimensional B-tree.

5.1 B-tree

The B-tree is a dynamic data structure, which enables a set of objects to be indexed according to one attribute. We note that in the past 40 years

B-tree and its various variants have been used extensively, both for database systems and file systems. They are attractive because they can be kept balanced during insertions and deletions at a small cost.

The B-tree is multiway balanced search tree, which is composed of nodes. The B-tree was originally developed for data organization in external memories such as disk drives. Retrieval of data from the disk drive is realized by accessing particular disk block as data unit of the disk. Therefore one node of a B-tree typically corresponds to one disk block.

We distinguish three types of nodes. Each B-tree has one root node, internal nodes and leaf nodes. The nodes are organized in levels. The root node is situated in the first level of the B-tree followed by next levels, which include internal nodes. The leaf nodes are situated in the last level of the B-tree. In general, each node in one level of a B-tree, can contain pointers to other nodes situated in the next level. Each node, except the root, has one parent node in the previous level.

More formally, we define B-tree of order m by the following definiton.

Definition 5.1a (B-tree) *A B-tree of order m is an m-ary tree, if it satisfies the following properties:*

1. *The root is either a leaf or has at least two children.*
2. *Each node, except for the root and the leaves, has at least $\lceil m/2 \rceil$ and at most m children.*
3. *Each node is composed of at least $\lceil m/2 \rceil - 1$ and at most $m-1$ elements (key values and their associated data).*
4. *Each path from the root to a leaf has the same length.*
5. *Data in a node are organized as follows:*

   $$p_0, (k_1, p_1, d_1), (k_2, p_2, d_2), ..., (k_n, p_n, d_n), un,$$

   where p_0, p_1, ..., p_n are pointers to children, k_1, k_2, ..., k_n are key values, d_1, d_2, ..., d_n are corresponding associated data, un is an unused space in the disk block, and $\lceil m/2 \rceil - 1 \leq n \leq m-1$ holds. Elements (k_i, p_i, d_i), where $i = 1, ..., n$, are sorted in ascending order according to the key values.
6. *If pointer p_i refers to its corresponding subtree $U(p_i)$, then*
   (a) *for each $k \in U(p_{i-1})$ inequality $k < k_i$ holds and*
   (b) *for each $k \in U(p_i)$ inequality $k > k_i$ holds, where $i = 1, ..., n$.*

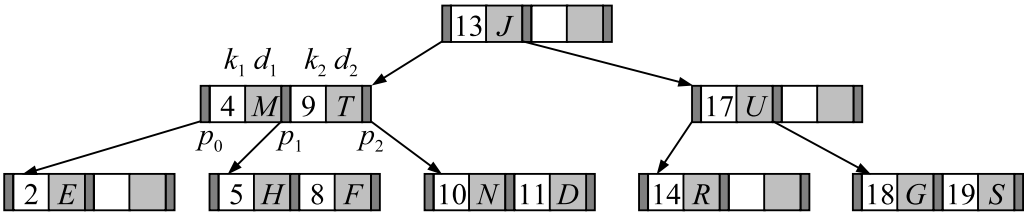


Figure 5.1: A B-tree (Definition 5.1a) with associated data in its all nodes.

A B-tree, which satisfies the properties of Definition 5.1a, we call *non-redundant B-tree*. In a non-redundant B-tree, each key occurs only once, because of property 6 of Definition 5.1a, and according to a key it is possible to find the associated data. We call this ability of B-tree the *random search*.

The pair k and d represents one data record with key k and the associated data by key k, i.e., an element of a node of the B$^+$-tree. If we assume that a set of objects X with an attribute A is indexed in a B-tree, then we can perceive the key k as an attribute value from actual attribute domain D_A^{act} and associated data as an object or a specific information about object, e.g., identifier of the object. An object $x \in X$ is indexed in the B-tree by key $k = A(x)$, where $k \in D_A^{act}$.

For example, Figure 5.1 depicts a B-tree of order 2 with three levels. The B-tree includes some numbers as the keys (attribute values) and the corresponding letters as the associated data (objects or object identifiers). The letters are indexed by numbers.

Self balancing of B-tree

We call the number of levels of the B-tree the *height* of the B-tree, which we denote as h. The main advantage of B-trees is that they are always height-balanced. It means that each path from the root to each leaf has the same length and all the leaf nodes are in the last level of the B-tree. This is guaranteed by algorithms for insert and delete operations, which maintain the B-tree constantly balanced.

The insert algorithm is based on a simple principle. First, the insert algorithm finds a node, where a new record (key and its associated data) should be added. If there are less than $m - 1$ records, then the new record is added into this node. Otherwise the node is simply split into two nodes, which are half-filled, and the corresponding median record is added into their parent node. Analogously, this process is propagated into higher levels of the B-tree. That is why the B-tree is defined in such a way that the nodes can include from $\lceil m/2 \rceil - 1$ to $m - 1$ records.

The delete algorithm is based on a similar principle. Sometimes an insert or delete can occur that it causes a more complicated adjustment of the B-tree. But even in that case, only a few additional nodes of the B-tree are changed, and thus few disk blocks are accessed. For example, if a new record should be added into a root which is full then the height of the B-tree increases by one because the root is split into two nodes and a new root of the B-tree has to be created, which will be their parent. Such details are not crucial for purposes of this work, therefore we do not need to describe them here.

Complexity of B-tree

What is important to us is that the B-tree is kept height-balanced after each insert or delete operation. More exactly, let h be the height of the B-tree. Comer shows in [Com79] that the B-tree of order m, which includes n keys, has height

- $h = \lceil \log_m(n+1) \rceil$, when all the nodes are full-filled, and
- $h = \lfloor \log_{\lceil m/2 \rceil}(\frac{n+1}{2}) + 1 \rfloor$, when all nodes are half-filled and the root node includes only one key and has two children.

If the height of the B-tree is h, the random search requires to access at most h nodes, i.e., one node on each level of the B-tree until it finds the searched key. That is the why the random search has logarithmic complexity $O(\log_{\lceil m/2 \rceil}(n))$.

For example, we can suppose a B-tree of order 100 with 5 levels. Then it can include from 12.5 million to 10 000 million keys. It means that, for finding the record according to a particular key, it is necessary to access at most 5 nodes of the B-tree despite the fact it includes millions of keys.

We note that insert and delete operations also have logarithmic complexity. On average according to [Com79], they need to access and modify approximately two times more nodes than the random search. It means insert and delete also affect only a few disk blocks as nodes of B-tree.

5.1.1 Redundant B-tree

The stated properties in Definition 5.1a are often modified in practical implementations of real systems. The most common requirement is to minimize the height of the B-tree, i.e., the length of each path from the root to a leaf. For this purpose, the associated data are moved into the leaf nodes of the B-tree. Threfore, the internal nodes of B-tree include only keys and pointers

to child nodes. For the same size of the disk block, the number of keys in each internal node can be incrased and thus the height of the B-tree can be decreased. More formally, we define this modification as *redundant B-tree*, where the properties 5 and 6 are changed compared to the original definition of the B-tree.

Definition 5.1b (redundant B-tree) *A redundant B-tree of order m is an m-ary search tree, if it satisfies properties from 1 to 4 of Definition 5.1a and the following properties:*

5. *Data in an internal node are organized as follows:*

   $$p_0, (k_1, p_1), (k_2, p_2), ..., (k_n, p_n), un,$$

   and data in a leaf node are organized as follows:

   $$(k_1, d_1), (k_2, d_2), ..., (k_n, d_n), un,$$

   where p_0, p_1, ..., p_n are pointers to children, k_1, k_2, ..., k_n are key values, d_1, d_2, ..., d_n are corresponding associated data, un is an unused space in the disk block and $\lceil m/2 \rceil - 1 \leq n \leq m - 1$ holds. Elements (k_i, p_i), or (k_i, d_i), where $i = 1, ..., n$, are sorted in ascending order according to the key values.

6. *If pointer p_i refers to its corresponding subtree $U(p_i)$, then*

   (a) *for each $k \in U(p_{i-1})$ inequality $k \leq k_i$ holds and*

   (b) *for each $k \in U(p_i)$ inequality $k > k_i$ holds, where $i = 1, ..., n$.*

In a redundant B-tree, a key can occur more times in various nodes of the B-tree, because the property 6.(a) is less strict as in Definition 5.1a. Keys of an internal node are used as border keys. Each key k_i determines the border between $U(p_{i-1})$ and $U(p_i)$. According to the property 6.(a), which enables equality, key k_i can occur in subtree $U(p_{i-1})$. Therefore, the key can be duplicated in more internal nodes in different levels of the B-tree. Moreover, all the values from D_A^{act} are included in the leaf level of the redundant B-tree and are sorted in ascending order (from the left to the right).

As in the case of the B-tree, the redundant B-tree is kept balanced after each insert or delete operation. In contrast to the B-tree, the random search has to access one node on each level of the redundant B-tree, because the associated data are placed in its leaf nodes. The random search has logarithmic complexity, since it depends on the number of levels of the redundant B-tree. Figure 5.2 depicts a redundant B-tree of order 2 with three levels, which includes the same set of letters as in Figure 5.1 indexed according to the corresponding numbers, which we percieve as the keys.

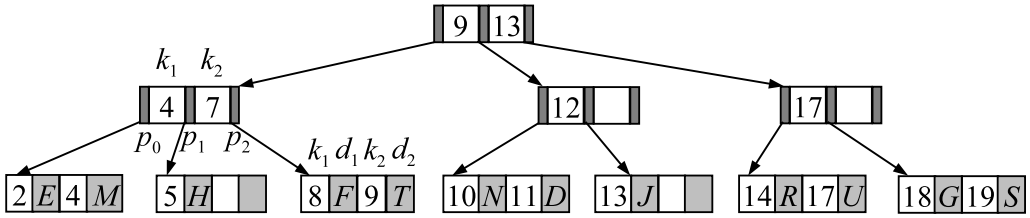


Figure 5.2: A redundant B-tree (Definition 5.1b) with associated data only in its leaf nodes.

Range query

A *range query* is an operation that retrieves all objects, whose attribute values are between lower and upper boundary. The range query with a lower bound w and a upper bound u returns all objects, which are indexed in a B-tree by such a key k, for which $w \leq k \leq u$ holds. Sometimes, w or u does not have to be defined, then the range query returns all objects, for which $k \leq u$ or $w \leq k$, respectively, holds. We note that for a range query, it is not generally known in advance how many entries the range query will return, or if it will return any at all.

The B-tree and the redundant B-tree are primarily designed for the random search. The execution of a range query in these B-trees is not efficient, because the sequential searching of the nodes, in order to obtain neighboring keys, is too complicated. The B-tree has to be searched recursively. It requires to access more nodes than it is really needed. This inefficiency can be solved by using the B$^+$-tree.

5.1.2 B$^+$-tree

The B$^+$-tree is such an extension of a redundant B-tree that its leaf nodes are linked. Each leaf node includes an additional pointer to its right neighboring leaf node. Based on this fact, the B$^+$-tree provides the possibility to process the range query efficiently.

Definition 5.1c (B$^+$-tree) *A B$^+$-tree of order m is a redundant B-tree of order m, whose leaf nodes are organized as follows:*

$$(k_1, d_1), (k_2, d_2), ..., (k_n, d_n), r, un,$$

where r is a pointer to the right neighboring leaf node.

Let a set of objects X be indexed according to the values of one attribute A in a B$^+$-tree. Because the B$^+$-tree is derived from the redundant B-tree, all the keys, i.e., all values from D_A^{act}, are included in leaf nodes in the last

level of the B$^+$-tree. Since the leaf nodes are linked, it is possible to cross the B$^+$-tree through the leaf level and to get its keys and its associated data sequentially. We call this ability of B$^+$-tree the *sequential search*.

The sequential search can be used for execution of a range query in the B$^+$-tree, since all the keys are sorted (from the left to the right) in ascending order according to their values. The execution of the range query in the B$^+$-tree is more efficient than in the B-tree (redundant B-tree), which has to be searched recursively.

A range query has a lower bound w and an upper bound u and its result contains all associated data, the keys of which lied inside this boundary. First, if it is possible, key k, which is equal or greater than w, is found by the random search in a leaf node. If such a key does not exist in the B$^+$-tree or key k is greater than u, the result of the range query is empty. Otherwise, the sequential search is used, which starts at the leaf node which contains the key k. The leaf nodes are scanned one by one by using the pointer to the neighboring leaf node. Keys are obtained from each leaf node sequentially in ascending order according to values of the keys, until they are equal or lower than u. For each such a key, associated data are added into the result of the range query.

In this work, we extensively use a variant of the B$^+$-tree, in which the leaf nodes are linked in two directions. Each leaf node includes one additional pointer.

Definition 5.1d (double linked B$^+$-tree) *A B$^+$-tree of order m is a redundant B-tree of order m, whose leaf nodes are organized as follows:*

$$l, (k_1, d_1), (k_2, d_2), ..., (k_n, d_n), r, un,$$

where l is a pointer to the left neighboring leaf node (or is empty) and r is a pointer to the right neighboring leaf node (or is empty).

In such case we can execute the sequential access in two directions. Let a set of object X be indexed a B$^+$-tree according to attribute A. When we cross the leaf level from the left to the right, we can obtain all the objects $x \in X$ sequentially in ascending order according to $A(x)$ and from the right to the left in descending order.

This extension of the original B$^+$-tree enables us to process the range query in reversed direction, which provides the results sorted in descending order according to key values. First, if it is possible, key k, which is equal or less than upper bound u, is found by the random search in a leaf node. If such a key does not exist in the B$^+$-tree or key k is lower than w, the result of the range query is empty. Otherwise, the sequential search is used. It starts at the leaf node containing k. The leaf nodes are scanned from the

right to the left one by one by using the pointers to the neighboring leaf nodes. From each leaf node its keys and corresponding associated data are obtained sequentially in descending order according to the values of the keys, until they are equal or greater than w. Then the result of the range query includes associated data sorted in descending order.

5.1.3 External associated data

In general, different objects with the same value of attribute A can exist in X. It means that more objects in X should be associated for a key $k \in D_A^{act}$. There are two possibilities how to solve it for a B-tree or B$^+$-tree.

The first possibility requires changing the structure and maintenance of the nodes, in which it should be possible to store more objects for one key. In this case, the size of one record, which is composed of one key and one or more objects, can be bigger. If we suppose that the size of the disk block is the same, then the number of records in the disk block will be smaller. It can cause the higher number of levels in the B-tree and thus the complexity of the search, as well as an increase of insert and delete operations.

The second possibility is to move the associated data out of the B-tree. In the leaf nodes, a pointer to associated data is used instead of the associated data. In contrast to the definitions established in this section, the associated data from each record (k_i, d_i) are referred by d_i, which functions as a pointer to the associated data. Then we can use a suitable structure of the associated data, which can be independent from the structure of the B-tree and its nodes. The solution using with external associated data does not affect the number of records in the nodes. Thus, the properties, which are based on the order of B-tree, will be the same with regard to size of the disk block.

5.1.4 Object array

Let us suppose more objects with the same attribute value $k \in D_A^{act}$. If we want to index the set of objects in the B$^+$-tree, these object have to be indexed by the same key k as its associated data.

Therefore, in this thesis, we use an *object array*. The object array we perceive as the external associated data stored out of the B$^+$-tree. In general, for each value $k \in D_A^{act}$ we can use one object array, which includes all objects with attribute value k.

By using the object arrays, the structure of leaf nodes of the B$^+$-tree is slightly changed. For every key k there is a pointer d to associated data as an object array, which includes one or more objects with attribute values equal to k. Moreover, if we deal in terms of disk blocks, we can suppose an

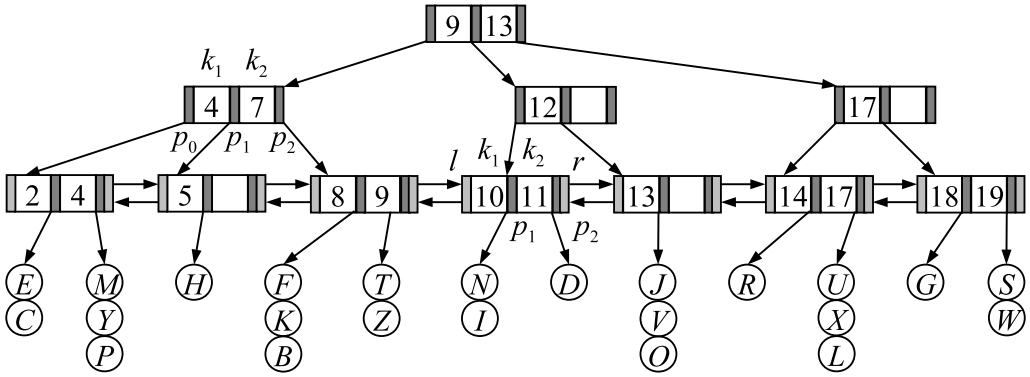


Figure 5.3: A B$^+$-tree of order 2 with leaf level linked in two directions. Associated data are referred only from leaf nodes of the B$^+$-tree.

object array as a new type of node. In this case, a pointer to the associated data refers to a disk block as the other pointers refer to children nodes in the B-tree. If a number of the objects is higher than the number of the objects which fit in the disk block, then we can use more concatenated disk blocks, which can be perceived as one object array.

Figure 5.3 depicts the B$^+$-tree of order 2, the leaf nodes of which are linked in two directions. Objects with the same value are stored in one object array, which is represented by the sequence of circles in the figure. We notice that the set of objects stored in this B$^+$-tree has the same actual attribute domain as the set of objects depicted in figures of the B-tree and the redundant B-tree (see Figure 5.1 and Figure 5.2). The following example shows the processing of a range query in the B$^+$-tree from Figure 5.3.

Example 3.3 The range query has its lower bound 9 and its upper bound 11, and we need a sequence of objects as a result sorted in descending order according to the value of attribute. The processing starts with the random search, which finds such a key in a leaf node which is smaller or equal to the upper bound 11. Then the sequential search to the left is used. Keys 11 and 10 are obtained from the leaf node and their corresponding associated data are added to the result. Namely, object x_D is obtained from the object array of key 11, objects x_N and x_I are obtained from the object array of key 10. Then the sequential search continues to the neighboring leaf node on the left by pointer l. In this leaf node objects x_T and x_Z for key 9 are obtained. Then the processing of the range query finishes because the next key in the leaf node is 8 and it is lower than the lower bound of the range query. This example shows that the range query requires only one execution of the random search. Afterwards it uses only the sequential search.

5.2 Multidimensional indexing by reduction of dimensionality

Similarly to the previous sections, we propose the multidimensional data as a set of objects X with m attributes $A_1, ..., A_m$. This order of the attributes is fixed. We understand an ordered set $(k_1, ..., k_m)$ as a key value, where $k_i \in D_{A_i}^{act}$. Thus each object $x \in X$ is indexed in such a data structure according to a multidimensional key $(k_1, ..., k_m)$, where each k_i is the value of attribute A_i of object x, i.e., $k_i = A_i(x)$. We perceive the key $(k_1, ..., k_m)$ as a sequence of attribute values of an object, which uniquely determines the position of the object in the data structure, e.g., in the MDB-tree.

In this thesis, we focus on the use of the multidimensional B-tree [OS81, SO82], which we denote shortly as *MDB-tree*. The MDB-tree is a multiway balanced search tree composed of B-trees. By using the MDB-tree, it is possible to index the objects according to more attributes simultaneously in one data structure. In Section 5.2.1, we also introduce the multiple attribute tree data structure [GM80], which we denote shortly as the *MA-tree*. The MA-tree is a multiway search tree which enables us to index the objects according to more attributes simultaneously. We focus on the MA-tree because the main organization of the nodes in the MDB-tree is adopted from the MA-tree data structure.

In the past 40 years, various variants of multidimensional access methods have been proposed. The most influential tree-based multidimensional access methods have been developed at the end of the seventies and in the early eighties. Most of these tree structures as the R-tree [Gut84], the K-D-tree [Ben75] or the K-D-B-tree [Rob81] are based on the concept of multidimensional 'divide and conquer'. In all these approaches, the multidimensional domain is recursively divided into smaller regions of the same dimensionality, and the results from these smaller regions are combined to produce the answer to the problem.

But the MA-tree and also the MDB-tree are based on a different concept, where the problem of the multidimensional domain is solved by gradual reduction of the dimensionality of the multidimensional domain.

5.2.1 Multiple attribute tree

When we suppose that the objects have m attributes, the MA-tree has m levels and values of one particular attribute are stored only in one corresponding level of the MA-tree. Values of A_1 are stored on the first level, values of A_2 are stored on the second level and so on. Thus, the order of the

levels and the attributes is fixed. More formally, we adopted the following definition of the MA-tree based on recursion from [RIM85]. The MA-tree and its construction and properties are discussed exhaustively in [GM80].

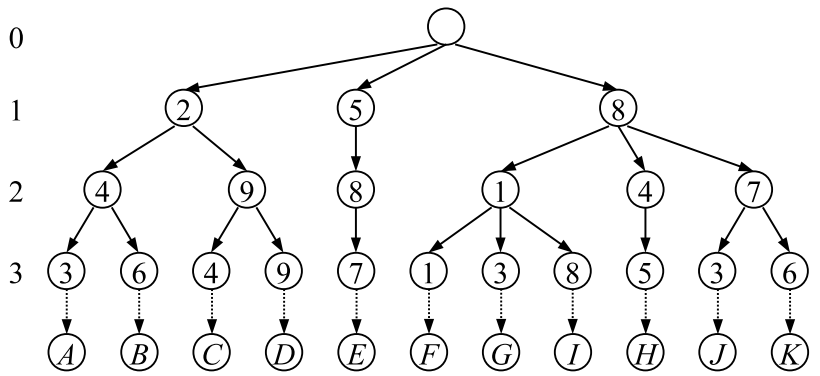
Definition 5.3a (MA-tree) *An m-dimensional MA-tree on m attributes $A_1, ..., A_m$ for a set of objects is defined as a tree of depth m, with the following properties:*

1. *It has a root.*
2. *The child nodes of the root are in ascending order according to their values of the first attribute A_1.*
3. *Each child of the root is a root of an $(m-1)$-dimensional MA-tree on attributes $A_2, ..., A_m$, which contains the subset of objects included in the m-dimensional MA-tree that have the same value of attribute A_1 as the value of the root of the $(m-1)$-dimensional MA-tree.*
4. *An 1-dimensional MA-tree has one child node containing an object.*

Figure 5.4 depicts the MA-tree, in which the set of eleven objects with three attributes is stored. The same set of objects is stored in the table on the right side of the figure. Because of the properties of Definition 5.3a, there is an additional level zero, which includes a dummy node.

From Definition 5.3a, we observe that the root is at level zero and the m attributes correspond to the next m levels of the MA-tree. The root node does not have a value associated with it. All the other nodes of the MA-tree are associated with the corresponding attribute values. The MA-tree is constructed recursively, so that all objects with the same values of attributes $A_1, ..., A_i$ are combined into a node on the ith level of the MA-tree. We can perceive this node as a root of the MA-tree of depth $m - i$, which is a subtree of the main m-dimensional MA-tree. The $(m-i)$-dimensional MA-tree includes values of attributes $A_{i+1}, ..., A_m$ of such objects $x \in X$ that have the same first i values of attributes A_1, ..., A_i, which are included in parent nodes of the node in the ith level with the attribute A_i value in the main MA-tree.

One of the notable features of the MA-tree is that each distinct combination of the values of m attributes, i.e. each different multidimensional key (k_1, ..., k_m) is represented by a unique path from the root to a terminal node. Every terminal node contains an object (or object identifier) or pointer to an object. If we suppose more objects with the same values of all the attributes, i.e., the object with the same multidimensional key (k_1, ..., k_m), we can store the object in the object array. Then the pointer to the object array can be used instead of the pointer to only one object.



	A_1	A_2	A_3
A	2	4	3
B	2	4	6
C	2	9	4
D	2	9	9
E	5	8	7
F	8	1	1
G	8	1	3
H	8	1	8
I	8	4	5
J	8	7	3
K	8	7	6

Figure 5.4: The multiple attribute tree data structure (MA-tree), in which eleven objects are indexed according to their values of three attributes.

5.2.2 Filial set

For better understanding of the MDB-tree, we introduce the *filial set* of attribute values. For each key $k_i \in D^{act}_{A_i}$ for attributes $A_1, ..., A_{m-1}$ the filial set includes some values from $D^{act}_{A_{i+1}}$. More formally, we define the filial set by the following definition.

Definition 5.3b (filial set of key) *Let X be a set of objects with m attributes in a fixed order $A_1, ..., A_m$. Let k_i be a value of the ith attribute and X^{k_i} be such a subset of X that all objects $x \in X^{k_i}$ have values of attributes $A_1, ..., A_i$ the same and their value of attribute A_i is equal to k_i. Then the set of values of attribute A_{i+1} appearing in all objects $x \in X^{k_i}$ is called a filial set of key k_i.*

In case of the MA-tree, a filial set is a set of nodes, at the same level, which have the same parent. For example in the MA-tree in Figure 5.4, the filial set of key $k_1 = 2$ (node including value 2 on the first level) is constituted by nodes including value 4 and value 9 on the second level of the MA-tree, i.e., the filial set is $\{4, 9\} \in D^{act}_{A_2}$. According to Definition 5.3b, the corresponding subset of objects X^{k_1} contains objects x_A, x_B, x_C and x_D.

5.3 Multidimensional B-tree

The MDB-tree is a multiway balanced search tree composed of B-trees. The MDB-tree adopts its main organization of the nodes from the MA-tree and each node of the MDB-tree is B-tree.

The MDB-tree allows to index a set of objects according to m attributes $A_1, ..., A_m$ simultaneously. In this case, the MDB-tree has m levels. Values of one attribute are stored in one corresponding level. The order of levels of MDB-tree is fixed. Values of attribute A_1 are stored on its first level, values of attribute A_2 are stored on its second level and so on.

Analogously to the MA-tree, the MDB-tree is constructed recursively so that all objects with the same values of attributes $A_1, ..., A_i$ are combined into nodes on its ith level. These nodes are constituted of B-trees. To explain the MDB-tree, we utilize the term of the filial set introduced by Definition 5.3b and the B-tree introduced by Definition 5.1a. We define MDB-tree by the following definition, which is derived from the original work [SO82].

Definition 5.3c (MDB-tree) *A MDB-tree is a multiway balanced search tree, which satisfies the following properties:*

1. *The MDB-tree has m levels, where m is the number of indexed attributes. Each ith level of the MDB-tree corresponds to attribute A_i.*

2. *Nodes of the MDB-tree are (non-redundant) B-trees. Each B-tree in ith level includes only the values of attribute A_i.*

3. *Nodes of particular B-trees of order n on the ith level are organized as follows:*

   $$p_0, (k_1, p_1, f_1), (k_2, p_2, f_2), ..., (k_n, p_n, f_n),$$

   where $k_1 < k_2 < ... < k_n$ are key values and p_0, p_1, ..., p_n are pointers to children nodes in the same B-tree.

   - *For $i < m$, each f_j is a pointer to the root of a B-tree on the $(i+1)$th level of the MDB-tree, which includes the values of A_{i+1} constituting the filial set of key k_j.*
   - *For $i = m$, each f_j is a pointer to the associated data.*

4. *All filial sets on level $i > 1$ are linked by a pointer NEXT included in the root of the B-tree, which points to the root of the following B-tree. The order of these B-trees corresponds with the order of the keys on the $(i-1)$th level.*

5. *For each level i of the MDB-tree, an entry point LEVEL(i) is provided, which points to the beginning of each such linked list on the ith level.*

6. *Each root node of the B-tree on the ith level includes the pointers LEFT and RIGHT, which point to the root of the leftmost B-tree and the root of the rightmost B-tree, respectively, among the B-trees on the $(i+1)$th level, which are children of the B-tree on the ith level.*

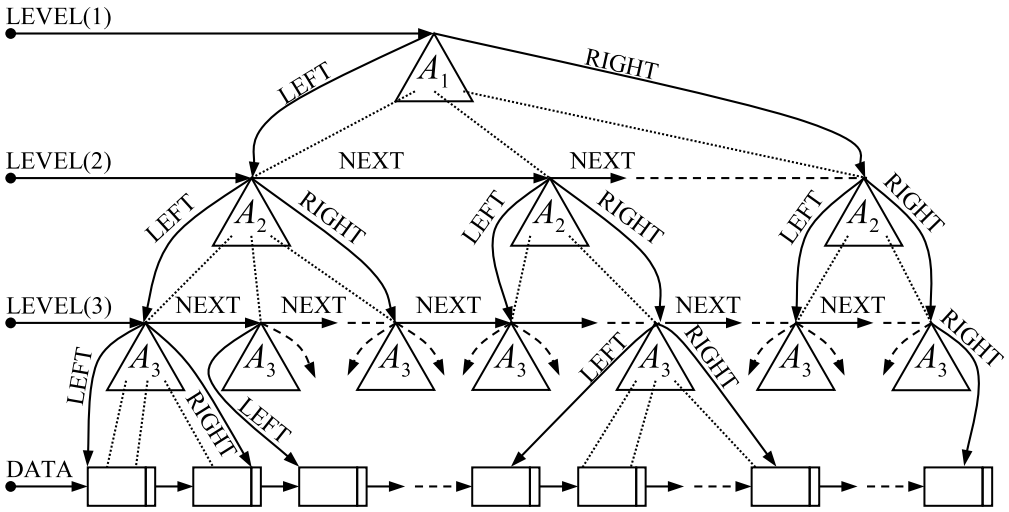


Figure 5.5: The multidimensional B-tree (MDB-tree) [SO82], in which a data set is indexed according to three attributes.

According to Definition 5.3c, only one B-tree is situated on the first level of the MDB-tree (see Figure 5.5). The B-tree contains all values from $D^{act}_{A_1}$ as the keys in its nodes. Each such key k_1 has its corresponding pointer to a B-tree on the second level of the MDB-tree. This B-tree includes every key $k_2 \in D^{act}_{A_2}$ for which such an object $x \in X$ exists that $A_1(x) = k_1$ and $A_2(x) = k_2$. All such keys k_2 form the filial set of k_1. Thus, a B-tree on the second level of the MDB-tree may not include all the values of $D^{act}_{A_2}$ as its keys. On the other hand, all the B^+-trees on the second level of the MDB-tree together contain all values of $D^{act}_{A_2}$. In general, a B^+-trees on the ith level of MDB-tree, where $i > 1$, may not include all the values of $D^{act}_{A_i}$, but each value of $D^{act}_{A_i}$ has to occur in at least one of the B-trees on the ith level of the MDB-tree.

Figure 5.5 depicts the graphical representation of the MDB-tree introduced in [SO82], where the authors noted that the associated data can be interpreted as filial sets residing at level $m + 1$. Hence, the linking and ordering definitions for the filial sets can be applied also for the list of associated data. Therefore, the pointers LEFT and RIGHT are included also in the roots of B-trees on the mth level of the MDB-tree and refer to the corresponding positions in the linked list of the associated data.

5.3.1 Dynamic updates

Since the MBD-tree is composed of B-trees, it is a dynamic data structure. The B-trees are always height-balanced and have the logarithmic complexity of update operations (see Section 5.1). Therefore, operations as search, insert

or delete of an object in the MBD-tree require low-cost maintenance. In the original work [OS81], the authors have developed algorithms which maintain the MDB-tree structure when insertions and deletions occur. Both update algorithms are easy to implement. For better understanding of the MDB-tree we provide the folloving sketch of the insert and delete algorithm.

Insert

Let x be an object intended to be inserted with key (k_1, ..., k_m), i.e., values $k_1, ..., k_m$ of attributes $A_1, ..., A_m$, respectively. Let us suppose that the combination $k_1, ..., k_{i-1}$ of key values already exists in the MBD-tree, but $k_1, ..., k_{i-1}, k_i$ does not exist. Because keys $k_1, ..., k_i$ occur in B-trees from the first level to the ith level, we do not need to add the keys into them.

We first insert k_i into the B-tree at level i determined by the combination $k_1, ..., k_{i-1}$ of keys as a unique path from the root to the B-tree. Then a filial set of key k_i consisting of the single value k_{i+1}, must be inserted as a new B-tree at level $i + 1$. If the value of k_i is inserted between the keys k_i^{LO} and k_i^{HI} in the B-tree on the ith level, the filial set of key k_i as B-tree T_i must be inserted between the filial sets as B-trees T_i^{LO} and T_i^{HI} at level $i+1$ associated with keys k_i^{LO} and k_i^{HI}. Only one additional access is necessary in a case that the pointer NEXT in the root of T_i^{LO} has to be changed to refer to a new B-tree T_i. The pointer NEXT in the root of T_i has to refer to B-tree T_i^{HI}. We note that the situation is slightly more complicated if k_i becomes the smallest value in its corresponding B-tree.

Analogously, we continue for the rest of the keys $k_{i+2}, ..., k_m$ by inserting new filial sets as B-trees on levels $i + 2, ..., m$ of the MDB-tree. Finally, we insert the object x as a new record into the linked list of associated data. Additionally, we have to maintain the pointers LEFT and RIGHT of each newly created B-tree. It is straightforward because both pointers are referring to the same new filial set, i.e., to the root of the new B-tree on the next level, or to the new record in case of the last level of the MDB-tree.

Delete

The delete algorithm is straightforward. In contrast to the insert algorithm, first we delete the record in the linked list of associated data and then we delete the unnecessary keys starting from the last level towards the first level of the MDB-tree. The main effort of the algorithm is to maintain the correct references between the nodes of the MDB-tree. We have to update the pointers NEXT, LEFT and RIGHT in each root of the B-tree which has been affected by the delete algorithm.

The authors of [OS81] have also shown that the performance of the insert and delete algorithms has the logarithmic complexity in an average case and also in the worst case. Moreover, there is no need to reorganize the MDB-tree, as in the case of the MA-tree [GM80], or to perform a new optimization, as may be required with the K-D-tree [Ben75]. The MDB-tree can be maintained with relatively low cost when any dynamic changes occur. This is the why we focus on the MDB-tree as on a promissing data structure for the top-k search.

5.3.2 Associative queries

The MDB-tree as an index method is well suited to handle several types of associative queries efficiently, since its nodes are B-trees linked by the pointers NEXT, LEFT, RIGHT and by the additional pointer LEVEL to each level of the MDB-tree. The authors of [OS81] consider search strategies and search algorithms, which ensures an efficient search for the most common types of queries in the MDB-tree.

In the following text we sketch the processing of three types of associative queries, which are classified as follows:

- *exact match query*, which specifies a value for each of m indexed attributes,
- *partial match query*, which specifies values for $d < m$ of the indexed attributes, and
- *range query*, in which a range, i.e. an interval $[w_i, u_i]$, is specified for each indexed attribute.

Exact match query

The processing of the exact match query is straightforward. For a multidimensional key (k_1, ..., k_m), we gradually search for each particular key k_i on the corresponding B-tree on the ith level, starting in the first level.

Partial match query

The processing of the partial match query is executed analogously to the processing of the exact match query in case of specified values of the indexed attributes. In case of unspecified attribute values, it uses additional pointers NEXT, LEFT and RIGHT included in root of each B-tree of the MDB-tree and the pointer LEVEL to each level of the MDB-tree.

Let us suppose that we have a partial match query, where we require such objects that have attribute values k_1, ..., k_{i-1}, k_{i+1}, ..., k_m, i.e., value k_i of attribute A_i does not matter. According to keys k_1, ..., k_{i-1} we find the B-tree, which corresponds to filial set of the key k_{i-1}, on the ith level of the MDB-tree. We utilize the pointers LEFT and RIGHT of the B-tree, which reffer to the B-trees T_{i+1}^{LEFT} and T_{i+1}^{RIGHT} on the $(i+1)$th level of the MDB-tree.

Since the value of k_i is irrelevant, we skip the B-tree by its pointer LEFT to B-tree T_{i+1}^{LEFT}. We search for the required objects according to keys $k_{i+1}, ..., k_m$ as in case of an exact match query, starting from the B-tree on the $(i+1)$th level towards the last level of the MDB-tree. The obtained objects are appended into the result of the query. Then we move by pointer NEXT to the next B-tree on the $(i+1)$th level and search for further required objects starting from this B-tree. This action is repeated, the following B-trees on the $(i+1)$th level are processed one by one, until the B-tree T_{i+1}^{RIGHT} is processed.

Let us suppose that we have a partial match query, where we require objects that have attribute values k_{i+1}, ..., k_m, i.e., it does not matter what the first i values k_1, ..., k_i of attributes A_1, ..., A_i are. We use the pointer LEVEL$(i+1)$ referring to the first B-tree on the $(i+1)$th level of the MDB-tree. In this B-tree we start the search. By using the NEXT pointers we go through all the B-trees on the $(i+1)$th level of the MDB-tree and we perform the search in each one of them.

Range query

The processing of the range query can be perceived as an extension of the partial match query. Therefore, to explain the range query processing we sketch the main difference from the partial match query.

Let us suppose that we have a range query $[w_1, u_1] \times ... \times [w_m, u_m]$, which specifies required values for each ith attribute by the interval $[w_i, u_i]$, where w_i and u_i are the bounds for attribute A_i. Let a B-tree on the ith level of the MDB-tree be processed at some moment of the execution of the range query.

Again, the order of the filial sets on a given level and the corresponding pointers allow us to avoid the retrieval in a given B-tree at level i of all attribute values between w_i and u_i. Thus only two attribute values, as keys W and U, have to be retrieved from the B-tree. Key W is equal either to w_i or to the closest value superior to w_i and, similarly, key U is equal either to u_i, or to the closest value inferior to u_i. We note that the execution do not continue in the B-tree, if either one of the two keys W and U cannot

be defined. The corresponding pointers f_W and f_U of the keys W and U point to the B-tree T_W and the B-tree T_U on the $(i+1)$th level of the MDB-tree. These B-trees T_W and T_U constitute the filial sets of keys W and U, respectively. Then the execution of the range query continues by processing of the B-trees on the $(i+1)$th level.

Analogously to the partial match query for irrelevant attribute value, we move by pointer f_W to the B-tree T_W and by using the NEXT pointers we go through all the B-trees on the $(i+1)$th level, and we perform the search in each of them until we perform the search in the B-tree T_U determined by pointer f_U, i.e NEXT and f_U point to the same B-tree. In other words, in contrast to the partial match query for irrelevant attribute value, pointers f_W and f_U were used instead of the pointers LEFT and RIGHT.

To conclude this sketch, we note that authors of [SO82] propose the processing of partial match and range queries by means of breadth-first search. Consequently, they suppose one queue holding the attribute values which qualify at the currently searched level and another queue holding the pairs of pointers delimiting the portions of linked lists of B-trees to be searched at the next specified level.

5.4 Multidimensional B-tree composed of B$^+$-trees

The authors of the MDB-tree originally considered in [SO82] that the nodes of the MDB-tree are the B-trees, but their solution can be readily extended to include some of these variants of the B-trees. Many variants of B-trees have been proposed (see, e.g., [Com79]), such as the redundant B-tree, B$^+$-trees or B*-trees, where each node is at least $\frac{2}{3}$ full instead of just $\frac{1}{2}$ full compared to the B-tree.

For the purposes of this thesis, it is the most suitable to use an extension based on B$^+$-trees, which we proposed in [OP08]. Therefore, we introduce a variant of the MDB-tree composed of B$^+$-trees, in which the leaf nodes are linked in two directions (see double linked B$^+$-tree in Definition 5.1d). The main organization of the nodes of the variant is the same as in the case of the MDB-tree, but the double linked B$^+$-trees are used as the nodes of the MDB-tree instead of the B-trees.

More formally, we define the MDB-tree composed of B$^+$-trees by the following definition. For the purposes of the definition, we denote this variant of the MDB-tree as the *MDB$^+$-tree*, but in the rest of this thesis we denote the MDB$^+$-tree simply as the MDB-tree because we use this variant.

Definition 5.4d (MDB$^+$-tree) *MDB$^+$-tree is a multiway balanced search tree, which satisfies the following properties:*

1. *MDB$^+$-tree has m levels, where m is the number of indexed attributes. Each ith level of the MDB$^+$-tree corresponds to attribute A_i.*

2. *Nodes of the MDB$^+$-tree are double linked B$^+$-trees. Each B$^+$-tree on the ith level includes only values of attribute A_i.*

3. *The leaf nodes of particular B$^+$-trees on the ith level are organized as follows:*

$$l, (k_1, f_1), (k_2, f_2), ..., (k_n, f_n), r,$$

   where $k_1 < k_2 < ... < k_n$ are key values, l is a pointer to the left neighboring leaf node in the same B$^+$-tree and r is a pointer to the right neighboring leaf node in the same B$^+$-tree.

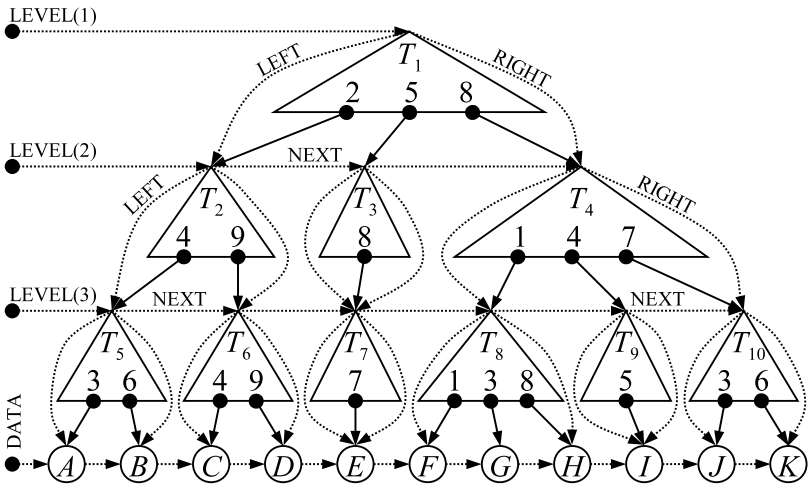
   - *For $i < m$, each f_j is a pointer to the root of the B$^+$-tree on the $(i+1)$th level of the MDB$^+$-tree, which includes values of A_{i+1} constituting the filial set of key k_j.*
   - *For $i = m$, each f_j is a pointer to the associated data.*

4. *All filial sets at level $i > 1$ are linked by the pointer NEXT included in the root of the B$^+$-tree, which points to the root of the following B$^+$-tree. The order of these B$^+$-trees corresponds with the order of the keys on the $(i-1)$th level.*

5. *For each level i of the MDB-tree, an entry point LEVEL(i) is provided, which points to the beginning of each such linked list on the ith level.*

6. *Each root node of the B-tree on the ith level includes the pointers LEFT and RIGHT, which point to the root of the leftmost B-tree and the root of the rightmost B-tree, respectively, among the B-trees on the $(i+1)$th level, which are children of the B-tree on the ith level.*

As in the case of the B$^+$-tree, we introduce an object array. If we suppose a set of objects with arbitrary values of an attributes A_1, ..., A_m, some objects can have the same value of all the attributes, i.e., objects with the same multidimensional key (k_1, ..., k_m). We perceive the object array as external associated data stored outside of the MDB$^+$-tree. For each multidimensional key $(k_1, ..., k_m)$ appearing in the MDB$^+$-tree we use one object array, which includes all objects with values $k_1, ..., k_m$ of attributes $A_1, ..., A_m$, respectively. We note that only the pointers in the leaf nodes of B$^+$-trees, which are on the last level of the MDB$^+$-tree, point to the object arrays.



	A_1	A_2	A_3
A	2	4	3
B	2	4	6
C	2	9	4
D	2	9	9
E	5	8	7
F	8	1	1
G	8	1	3
H	8	1	8
I	8	4	5
J	8	7	3
K	8	7	6

Figure 5.6: The MDB$^+$-tree composed of doule linked B$^+$-trees.

Figure 5.6 depicts a set of eleven objects with their values of three attributes A_1, A_2, A_3 stored in the MDB-tree composed of double linked B$^+$-trees. The same set of objects is stored in the table on the right side of the figure. The objects arrays are depicted as circles. For the sake of clarity, each object array includes only one object. In the case of three indexed attributes, the MDB-tree has three levels. On the first level only one B$^+$-tree T_1 is situated, which includes all values from $D^{act}_{A_1}$ as its keys. Analogously, the second level is composed of B$^+$-trees T_2, T_3, T_4, which include keys from $D^{act}_{A_2}$, and the third level is composed from B$^+$-trees T_5, T_6, T_7, T_8, T_9, T_{10}, which include keys from $D^{act}_{A_3}$. For better understanding of the MDB-tree and MDB$^+$-tree, we provide a short example of two associative queries.

Let us suppose a partial match query, which requires all the objects with value $k_1 = 8$ of attribute A_1 and value $k_3 = 3$ of attribute A_3. First, in B$^+$-tree T_1 we find the key with value 8 and we move to the root node of B$^+$-tree T_4. Because the value of attribute A_2 is not defined, we skip to the third level of the MDB$^+$-tree by the pointer LEFT of B$^+$-tree T_4 and by the pointers NEXT we search for the required objects in B$^+$-trees T_8, T_9 and, finally, T_{10}, to which the pointer RIGHT of B$^+$-tree T_4 refers. Thus, from B$^+$-trees T_8 and T_{10} we obtain the objects x_G and x_J with value $k_3 = 3$ of attribute A_3.

Let us suppose a partial match query, which requires all the objects with value $k_2 = 4$ of attribute A_2. First, we use the pointer LEVEL(2) to B$^+$-tree T_2 because the value of attribute A_1 is not defined. By the pointers NEXT we search for the required objects with value $k_2 = 4$ of attribute A_2 in all B$^+$-trees on the second level of the MDB$^+$-tree. The B$^+$-trees on the third

level are skipped. Finally, we obtain the objects x_A, x_B and x_I.

5.4.1 Degenerated multidimensional B-tree

In this thesis, we perceive a multidimensional data set as a set of objects with more attributes. In some cases, when a set of object is indexed by using the MDB-tree, some B-trees of the MDB-tree can include only one key. We call a B-tree with one key the *degenerated B-tree*. Analogously, we call the MDB-tree which includes a fairly large amount of degenerated B-trees the *degenerated MDB-tree*.

Now we describe the worst case of a degenerated MDB-tree. For example, we have a MDB-tree with m levels, in which N objects are indexed by values of attributes $A_1, ..., A_n$. Let us assume that each object has a different value of attribute A_1. The B^+-tree on the first level of the MDB-tree contains all values of attribute A_1, but this is valid for every MDB-tree. However, the second level of the MDB-tree has to contain N B^+-trees, one for each value of attribute A_1, and thus for each object because each object has a different value of attribute A_1. In this case each B^+-tree on the second level of the MDB-tree is degenerated because it contains only one key, which is the value of attribute A_2 of the corresponding object. Then the filial set of this key is only value of the next attribute A_3 of the corresponding object. The same holds for the remaining levels of the MDB-tree. All levels of the MDB-tree except the first level contain N degenerated B^+-trees. We can say that this worst case of a degenerated MDB-tree has a shape of a "string curtain".

5.5 Multidimensional B-tree with lists

In general, particular attribute domains of the data set can be different. There are several attribute types according to the character of their attribute domain. Moreover, for a given data set, the sizes of actual domains of particular attributes can be different. In this thesis, we differentiate between categorical attribute and numerical attribute as the two basic attribute types (see Section 3.3.2). We perceive the categorical attribute as an attribute with a small-sized domain and the numerical attribute as an attribute with a big-sized domain or with an infinite range of allowable values. For example, the price or the distance is usually a nominal attribute and the brand of a product or city name is a categorical attribute. A nominal attribute represents a quantity of something, e.g., in this case a square meter and money. Other typical examples of nominal attributes are latitude or longitude in some geographical data.

Example 5.5 We have a real data set containing 8 822 flats for rent in Prague at our disposal. These flats for rent have four important attributes, namely, *District*, *Type*, *Area*, and *Price*, according to which we want to index them. Each attribute has the size of its actual domain as follows:

$$\|D^{act}_{District}\| = 10, \quad \|D^{act}_{Type}\| = 10, \quad \|D^{act}_{Area}\| = 229, \quad \|D^{act}_{Price}\| = 411.$$

It is straightforward that *District* and *Type* are the categorical attributes because they have small-sized actual domains. *Area* and *Price* have big-sized actual domains and they are numerical attributes.

In real data sets, the occurrence of more numerical attributes is very common. Let us suppose a set of object has more numerical attributes with actual attribute domains including large numbers of different attribute values. If we index the set of objects using the MDB-tree, the MDB-tree can become a degenerated MDB-tree.

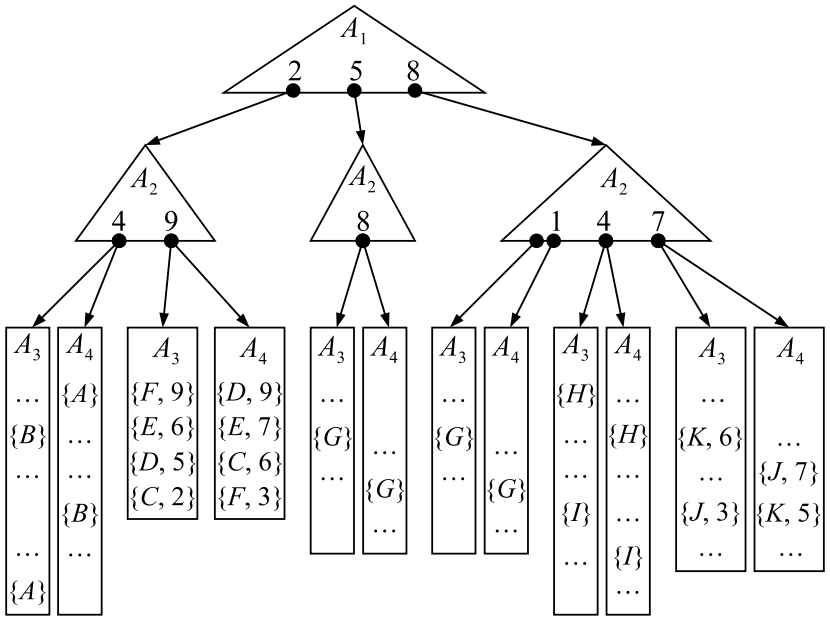
Therefore, we developed a new data structure in [OP09, OP10], the *MDB-tree with lists*, which is well suited for indexing the set of objects with more categorical attributes and more numerical attributes. The use of the MDB-tree with lists can partially solve the problem of the degenerated MDB-tree. The MDB-tree with lists is composed of the MDB-tree and Fagin's sorted lists (see Section 4.1.1).

For better understanding of the MDB-tree with lists, we introduce the following definition as an extension of Definition 5.3b of the filial set.

Definition 5.5a (filial objects and filial attribute values) *Let X be a set of objects with m attributes $A_1, ..., A_m$, where $A_1, ..., A_n$ are categorical attributes and $A_{n+1}, ..., A_m$ are numerical attributes. Let k_n be a value of the nth attribute and X^{k_n} be such a subset of X that all objects $x \in X^{k_n}$ have the same values of the first n attributes $A_1, ..., A_n$ and their values of attribute A_n are equal to k_n.*

- *We call the objects from X^{k_n} the filial objects of key k_n.*
- *We call the set of values of attribute A_i, where $i = n+1, ..., m$, appearing in X^{k_n} the filial attribute A_i values of key k_n.*

In the MDB-tree with lists the categorical attributes $A_1, ..., A_n$ are stored as the MDB-tree with n levels and numerical attributes $A_{n+1}, ..., A_m$ are stored as groups of $m - n$ Fagin's sorted lists. These $m - n$ lists $L_{n+1}, ..., L_m$ include data items composed of an *object-value pair*, i.e., object x (or identifier of object x) and its value $A_i(x)$ of corresponding attribute A_i, which we denote $\{x, A_i(x)\}$.



	A_1	A_2	A_3	A_4
A	2	4	1	8
B	2	4	6	2
…	…	…	…	…
C	2	9	1	4
D	2	9	3	9
E	2	9	5	7
F	2	9	8	1
…	…	…	…	…
G	8	1	4	3
…	…	…	…	…
H	8	4	5	1
I	8	4	8	9
…	…	…	…	…
J	8	7	3	7
K	8	7	6	5
…	…	…	…	…

Figure 5.7: The MDB-tree with lists in which two categorical attributes A_1 and A_2 are stored as in the MDB-tree and two numerical attributes A_3 and A_4 stored as in Fagin's lists.

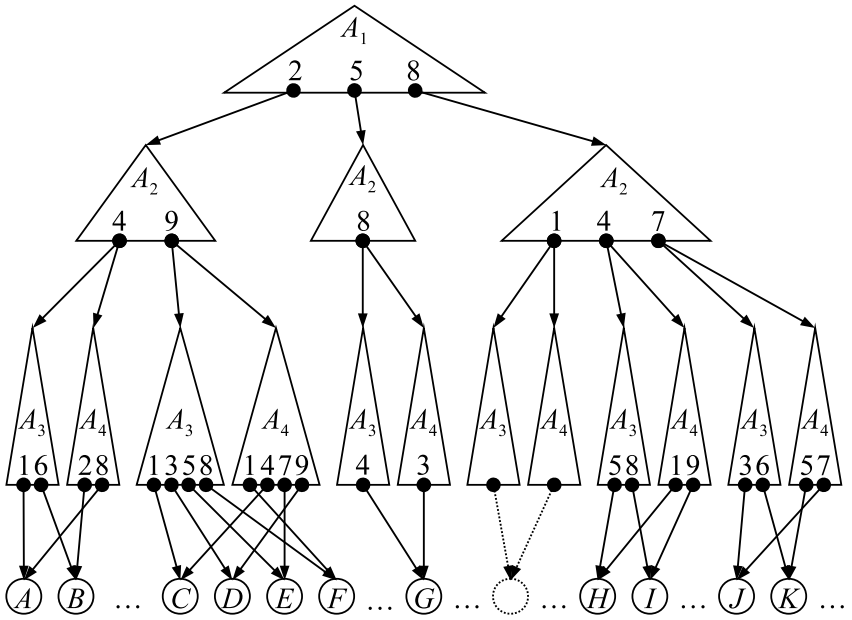
Instead of each filial set of key $k_n \in D^{act}_{A_n}$, there is a group of $m-n$ Fagin's sorted lists. Instead of pointer f_n of key k_n, there are $m-n$ pointers $f_n^{n+1}, ..., f_n^m$, which point to lists $L_{n+1}, ..., L_m$, respectively. For key k_n, each list L_i of the group, where $i = n+1, ..., m$, includes the filial objects of key k_n and the filial attribute A_i values of key k_n, which constitute the sorted list of the object-value pairs.

Figure 5.7 depicts a set of objects indexed in the MDB-tree with lists in which two categorical attributes are stored as the MDB-tree with two levels and two numerical attributes are stored as groups of Fagin's lists. The corresponding objects are depicted in the table on the right side of Figure 5.7.

5.5.1 Multidimensional B-tree with groups of B$^+$-trees

In our research we discovered that in some cases it is useful to avoid the use of Fagin's sorted lists. Therefore, in [OP09, OP10] we developed the *MDB-tree with groups of the B$^+$-trees* as an extension of the MDB-tree with lists, where each Fagin's sorted list is substituted by the B$^+$-tree.

This new data structure can be maintained with relatively low cost when any dynamic changes occur because it is composed only of the B$^+$-trees. Moreover, the MDB-tree with groups of the B$^+$-trees enables us to use the



	A_1	A_2	A_3	A_4
A	2	4	1	8
B	2	4	6	2
...	...	...	...	...
C	2	9	1	4
D	2	9	3	9
E	2	9	5	7
F	2	9	8	1
...	...	...	...	...
G	8	1	4	3
...	...	...	...	...
H	8	4	5	1
I	8	4	8	9
...	...	...	...	...
J	8	7	3	7
K	8	7	6	5
...	...	...	...	...

Figure 5.8: The MDB-tree with the B$^+$-trees in which two categorical attributes A_1 and A_2 are stored as in the MDB-tree and two numerical attributes A_3 and A_4 stored as the groups of the B$^+$-trees.

model of user preferences based on the local user preferences introduced in this work for the top-k search (see Section 8.4.1).

The first n categorical attributes are stored as in the MDB-tree. The numerical attributes are stored as groups of the B$^+$-trees. Instead of each filial set of key $k_n \in D^{act}_{A_n}$ (in the original MDB-tree), there are $m - n$ B$^+$-trees $T_{n+1}, ..., T_m$. Instead of pointer f_n of key k_n (on the nth level of the MDB-tree), there are $m - n$ pointers $f_n^{n+1}, ..., f_n^m$, which point to B$^+$-trees $T_{n+1}, ..., T_m$, respectively. For key k_n, each B$^+$-tree T_i of the group where $i = n + 1, ..., m$, includes the filial objects of key k_n indexed according to the filial attribute A_i values of key k_n.

Figure 5.8 depicts the MDB-tree with B$^+$-trees, in which two categorical attributes are stored as an MDB-tree with two levels and two numerical attributes are stored as groups of B$^+$-trees. For each key k_2 in the leaf node of every B$^+$-tree on the second level of the MDB-tree, the group of B$^+$-trees corresponding to key k_2 includes two B$^+$-trees. The first B$^+$-tree includes filial objects indexed according to the filial values of attribute A_3 of key k_2. The second B$^+$-tree includes the same set of filial objects indexed according to the filial values of attribute A_4 of key k_2.

Chapter 6

Support of Local User Preferences Based on B$^+$-tree

In this chapter, we introduce a new model of a sorted list, which enables us to access a set of objects with regard to local user preference for an attribute. Our main motivation is that Fagin's algorithms do not support local preferences directly because they use a model based only on a monotone aggregate function and sorted lists. The model is based on a B$^+$-tree, by which it is possible to obtain the objects sequentially in descending order according to an arbitrary fuzzy function, which represents a local user preference for an attribute of the objects [OP08, HOP11]. Initially, we adopted concept from Fagin et. al [FKS03] and the approach from [EPV07, GVP08]. In this chapter, we redefine the theoretical background of the model. Thus, it is possible to apply the model in Web environment.

In Section 6.1, we introduce the abilities of a B$^+$-tree and explain the principle of the sequential access into the B$^+$-tree according to a fuzzy function. In Section 6.2, we discuss the application of the new model of the sorted list in Fagin's algorithms, which makes it possible to apply the local user preferences directly during the computation of Fagin's algorithms. In Section 6.3, we focus on the use of this model of the sorted list and Fagin's algorithms in Web environment, where the values of each attribute are stored on different remote servers on the Internet. In Section 6.4 we show experimental results. In Section 6.5 we provide a summary.

6.1 Sorted list based on B$^+$-tree

In Section 4.1.1, we described a sorted list, which contains all the objects from a set of objects X with values of an attribute A. The sorted list includes

data items composed of an object-value pair $\{x, A(x)\}$, which are sorted in descending order according to values of $A(x)$. The sorted list is processed by the sorted access (see Section 4.1.2), which obtains the first pair $\{x_1, A(x_1)\}$ with the highest value of $A(x_1)$, then the second pair $\{x_2, A(x_2)\}$ with the second highest value of $A(x_2)$ and so on. Thus, the model of the sorted list enables us to obtain the objects sequentially by using the sorted access in descending order according to values from the attribute domain D_A.

In this work, we suppose the local preference of user U for an attribute A as a fuzzy function f^U (see Section 3.3). If we want to enable Fagin's algorithms to find the top-k objects with the support of local preferences, we need to use a new model of sorted list which enables the sorted access to objects with regard to a local preference of user U. Therefore, there is a requirement to access the objects $x \in X$ sequentially in descending order according to their values of $f^U(x)$. Moreover, when we suppose a data item of the sorted list as an object-value pair, then we need to obtain the objects one by one in the form of pairs $\{x, f^U(x)\}$. The new model has to be able to provide the first pair $\{x_1, f^U(x_1)\}$ with the highest value of $A(x_1)$, then the second pair $\{x_2, f^U(x_2)\}$ with the second highest value of $A(x_2)$ and so on. In order to do so, we used the B$^+$-tree,which provides this new model of the sorted list.

6.1.1 Abilities of B$^+$-tree

In the B$^+$-tree, objects from X are indexed according to the values of one particular attribute A. We use a variant of the B$^+$-tree, the leaf nodes of which are linked in two directions (see Definition 5.1d). Every leaf node of the B$^+$-tree contains pointers to its left and right neighbour (see Figure 6.1). All the keys, i.e., all different values from D_A, are included in the leaf nodes of the B$^+$-tree. Associated data is referred to only from the leaf nodes (see Section 5.1.3).

Every object $x \in X$ is indexed by the key $k \in D_A$, a value of which is equal to the value of attribute $A(x)$ of the object x. We can perceive an object $x \in X$ as data associated by key $k = A(x)$ In general, different objects with the same value of attribute A can exist in X. Therefore, we use an object array, in which objects with the same value of key k are stored. For every key k there is a pointer to its object (or object array). In the B$^+$-tree, it is possible to find an object according to its key k. We call this ability of the B$^+$-tree the *random search*.

We assume that the keys in the leaf nodes of the B$^+$-tree are in ascending order from the left to the right. Since the leaf nodes are linked in two directions, it is possible to cross the B$^+$-tree through the leaf level and to get

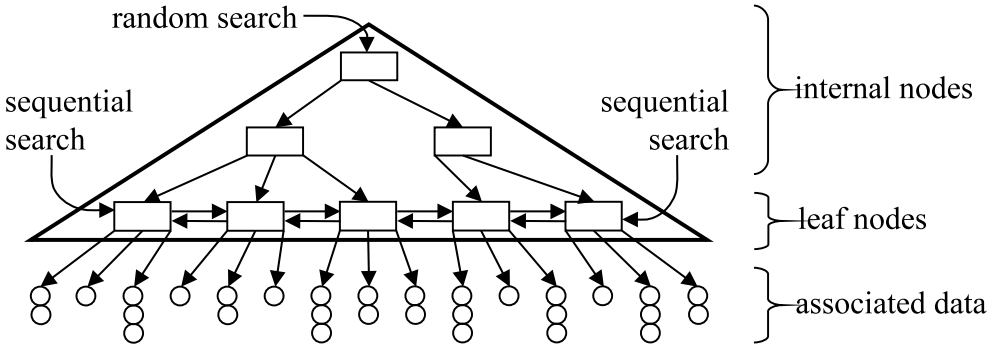


Figure 6.1: The B$^+$-tree, whose leaf nodes are linked in two directions.

all its keys sequentialy in two directions. Additionally, with the pointers to the object arrays and with the pointers betwen nodes in the leaf level, it is possible to get the objects $x \in X$ according to their values of $A(x)$. When we go through the leaf nodes from the left to the right, we can obtain the objects as pairs $\{x, A(x)\}$ sequentially in ascending order according to $A(x)$ and from the right to the left in descending order. This ability of B$^+$-tree we call the *sequential search*.

Figure 6.1 depicts the random search and the sequential search. The random search starts in the root node of the B$^+$-tree and goes through some internal nodes to a leaf node. The sequential search starts in a leaf node and goes through the leaf nodes one by one, using the pointers to their neighbours.

6.1.2 Monotone local preference

Initially in this section, we assume that a local preference of user U for attribute A is given as a monotone fuzzy function f^U. We distinguish two cases of monotone fuzzy function. It can be nondecreasing or nonincreasing on the attribute domain D_A. For the support of the user's local preference in the new model of sorted list, we need to access the objects as pairs $\{x, f^U(x)\}$ sequentially in descending order according to values of $f^U(x)$.

For this purpose, we utilize only the sequential search of the leaf nodes of the B$^+$-tree. The random search is used only for finding the first (the leftmost) leaf node or the last (the rightmost) leaf node in the B$^+$-tree. Moreover, if there are pointers to these outermost leaf nodes (as in Figure 6.1), we do not need to use the random search of any internal nodes of the B$^+$-tree. Then for the nondecreasing f^U the sequential search from the rightmost leaf node to the left is used. For the nondecreasing f^U the sequential search from the leftmost leaf node to the right is used. This principle is formulated by the following statements.

Statement 6.1.2a *Let a set of objects X be indexed in a B^+-tree according to an attribute A and a user's fuzzy function f^U be nondecreasing, i.e.,*

$$\forall x, y \in X : A(x) \leq A(y) \Rightarrow f^U(x) \leq f^U(y).$$

Then by crossing the leaf nodes of the B^+-tree, from the right to the left, it is possible to obtain the pairs $\{x, f^U(x)\}$ sequentially in descending order according to f^U.

Statement 6.1.2b *Let a set of objects X be indexed in a B^+-tree according to an attribute A and a user's fuzzy function f^U be nonincreasing, i.e.,*

$$\forall x, y \in X : A(x) \geq A(y) \Rightarrow f^U(x) \leq f^U(y).$$

Then by crossing the leaf nodes of the B^+-tree, from the left to the right, it is possible to obtain the pairs $\{x, f^U(x)\}$ sequentially in descending order according to f^U.

When we obtain an object x from the B$^+$-tree by the sequential search, we receive only its value $A(x)$, but $f^U(x)$ can be calculated and the pair $\{x, f^U(x)\}$ is created immediately. Since the previous statements hold, it is possible to obtain sequentially the pairs $\{x, f^U(x)\}$ in descending order according to f^U, which is nondecreasing or nonincreasing on the whole D_A.

6.1.3 Arbitrary local preference

In general, a local preference of user U for attribute A as fuzzy function f^U might not be monotone on the attribute domain D_A, which we can perceive as an interval. Therefore, we focus on possibility of obtaining the objects from the B$^+$-tree sequentially according to an arbitrary fuzzy function.

In Section 6.1.2 we showed how we can obtain the objects from the B$^+$-tree in descending order according to monotone f^U. Let a continuous interval I be a subset of D_A and f^U be monotone in this interval. Then we can utilize Statement 6.1.2a or Statement 6.1.2b for obtaining the object according to the monotone f^U by the sequential search of the leaf nodes of the B$^+$-tree. Moreover, we have to find the first object, from which we start the sequential search. We can do it by the random search from the root of the B$^+$-tree.

More exactly, we can suppose an interval $[a, b]$, which is a subset of D_A, and a fuzzy function f^U, which is nonincreasing in this interval. First we use the random search to find the first object and its key (attribute value), which is higher than or equal to a. Then we proceed through the leaf nodes of the B$^+$-tree to the right by the sequential search and we obtain other objects sequentially until their key is lower than (or equal to) b. For the

nonincreasing f^U in this interval, first we use the random search to find the first object, the attribute value of which is lower than b, and then we use the sequential search in the opposite direction until object's key is higher than b.

We note that the random search and the sequential search of the B$^+$-tree are applied in a similar way as in the execution of a range query (see Section 5.1.2). This is the main reason why we use the B$^+$-tree, which has its leaf nodes linked in two directions.

Generally, it is possible to perceive an arbitrary fuzzy function f^U, which is non-monotone in the whole attribute domain D_A, as a function which is monotone in some parts of D_A. Moreover, the domain D_A as an interval can be divided into n continuous disjoint intervals $I_1, ..., I_n$ that f^U is monotone in each of the intervals. This division into these intervals is determined by local maxima and local minima of the fuzzy function f^U on D_A, which identify bounds of these intervals. The following fact holds.

Notice 6.1.3 *If a fuzzy function f^U has $n + 1$ local extrema, then it has to be divided into n intervals, on which f^U is monotone.*

Consequently, global maxima and global minima of f^U have to alternate in its domain D_A. If a fuzzy function is monotone, then it has one such interval and two local extrema, which are also global maximum and global minimum. A fuzzy function with three local extrema has two such intervals and for a fuzzy function with more local extrema the number of intervals increases respectively. For example, the fuzzy function which is depicted in Figure 6.2 has six local extrema, which alternate between local maximum and local minimum. This fuzzy function is therefore divided into five intervals $I_1, ..., I_5$.

Depending on the division of the intervals, the leaf level of the B$^+$-tree can be divided into corresponding parts. Then it is possible to obtain the objects, one by one, in the descending order according to f^U from each part by going through it in a suitable direction. The objects can be obtained according to the f^U concurrently from all these parts of the leaf level of the B$^+$-tree. If we need to obtain the objects one by one, we can use a selection based on the highest value. If there are more object candidates for the next best pair $\{x, f^U(x)\}$, the object x with the highest value of $f^U(x)$ is chosen.

We introduce a *candidate set* C, which contains some objects from the leaf level of B$^+$-tree. The set of candidates C may contain only one object from each interval $I_1, ..., I_n$. For each object $x \in C$ we remember a pointer to the leaf level of B$^+$-tree, its attribute value and the interval $I_i \in I$ to which it belongs. Now we introduce two steps, which describe a method of sequential obtaining of the objects, which are indexed in the B$^+$-tree, according to an arbitrary fuzzy function f^U.

1. Initialization of the candidate set C.
   - First we divide D_A into the disjoint intervals $I_1, ..., I_n$ according to local extrema of f^U.
   - By the random search of internal nodes of the B$^+$-tree, we find an object x with the highest value of $f^U(x)$ on each interval and add it into C.
2. Obtaining the next best object.
   - An object x with the highest current value of $f^U(x)$ in C is chosen and replaced by the next one object y from the corresponding interval I_i.
   - (a) If it is possible, this new object y is obtained as the next object in the same object array.
     (b) Otherwise, we move to the neighbouring object array by the sequential search of the leaf nodes of the B$^+$-tree and we obtain the first object of this object array as object y. We move between two object arrays in the corresponding direction according to the course of f^U on interval I_i.
   - Finally, the pair $\{x, f^U(x)\}$ is returned.

Note that in the second step we do not need to use the random search of internal nodes of the B$^+$-tree. Thus, we use the random search only in the first step. The random search is executed only n times, i.e., for each interval there is one random access. In the second step, we use only the sequential search of leaf nodes of the B$^+$-tree. Moreover, the execution of the second step only causes obtaining one new object by the sequential search.

The pseudo-code depicted in Algorithm 4 describes the BtreeList algorithm and its two procedures, which implement the two steps. For the sake of clarity, we assume that the pointer of each key only refers to one object. This restriction of he pseudo-code can be easily extended to the use of the object arrays. With the BtreeList algorithm, it is possible to implement the sequential access to the objects with the usage of user's local preference $f^U(x)$ for an attribute. Procedure createCandidates performs the initialization of the candidate set (the first step). Procedure getNextPair performs the obtaining of the next best object according to $f^U(x)$ (the second step).

Example 6.1.3 Figure 6.2 depicts fifteen objects indexed in the B$^+$-tree according to an attribute and a non-monotone user's fuzzy function f^U. By the execution of the procedure createCandidates, the domain of the attribute is divided into five intervals $I_1, ..., I_5$, in which the fuzzy function is

Algorithm 4 BTREELIST algorithm (list with support of local preference)

BTREELIST(B$^+$-tree T, fuzzy function f^U)

1: $C :=$ CREATECANDIDATES(T, f^U);
2: **while** $C \neq \emptyset$ **do**
3: $\{x, f^U(x)\} :=$ GETNEXTPAIR(T, f^U, C);
4: **end while**
5: **return** ;

procedure CREATECANDIDATES(B$^+$-tree T, fuzzy function f^U)

1: Create n disjoint intervals I_1, ..., I_n in T according to the course of f^U;
2: $C := \emptyset$;
3: **for** $1 \leq i \leq n$ **do**
4: **if** f^U is nondecreasing on I_i **then**
5: If possible, find the first object $x \in I_i$ in B$^+$-tree T by the random search, which has $A(x)$ less or equal to upper bound of interval I_i ;
6: **end if**
7: **if** f^U is nonincreasing on I_i **then**
8: If possible, find the first object $x \in I_i$ in B$^+$-tree T by the random search, which has $A(x)$ greater or equal to lower bound of interval I_i;
9: **end if**
10: **if** $x \in I_i$ **then**
11: $C := C \cup \{x\}$;
12: **end if**
13: **end for**
14: For each $x \in C$ remember a pointer to the leaf level of B$^+$-tree T;
15: **return** C;

procedure GETNEXTPAIR(B$^+$-tree T, fuzzy function f^U, candidates C)

1: Choose object $x \in C$ with the highest value $f^U(x)$;
2: $I_i :=$ an interval, for which $x \in I_i$ holds;
3: $C := C - \{x\}$;
4: **if** f^U is nondecreasing on I_i **then**
5: If possible, move from x to its left neighbouring object $y \in I_i$ in B$^+$-tree T by the sequential search;
6: **end if**
7: **if** f^U is nonincreasing on I_i **then**
8: If possible, move from x to its right neighbouring object $y \in I_i$ in B$^+$-tree T by the sequential search;
9: **end if**
10: **if** $y \in I_i$ **then**
11: $C := C \cup \{y\}$;
12: **end if**
13: **return** $\{x, f^U(x)\}$;

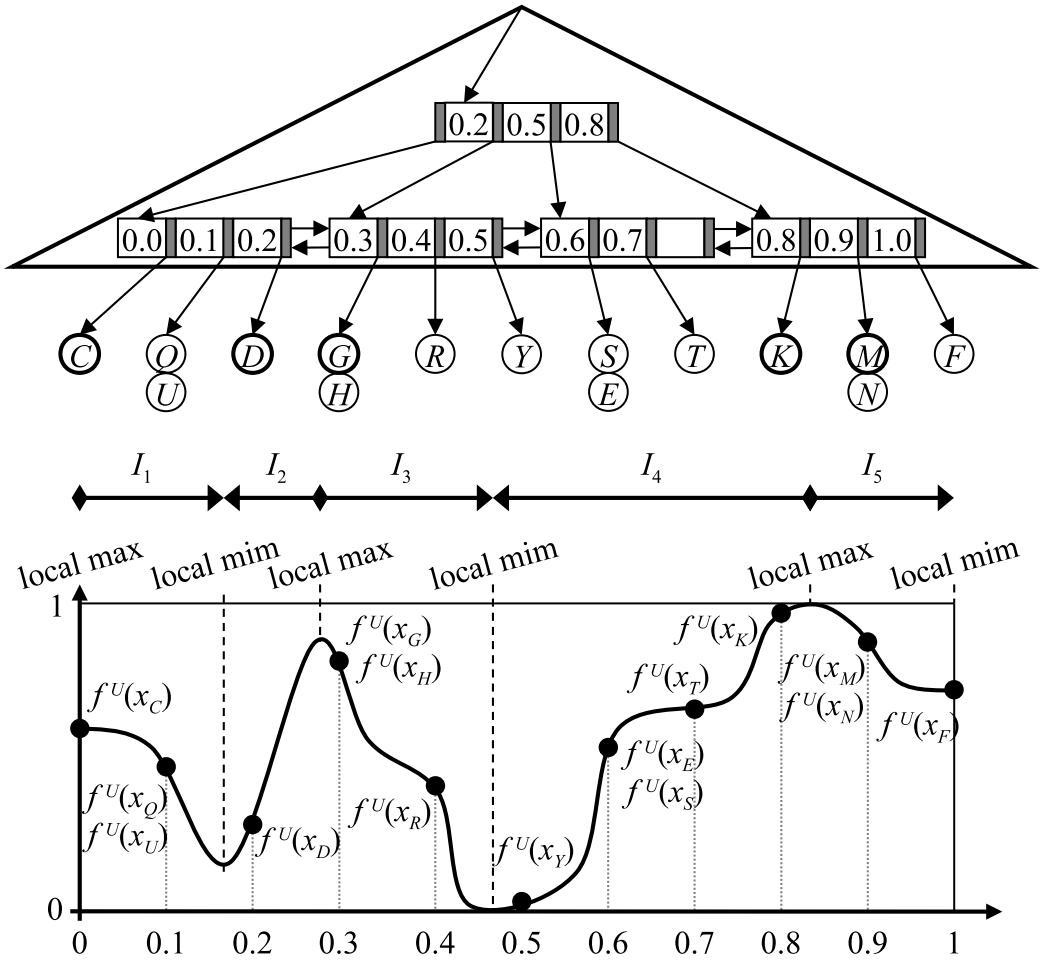


Figure 6.2: The sequential obtaining of the objects from a B$^+$-tree according to a non-monotone fuzzy function, which is monotone in five continuous disjoint intervals I_1, ..., I_5.

monotone, according to local extrema. The created candidate set C includes the objects x_C, x_D, x_G, x_K, x_M. These objects are highlighted in the picture of the B$^+$-tree. Because the object x_K has the biggest value $f^U(x_K)$ in C, the pair $\{x_K, f^U(x_K)\}$ is obtained by the first execution of procedure GETNEXTPAIR. Object x_K is obtained from interval I_4. Therefore, the next object has to be obtained in interval I_4. Since f^U is nondecreasing in interval I_4, the left neighbouring object x_T of x_K is obtained and added into C. The pair $\{x_M, f^U(x_M)\}$ is obtained by the second execution of procedure GETNEXTPAIR analogously. Finally, we obtain all objects from the B$^+$-tree in descending order according to the fuzzy function f^U, namely, x_K, x_M, x_N, x_G, x_H, x_F, x_T, x_C, x_S, x_E, x_Q, x_U, x_R, x_D, x_Y, respectively.

6.2 Usage of B$^+$-trees in Fagin's algorithms

Fagin's algorithms as the main representatives of the top-k search algorithms, offer the possibility to find the best k object only according to a monotone aggregate function g as a global preference of user U. Thus, they do not support local user preferences directly. To find the top-k objects with a support of local preferences, we would have to create new sorted lists according to the local preferences. That would cause an unnecessary amount of data preprocessing before each computation of Fagin's algorithms.

In this chapter, we have established a new model of the sorted list with the support of the user's local preference, which enables the access to the objects as pairs $\{x, f^U(x)\}$ sequentially in descending order according to values of $f^U(x)$. This new model can be used by Fagin's algorithms in such a way that instead of the sorted lists $L_1, ..., L_m$ we use B$^+$-trees $T_1, ..., T_m$. In each T_i all the objects are indexed by values of attribute A_i. We create m of B$^+$-trees only before the first run of Fagin's algorithms. The B$^+$-trees are independent of any user preferences and common for all users.

A particular user U only specifies his/her local preferences as fuzzy functions $f_1^U, ..., f_m^U$ and global preferences as an aggregate function g^U. Then the algorithm obtains pairs $\{x, f_i^U(x)\}$ sequentially from B$^+$-trees from each T_i according to fuzzy function f_i^U and finds the top-k objects according to the preferences of user U. For each new user's preferences or for the preferences of the next user, the algorithm is run again and finds other top-k objects, and we do not need to do any preprocessing of all data before the next computation of the algorithm.

Additionally to the original Fagin's algorithms, when we use B$^+$-trees instead of sorted lists, it is necessary to initialize the candidate set according to fuzzy function f_i^U for each B$^+$-tree T_i before the computation of Fagin's algorithm. The CREATECANDIDATES procedure has to be executed for each B$^+$-tree T_i. If we suppose that each fuzzy function f_i^U has n_i+1 local extrema, then, according to Notice 6.1.3, n_i candidates have to be initialized, each by performing the random search in each B$^+$-tree T_i. The total number of candidates is $\sum_{i=1}^{m} n_i$. But in general, a course of real users fuzzy functions is not usually too complicated. Thus, we suppose that the total number of candidates is not too big. For example, Figure 6.3 depicts four typical fuzzy functions of real user and four B$^+$-trees, in which some objects are indexed according to four attributes. The leaf level of each B$^+$-tree is divided into one or two disjoint intervals according to two or three local extrema of the corresponding fuzzy function.

After initialization of C, the algorithm obtains the objects only by the sequential search in the leaf level of B$^+$-trees, i.e., the procedure GETNEXTPAIR

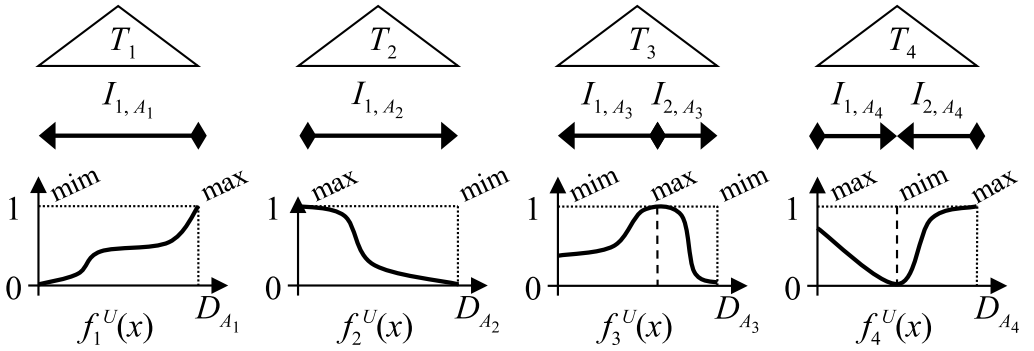


Figure 6.3: Local preferences of user U as fuzzy functions $f_1^U, f_2^U, f_3^U, f_4^U$ for four attributes A_1, A_2, A_3, A_4, whose domains are divided into intervals I_{1,A_1}, I_{1,A_2}, I_{1,A_3} and I_{2,A_3}, I_{1,A_4} and I_{2,A_4}, respectlively.

is executed repeatedly for one of the B$^+$-trees (in round-robin style, see Section 4.1.2). Despite the fact that the number of candidates is $\sum_{i=1}^{m} n_i$, by obtaining a new pair $\{x, f_i^U(x)\}$ from T_i it is necessary to replace only the best candidate object x by the next one from the same interval. Additionally, if we perceive the leaf level of the B$^+$-tree as a kind of sorted list, then the obtaining of the next pair $\{x, f_i^U(x)\}$ from each T_i is equivalent to the sorted access because it is a movement to a neighbouring data item.

Note that we choose the same name for the procedure getNextPair in Algorithm 4 as the name of the getNextPair procedure in Fagin's algorithms TA and NRA (see Algorithm 1 and Algorithm 2). Original algorithms TA and NRA obtain the pairs $\{x, A_i(x)\}$ sequentially by the execution of the command getNextPair(L_i), which performs the sorted access in the sorted list L_i. For the support of local preferences, the pairs $\{x, f_i^U(x)\}$ have to be obtained sequentially by the execution of the getNextPair$(T_i,\ f_i^U,\ C_i)$ procedure in B$^+$-tree T_i. Additionally to the original Fagin's algorithms, for each B$^+$-tree T_i the candidate set C_i has to be created by the createCandidates$(T_i,\ f_i^U)$ before the first execution of the getNextPair$(T_i,\ f_i^U,\ C_i)$ procedure.

The pseudo-code depicted in Algorithm 5 describes the NRALP algorithm as an extension of the NRA algorithm, which uses the local preferences of user U. The NRALP algorithm enables user U to find the top-k objects according to his/her global preference and local preferences assigned as aggregate function g and fuzzy functions $f_1^U, ..., f_m^U$. The objects are indexed in m B$^+$-trees $T_1, ..., T_m$ and each B$^+$-tree contains all the objects indexed according to their values of one corresponding attribute. The main part of the NRALP algorithm is identical to the NRA algorithm depicted in Algorithm 2. The parts of the NRA algorithm which have a comment at beginning, are referred

Algorithm 5 NRALP algorithm (NRA algorithm with local preferences)

NRALP(B^+-trees $T_1, ..., T_m$, aggergation g, local preferences $f_1^U, ..., f_m^U$, number k)

```
 1: top-k := ∅;
 2: candidates := ∅;
 3: for 1 ≤ i ≤ m do high_i := 1;
 4: min-k := 0;
 5: for 1 ≤ i ≤ m do C_i := createCandidates(T_i, f_i^U);
 6: repeat
 7:    for 1 ≤ i ≤ m do
 8:       // Obtaining the next pair from one of the lists
 9:       {x, f_i^U(x)} := getNextPair(T_i, f_i^U, C_i); // sorted access
10:       high_i := f_i^U(x);
11:       for 1 ≤ j ≤ m do
12:          if A_j(x) is known then
13:             worst_j := f_j^U(x); best_j := f_j^U(x);
14:          end if
15:          if A_j(x) is unknown then
16:             worst_j := 0; best_j := high_j;
17:          end if
18:       end for
19:       Worst(x) := g(worst_1, ..., worst_m);
20:       Best(x) := g(best_1, ..., best_m);
21:       // NRA: Processing of the object from new pair
22:    end for
23:    // NRA: Removing of non-viable candidates
24: until candidates = ∅ and top-k.size() = k
25: return top-k;
```

to in the NRALP algorithm by "NRA:", namely, they are "Processing of the object from new pair" from line 16 to line 30 and "Removing of non-viable candidates" from line 32 to line 38.

Additionally to the NRA algorithm, the TA algorithm and the CA algorithm (Section 4.4) also use the random access to lists $L_1, ..., L_m$. By the random access an attribute value $A_i(x)$ of the object x is obtained from list L_i. When we apply the local preferences, this can be realized similarly. In addition, the attribute value $A_i(x)$ is evaluated by f_i^U immediately, i.e., $f_i^U(x)$ is obtained.

The execution of the random access in the B^+-tree can be difficult because the B^+-tree does not provide the possibility to obtain the key (an attribute value) according to an object (or object identifier). For the realization of

Algorithm 6 TALP algorithm (TA algorithm with local preferences)

TALP(B^+-trees $T_1, ..., T_m$, aggergation g, local preferences $f_1^U, ..., f_m^U$, number k)

```
 1: top-k := ∅;
 2: for 1 ≤ i ≤ m do high_i := 1;
 3: threshold := 1;
 4: min-k := 0;
 5: for 1 ≤ i ≤ m do C_i := createCandidates(T_i, f_i^U);
 6: repeat
 7:    for 1 ≤ i ≤ m do
 8:       {x, f_i^U(x)} := getNextPair(T_i, f_i^U, C_i); // sorted access
 9:       high_i := f_i^U(x);
10:       threshold := g(high_1, ..., high_m);
11:       for 1 ≤ j ≤ m do
12:          if j ≠ i then
13:             A_j(x) := getValue(T_j, x); // random access
14:             f_j^U(x) := f_j^U(A_j(x));
15:          end if
16:       end for
17:       score(x) := g(f_1^U(x), ..., f_m^U(x));
18:       if score(x) > min-k then
19:          if top-k.size() = k then top-k.removeKthObject();
20:          top-k.insertInTheRightPlace(x);
21:          min-k := top-k.getMin-k();
22:       end if
23:    end for
24: until threshold ≤ min-k
25: return top-k;
```

the random access, we can use an additional data structure, for example, an associative array, in which the attribute values are indexed according to the objects.

We note that Fagin's NRA algorithm and also the NRALP algorithm use only the sorted access. They do not use the random access at all. When the B^+-trees are used instead of the sorted lists, the NRA algorithm does not need to use any additional data structure.

The pseudo-code depicted in Algorithm 6 describes the TALP algorithm as an extension of the TA algorithm, which enables user U to find the top-k objects according to his/her global preference g and local preferences $f_1^U, ..., f_m^U$. For the sake of clarity, we assume that in the TALP algorithm B^+-trees $T_1, ..., T_m$ provide the random access.

6.3 Lists based on B$^+$-tree on remote servers

In this section, we focus on searching the top-k objects with more attributes according to user preferences in the Web environment. As a motivation, we can use the following example of choosing the best holiday destination. When a user is choosing a holiday, he/she must consider many variables, which may change frequently and can be located on various remote servers on the Internet, such as airline tickets price, weather forecast, accommodation prices or the distance from places of interest. When the user wishes to do a more complex search, he/she is forced to access every location separately and put the obtained data together by himself/herself.

More formally, we assume that attributes of an object type are distributed on remote servers in a disjunctive way. In general, we suppose that all the values of single attribute are stored in only one remote server on the Internet. We developed TOPKNET system [HOP11], which is capable of accessing all these servers from a single location and finding the top-k objects with the support of local user preferences based on fuzzy functions efficiently.

To the best of our knowledge, one of the best solutions of the efficient processing of the top-k search in wide-area distributed data repositories is KLEE framework [MTW05]. The framework tries to minimize a computational cost, which includes network latency and local peer work. KLEE framework uses approximate algorithms and searches for the top-k documents but only according to an aggregation function.

The TOPKNET system is based on client-server architecture and communication via Web Services. We deal with the usage of Fagin's NRA algorithm, which can find the top-k objects without the need to access all the objects (see Section 4.3). In each remote server, the objects are indexed according to a particular attribute in the B$^+$-tree, which provide the support of the user's local preferences (see Section 6.1). The TOPKNET system is capable to minimize the amount of network communication by using the cache memory, which loads the data from remote servers in batches. Thus it reduces the amount of network communication and solves partially the network latency problem. Moreover, we focus on the crucial requirement that each server is stateless, i.e., independent of any previous requests.

6.3.1 Client-server architecture

Since we deal with top-k search in network environment and data is stored on remote servers, it was necessary to design suitable architecture for needs of network communication. Thus, we have chosen the client-server architecture. Specifically, for network communication requirements the Web Services tech-

nology, we used RPC (remote procedure call) type of SOAP (simple object access protocol) communicating via HTTP. The server operates as a service responding to client application requests.

For searching the top-k object according to user preferences, we designed a client application, which includes a search engine based on Fagin's NRA algorithm. The client application is able to obtain objects with their attribute values from remote servers and thus to find efficiently the top-k objects for a user.

For a large amount of objects stored on a server with values of attribute A, it is not efficient to load all the objects for searching the top-k objects, since the NRA algorithm does not need to access all the objects. Therefore, the client application loads objects from each server in batches. The client application sends requests repeatedly to individual servers, which respond to the requests by sending batches of objects. Figure 6.4 shows a simplified separation of the system into server and client component.

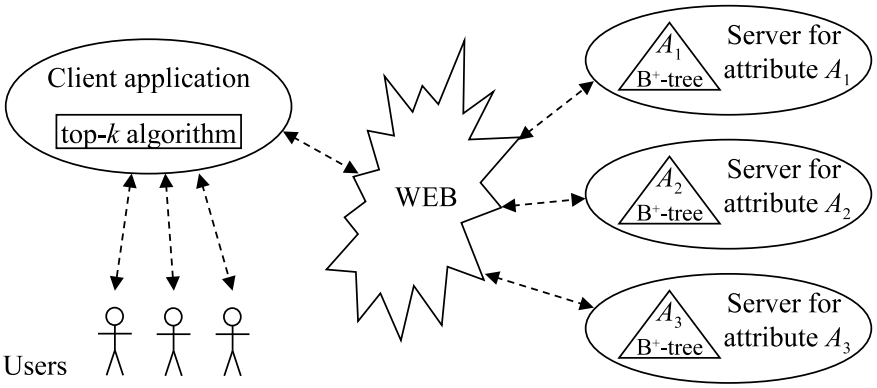


Figure 6.4: Architecture of the TOPKNET system.

6.3.2 Stateless server

The server part of the TOPKNET system handles the storage of data and the fastest access to the data possible. All the values of a single attribute A of all the objects are stored on the server in the B$^+$-tree. The core of the server part is an algorithm, which manages data and communicates with the client application via Web Service interface.

The algorithm is designed in such a way that the server has to be stateless and thus does not store any additional data for any connection. The server only provides a response for an immediate request and is independent of any previous requests from the client application. To minimize the amount network communication, the server provides the objects in batches.

When we want to use local preferences in the NRA algorithm the server has to provide pairs, which are in the form $\{x, f^U(x)\}$. These pairs must be provided in batches in descending order according to f^U, which is why objects are indexed according to attribute values in the B$^+$-tree.

For the requirements of network communication, we need to represent fuzzy functions in a suitable manner. Handling the analytical representation of a function is unpractical, especially from the implementation point of view. Every fuzzy function can be approximated by a sequence of linear functions (see Section 3.3.3), which can be represented as a sequence of points.

In general, a course of fuzzy functions of real users is usually not too complicated. For example, Figure 6.3 depicts the four most common types of fuzzy functions of real users. Therefore, such sequences of points contain only a few points. And that is why we use this representation of fuzzy functions in network communication.

Providing objects in batches

Since the server should be stateless and be able to provide objects in batches in descending order according to an arbitrary user's fuzzy function f^U, each client application request has to contain information about this function.

In the TOPKNET system we use, for representation of a fuzzy function, a sequence of points. In the case that a client application requests the first batch of S objects, we proceed as it is shown in Section 6.1.3. We divide the definition domain of arbitrary f^U according to its course into n intervals $I_1, ..., I_n$ that f^U is monotone on each of the intervals.

The pseudo-code depicted in Algorithm 7 describes the BtreeListServer algorithm. We adopt from Algorithm 4 the createCandidates procedure and the getNextPair procedure. The set of candidates C is created in the corresponding intervals on the leaf level of the B$^+$-tree by the createCandidates procedure. Then the getNextPair procedure is executed S times, i.e., S pairs $\{x, f^U(x)\}$ are obtained. In this way the server will provide the list of pairs *Batch* containing S objects sorted in descending order according to $f^U(x)$.

When the algorithm returns the *Batch* list of S pairs for the first time, it is sent, as the first batch, to the client application. In this moment C contains a pointer to the leaf level of the B$^+$-tree. Further obtaining objects from the B$^+$-tree must continue from the identical state of C.

We require statelessness of the server, and it is necessary to sent the batch of the next S objects according to f^U in the following request of client application. Therefore, the response for client application must contain not only the batch of pairs, but also the set C, which determines where the

Algorithm 7 BTREELISTSERVER algorithm (B^+-tree as list on server)

BTREELISTSERVER(B^+-tree T, fuzzy function f^U, batch size S, set of pairs P)

```
 1: if P = ∅ then
 2:    C := createCandidates(T, f^U);
 3: else
 4:    C := activateCandidates(T, f^U, P);
 5: end if
 6: Batch := ∅;
 7: for 1 ≤ i ≤ S do
 8:    {x, f^U(x)} := getNextPair(T, f^U);
 9:    Batch := Batch ∪ {x, f^U(x)};
10: end for
11: P := elements of C in form {x, A(x)};
12: return Batch and P;
```

procedure ACTIVATECANDIDATES(B^+-tree T, fuzzy function f^U, set of points P)

```
 1: Create n intervals I_1, ..., I_n according to the course of f^U;
 2: C := ∅;
 3: for all {x, A(x)} ∈ P do
 4:    // the B+-tree T is traversed top to bottom only once (starting at the root)
 5:    Find x according to A(x) in B+-tree T;
 6:    For x remember a pointer to the leaf level of B+-tree T to which it belongs;
 7:    For x remember the interval I_i to which it belongs;
 8:    C := C ∪ {x};
 9: end for
10: return C;
```

GETNEXTPAIR procedure last stopped when obtaining objects from the leaf level of the B^+-tree.

As it is defined in Section 6.1.3, for an object $x \in C$ we have to remember a pointer to the leaf level of the B^+-tree, its attribute value $A(x)$, and the interval I_i to which the value belongs. Since transferring of pointers through network communication is overcomplicated, we represent elements of candidate set C as a *set of pairs* P, elements of which are pairs $\{x, A(x)\}$ consisting of object x (or object identifier) and its attribute value $A(x)$ (see line 11 in Algorithm 7).

Consequently, the server sends as the response to the client application the batch of objects evaluated by fuzzy function f^U and set of pairs P (see line 12 in Algorithm 7). If the client application sends a request for the next batch of objects, it must contain the size of batch S, the sequence of points representing the f^U and P.

In this way we ensure statelessness of the server with a minimal increas in the size of transferred data in both request and response. The first request from the client application differs from the following ones, because its P is empty. In this case, C must be created in leaf level of the B$^+$-tree via the CREATECANDIDATES procedure. The following requests contain P, from which C is reconstructed (see line 4) in order to be able to continue obtaining objects in the leaf level of the B$^+$-tree, where the algorithm stopped after obtaining objects in the previous batch.

The pseudo-code depicted in Algorithm 7 describes also the ACTIVATECANDIDATES procedure for the reconstruction of a set of candidates C in the B$^+$-tree from a set of pairs P.

6.3.3 Client application

The client application of TOPKNET system is able to obtain objects with their attribute values from remote servers and efficiently find the top-k objects for a user. It uses the Web services interface for communication with remote servers. In our solution, we try to minimize the amount of network communication by using the cache memory, which loads the data from remote servers in batches, and solves the network latency problems. The client application uses one cache memory for each single remote server.

Cache memory

The TOPKNET system works on the client/server architecture principle. For searching top-k objects it is necessary to be as fast as possible. Therefore, it is not acceptable for the NRA algorithm, which uses the sequential access, to wait for every single object from every list to be loaded from the server. Thus the algorithm does not access the server directly but through a cache memory. The NRA algorithm then obtains objects from the cache sequentially in descending order according to f^U. The objects are obtained as pairs $\{x, f^U(x)\}$.

The main part of the cache memory consists of two parallel threads using one shared list, which contains pairs $\{x, f^U(x)\}$ loaded from the server. These pairs are sorted in descending order according to f^U. We denote the size of this list as h. The first thread, which we call *consumer*, only provides the pairs for the NRA algorithm. The second thread, which we call *producer*, ensures the loading of pairs in batches from remote servers. The consumer waits only for the NRA algorithm to request another pair $\{x, f^U(x)\}$ and loads the requested value from the shared list and returns it to the algorithm for further processing.

The main duty of the producer is to manage providing a sufficient amount of pairs in the shared list in order to prevent the NRA algorithm from waiting for it. As soon as the consumer realizes that the amount of objects in shared list decreases under previously established level l (e.g. $l = h \ / \ 2$), it wakes up the producer, which starts obtaining further pairs from the server in batches again and increases their amount in the shared list.

However, if the network communication with the server is so slow that the producer is not able to replenish the shared list, the consumer is put to sleep and waits for the producer to inform it that further pairs are now available. If this case occurs too frequently, it is suitable to increase the size of shared list h or to set a higher value of level l.

Various users of client application

Each one of the remote servers contains all the objects with values of single attribute A_i stored in the B^+-tree and is able to provide the pairs $\{x, f_i^U(x)\}$ in descending order according to user fuzzy function f_i^U. A particular user U of client application can specify his/her local preferences with fuzzy functions $f_1^U, ..., f_m^U$ and a global preference with an aggregation function g^U.

Then the client application gets pairs $\{x, f_i^U(x)\}$ sequentially in batches from the remote servers can efficiently find the top-k objects for user U. For each new user's preferences or for preferences of the next user, the NRA algorithm is run again and finds other top-k objects according to a user's ranking function, which is composed of his/her global preference and his/her local preferences.

The remote servers are stateless and thus independent of any previous requests. It means that each remote server only provides a response to an immediate request and can be used by different client applications concurrently. Moreover, these client applications can be situated in different places on the Internet and used by different users in the same moment.

6.4 Experiments

We developed the TOPKNET system based on client-server architecture. The implementation was developed in Java. We used Web Service framework, namely, Apache Axis2, for a communication between the client application and remote servers. The network messages are sent in XML format. The remote servers of the TOPKNET system have been running on the application server, namely, Apache Tomcat. These technologies are optimized to work together and ensure the stability of the TOPKNET system.

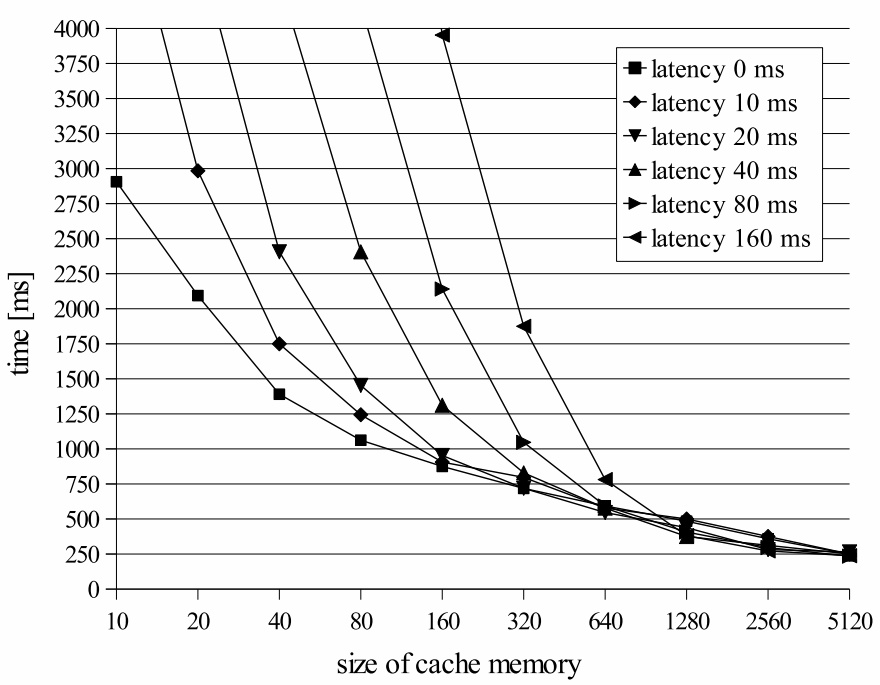


Figure 6.5: Computation time of the TOPKNET system for different values of network latency and cache memory size.

We focused on experiments in network environment. We used three sets of 100 000 objects with 5 attributes for the experiments. It means that the TOPKNET system was using the client application and five remote servers. We focused on the execution time, which is needed for finding the top-k objects in the client application according to user preferences. We used such local user preferences that some attribute values were the most preferred. The arithmetic average was used as the aggregate function.

We found out that the computation time of the NRA algorithm is most dependent on network communication between client application and remote servers. We used $k = 20$, i.e., the TOPKNET system searched for the best 20 objects because the difference in the computation time in our particular experiment was relatively small for any k smaller than 250.

We tested the TOPKNET system using various values of network latency, which was simulated as a delay of server response. We focused on the dependence of computation time of the NRA algorithm according to the size of cache memory. Naturally, the best results were achieved by zero network latency and the biggest size of cache memory. Figure 6.5 shows the results of this comparison.

6.5 Summary

We provided a new model of a sorted list, which enables Fagin's algorithms to search for the top-k objects with regard to user local preferences. This model of a sorted list based on the B$^+$-tree provides many possibilities of applying it. In the next parts of this thesis, we focus on the utilization of the model for the purposes of the top-k search in tree-based multidimensional data structures.

Moreover, we developed and implemented the TOPKNET system, which can efficiently find the top-k objects for various users with data stored on remote servers. Our solution can be extended and improved in many aspects. For example, our solution for server part of TOPKNET system assumes that fuzzy function and data on the server are not changing during the NRA algorithm computation. It would be interesting to solve the problem in a more dynamic environment, e.g., data stored in B$^+$-tree on the server can change very fast.

For example, in [GV08] the authors describe some heuristics for the NRA algorithm, which can automatically change the progress of the sorted access in the lists. Similarly, a motivation for future research could be to develop some heuristics based on different network latency for each of the servers.

And also, it is possible to monitor communication latency of servers during the run of the NRA algorithm. It would be more suitable to adjust the properties of cache memory, i.e., h and l (see Section 6.3.3), so that it could change its settings automatically. Moreover, this solution permits the size of batch to be different in each request of client application. This property can be utilized for further, more advanced optimization of the network communication.

Chapter 7

Top-k Search Based on MDB-tree

In this chapter, we focus on the use of the multidimensional B-tree, denoted as MDB-tree (see Section 5.3), for the top-k search. At the beginning of our research, there was an idea to apply Fagin's algorithms on the MDB-tree. It is not possible to apply them for this purpose directly because there are many obstacles which cannot be overcome. Therefore, we started developing new algorithms, which would utilize useful features of Fagin's algorithms. We were able to develop the MD algorithm [OP08], which can search for the top-k objects in the MDB-tree according to a monotone aggregate function (see Section 3.4.2). The MD algorithm is an efficient top-k algorithm it searches only in some parts of the MDB-tree. Thus, the MD algorithm does not need to obtain the greater part of the objects stored in the MDB-tree.

Moreover, we focus on the top-k search in the MDB-tree according to the model of user preferences based on the local preferences, which enable us to use a non-monotone ranking function (see Section 3.4.3). Therefore, we propose to use the variant of the MDB-tree, which is composed of B^+-trees (see Section 5.4). Thus, we can apply the model of sorted list based on B^+-tree (see Section 6.1). Then it is possible to apply the local preferences directly during the computation of the top-k search in the MDB-tree. We developed the MDLP algorithm as an extension of the MD algorithm, which is able to search for the top-k objects also by using the local preferences.

First, in Section 7.1 we focus on the abilities the MDB-tree, which help us to develop the MD algorithm. In Section 7.2, we focus on the top-k search in the MDB-tree. We introduce the depth-first search of the MDB-tree, by which we can obtain all the objects stored in the MDB-tree. Then, we focus on some useful advantages of Fagin's algorithms, which can be adopted and applied for the top-k search in the MDB-tree. We present a sequence of some statements with consequences and proofs. In Section 7.2.4, we focus on utilization of the monotonicity of the aggregate function and we provide

the most important statement, by which it possible to skip some parts of the MDB-tree during the execution of the top-k search in the MDB-tree.

In Section 7.3, we present the MD algorithm and the MDLP algorithm using the local preferences. In Section 7.4 we show experimental results of presented algorithms, which achieve better results in the number of accessed objects than Fagin's algorithms. In Section 7.5, we present an advantageous index structure derived from the MDB-tree. Finally, in Section 7.6, we provide a summary and an analysis of the efficiency of the MD algorithm.

7.1 Abilities of MDB-tree

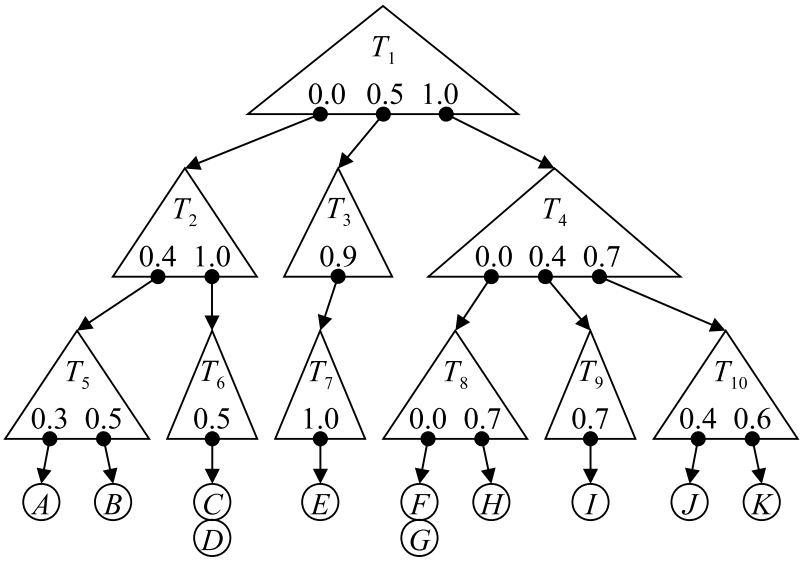
Usage of the MDB-tree allows us to index a set of objects X with more attributes $A_1, ..., A_m$ in one data structure. The number of levels in the MDB-tree is equal to the number of indexed attributes and all values of one attribute A_i are stored only in one corresponding level. Thus, each ith level of MDB-tree is composed from B$^+$-trees containing key values from the actual domain $D_{A_i}^{act}$ of corresponding attribute A_i. Figure 7.1 depicts a set of objects with three attributes indexed in the MDB-tree composed of the B$^+$-trees. The MDB-tree has three levels. The same set of objects is depicted in the table on the right side of the figure.

In this section, for the sake of clarity, we assume that the values of attributes $A_1, ..., A_m$ are real numbers from the interval $[0, 1]$, i.e., each $D_{A_i}^{act} \subseteq [0, 1]$. This restriction is not crucial for the purposes of the top-k search in the MDB-tree. Any other sortable attribute types (see Section 3.3.2) can be applied instead of the unit interval.

Each particular B$^+$-tree of the MDB-tree is determined by its unique position in the MDB-tree. This position depends on the path, by which we can reach the B$^+$-tree from the root of the MDB-tree. To explain the MD algorithm, we use a *tree identifier* to explicitly identify each B$^+$-tree in the MDB-tree. More formally, we define the tree identifier as follows.

Notation 7.1a (tree identifier) *Let a B$^+$-tree be situated on the ith level of the MDB-tree. Let a sequence of keys $k_1, ..., k_{i-1}$, where $k_i \in D_{A_i}^{act}$, determine the path from the root of the MDB-tree to the B$^+$-tree through other B$^+$-trees on levels $1, ..., i-1$ using their keys $k_1, ..., k_{i-1}$, respectively. Then we denote tree identifier of the B$^+$-tree as $(k_1, ..., k_{i-1})$. We denote tree identifier of the B$^+$-tree in the first level of the MDB-tree as $(\emptyset)$.*

For example, in Figure 7.1, $(k_1, k_2) = (1.0, 0.0)$ is the tree identifier of B$^+$-tree T_8 on the third level, which contains keys 0.0, 0.7 by which objects x_F, x_G, x_H are referred to, or B$^+$-tree T_3 has tree identifier (0.5).



	A_1	A_2	A_3
A	0.0	0.4	0.3
B	0.0	0.4	0.5
C	0.0	1.0	0.5
D	0.0	1.0	0.5
E	0.5	0.9	1.0
F	1.0	0.0	0.0
G	1.0	0.0	0.7
H	1.0	0.0	0.7
I	1.0	0.4	0.7
J	1.0	0.7	0.4
K	1.0	0.7	0.6

Figure 7.1: Eleven objects with their values of three attributes stored in the Multidimensional B-tree, on the left, and in a table, on the right.

Each B$^+$-tree on the ith level of the MDB-tree includes some values of attribute A_i as its keys in its leaf nodes. For each key, there is a corresponding pointer, which refers to another node of the MDB-tree.

Notation 7.1b (pointer of key) *Let a B$^+$-tree be situated on the ith level of the MDB-tree. Let a key $k_i \in D^{act}_{A_i}$ be situated on the leaf level of the B$^+$-tree. We denote the corresponding pointer of key k_i as $\rho(k_i)$.*

According to the property 3 of Definition 5.4d, the following holds. If k_i is the key of a B$^+$-tree on the ith level of the MDB-tree and $i < m$, then $\rho(k_i)$ refers to a B$^+$-tree on the $(i+1)$th level of MDB-tree. Otherwise, if this B$^+$-tree is on the mth level of the MDB-tree, $\rho(k_i)$ refers to the associated data. For example, in Figure 7.1, the pointer $\rho(0.4)$ of the key 0.4 in the B$^+$-tree T_2 on the second level of the MDB-tree refers to the B$^+$-tree T_5 on the third level of the MDB-tree.

In general, if we suppose a set of objects X with arbitrary values of the attributes, some objects with the same values of all the attributes can exist in X, i.e., objects with the same multidimensional key $(k_1, ..., k_m)$, where $k_i \in D^{act}_{A_i}$. Therefore, we introduce the *object array*, in which the objects with the same values of all m attributes are stored. All the objects indexed in the MDB-tree are stored in the object arrays. We perceive the object arrays as the associated data. The pointers which point to the associated data are situated only in the B$^+$-trees on the last level of the MDB-tree. Each object array includes objects with the same value of a multidimensional key.

For example, in Figure 7.1, the object arrays are depicted as a sequence of circles. There are two object arrays, which include more objects, namely, one object array including objects x_C and x_D, and another object array including objects x_F and x_G.

7.2 Top-k search in MDB-tree

In this section, first we focus on the top-k search in the MDB-tree according to a monotone aggregate function g (see Section 4.1.3). Analogously as in the case of Fagin's algorithms, we understand the aggregate function as a combination function, which combines (or aggregates) attribute values of an object $x \in X$ into its overall score, which we denote as $score(x)$. If an aggregate function g with m variables is given, then it is possible to evaluate the $score(x)$ of each object x by expression $score(x) = g(A_1(x), ..., A_m(x))$, where $A_1(x), ..., A_m(x)$ are values of m attributes of object x.

The easiest way of finding the top-k objects from X, which are indexed in the MDB-tree, is to use the exhaustive search of the MDB-tree. First, we obtain all objects $x \in X$ with all their attribute values from the MDB-tree. Then it is sufficient to evaluate ranking (overall score) for all objects $x \in X$ according to the aggregate function and to choose the top-k objects with the highest values of ranking.

7.2.1 Depth-first search

In each multiway balanced search tree, it is possible to obtain all objects using a recursive procedure starting at the root of the tree. For this purpose, we use a recursive procedure based on depth-first search. This procedure recursively searches the MDB-tree starting at the B$^+$-tree on the first level of the MDB-tree. The recursive procedure is always executed on one particular B$^+$-tree. If this B$^+$-tree through the pointers of its keys refers to B$^+$-trees on the next level, the procedure is run in each of these B$^+$-trees as well. In case of the mth (the last) level of the MDB-tree, i.e., if a B$^+$-tree through pointers of its keys refers to object arrays, the objects are obtained from them.

The pseudo-code depicted in Algorithm 8 describes the depth-first search of the MDB-tree by recursive procedure SEARCHMDBTREE, which enables us to obtain all the objects from the MDB-tree. The input of the algorithm is a MDB-tree and output is a list, denoted as *objects*, which contains all the objects stored in this MDB-tree.

To clearly determine B$^+$-tree, in which the procedure is being executed, we use the tree identifier introduced by Notation 7.1a. The SEARCHMDB-

Algorithm 8 searchMDBtree (depth-first search of the MDB-tree)

searchMDBtree(MDB-tree *MDBT*)

```
1: objects := ∅;
2: searchMDBtree(MDBT, (∅));
3: return objects;
```

procedure searchMDBtree(MDB-tree *MDBT*, tree identifier $(k_1, ..., k_{i-1})$)

```
1: T := B+-tree with identifier (k_1, ..., k_{i-1}) on the ith level of MDBT;
2: Find the position of key k_i with the highest value of A_i in the leaf nodes of T
   by the random search;
3: while ∃ the next key k_i ∈ T do
4:    Choose the next key k_i with the highest value of A_i in the leaf nodes of T
      by the sequential search;
5:    // B+-tree T is on the ith level of MDBT, where i < m.
6:    if ρ(k_i) refers to B+-tree then
7:       searchMDBtree(MDBT, (k_1, ..., k_{i-1}, k_i));
8:    end if
9:    // B+-tree T is on the mth level of MDBT.
10:   if ρ(k_i) refers to object array then
11:      Append all the objects from this object array to objects also with
         their values k_1, ..., k_{i-1}, k_i of attributes A_1, ..., A_m, respectively;
12:   end if
13: end while
14: return ;
```

tree procedure starts at the B$^+$-tree on the first level of the MDB-tree with tree identifier $(\emptyset)$.

In each B$^+$-tree, first the position of the key with the highest attribute value is found by only one execution of the random search (see Section 5.1), which starts in a root of the B$^+$-tree and goes through its internal nodes to the required leaf node (the rightmost leaf node in this case).

Afterwards, only the sequential search (see Section 5.1.2) is used, which goes through all the leaf nodes of the B$^+$-tree and sequentially obtains all the keys of the B$^+$-tree in descending order according to their values. Each key is obtained by the iteration of the while-loop (see Algorithm 8, from line 3 to line 13).

For each key obtained, the searchMDBtree procedure is run in the B$^+$-tree on the next level of the MDB-tree, which is referred to by the pointer of this key. In case of the last level of the MDB-tree, the pointer of each key refers to an object array. Finally, all the objects from each object array reached by the recursive procedure are gradually added in the *objects* list.

By the recursive procedure SEARCHMDBTREE we are able to obtain all objects $x \in X$. Moreover, for every new object x obtained from the MDB-tree all its attribute values are known. Let us suppose that the SEARCHMDBTREE procedure is run in the B$^+$-tree on the mth (the last) level of the MDB-tree. Then the B$^+$-tree is determined by its tree identifier $(k_1, ..., k_{m-1})$ and the object array, in which object x is obtained, is determined by key k_m of the B$^+$-tree. The values $A_1(x), ..., A_m(x)$ of attributes $A_1, ..., A_m$ of object x are equal to the values of the keys $k_1, ..., k_{m-1}, k_m$, respectively.

7.2.2 Advantages of TA algorithm

In this section, we focus on the advantages of the TA algorithm (see Section 4.2), which can be adopted for the top-k search in the MDB-tree. The TA algorithm suffices with remembering the best actual k objects, i.e., k objects with the highest evaluated rankings seen so far. For this purpose, the TA algorithm uses a temporal list *top-k*, in which it keeps the best current objects with their scores. Temporal list *top-k* is maintained in descending order according to ranking $score(x)$ of objects, which it contains. We denote *min-k* as the score of the kth best object in the *top-k* list, i.e., the smallest value of $score(x)$ included in *top-k*.

Algorithm 9 SELECTTOPK (choosing of the top-k objects)

```
selectTopK(list of objects objects, aggregate function g, number k)
 1: top-k := ∅;
 2: min-k := 0;
 3: for all x ∈ objects do
 4:     addObject(x, g, k);
 5: end for
 6: return top-k;

procedure addObject(object x, aggregate function g, number k)
 1: score(x) := g(A_1(x), ..., A_m(x));
 2: if score(x) > min-k then
 3:     if top-k.size() = k then top-k.removeKthObject();
 4:     top-k.insertInTheRightPlace(x);
 5:     min-k := top-k.getMin-k();
 6: end if
 7: return ;
```

The pseudo-code depicted in Algorithm 9 describes choosing the best k objects from the *objects* list according to an aggregate function g. The

addObject procedure represents the maintenance of the *top-k* list in descending order according to $score(x)$ of included objects. If object x, which should be added, has $score(x)$ higher than *min-k*, then the kth best object in *top-k* is removed by the removeKthObject procedure and object x is added by the insertInTheRightPlace procedure.

Using this advantage of the TA algorithm, we can improve the exhaustive search of the MDB-tree. Instead of the two steps, i.e., obtaining all the objects and choosing the top-k objects, we can maintain list *top-k* directly during the computation of the deep-first search of the MDB-tree because we know immediately all the attribute values of newly obtained object. For this purpose, we can use the functionality of the procedure addObject from the Algorithm 9 instead of the line 11 in the searchMDBtree procedure from the Algorithm 8. We utilize this improvement in Section 7.3.

7.2.3 Advantages of NRA algorithm

In this section, we focus on the advantages of the NRA algorithm (see Section 4.3), which can be adopted for the top-k search in the MDB-tree. Using the recursive procedure searchMDBtree, we can obtain all objects $x \in X$ and choose the top-k object according to an aggregate function, but Fagin's algorithms are able to find the top-k objects without searching all the objects. This fact is a new motivation for developing a more efficient top-k algorithm based on MDB-tree.

One of the very useful properties of the NRA algorithm is using the score estimations. The NRA algorithm is able to estimate the worst score and the best score of an object, which has some unknown attribute values (Statement 4.3a and Statement 4.3b). These score estimations do not need to be applied during the search of the MDB-tree because we know all the attribute values of each newly obtained object immediately.

On the other hand, we usually search only for the few top-k objects, which occur only in some parts of the MDB-tree. Therefore, it would be useful to know whether or not it is necessary to search some parts of the MDB-tree. For this purpose, we introduce the term *filial objects of a B^+-tree* by the following definition.

Definition 7.2a (filial objects of B^+-tree) *Let X be a set of objects with m attributes $A_1, ..., A_m$ indexed in the MDB-tree. Let T be a B^+-tree on the ith level of the MDB-tree with tree identifier $(k_1, ..., k_{i-1})$. Let X^T be such a subset of X that all objects $x \in X^T$ have the first $i-1$ attribute $A_1, ..., A_{i-1}$ values equal to $k_1, ..., k_{i-1}$, respectively, i.e., $A_j(x) = k_j$ for $j = 1, ..., i-1$. We call the set of objects $X^T \subseteq X$ the filial objects of B^+-tree T.*

In other words, the filial objects of B$^+$-tree T can be perceived as the objects which are available from B$^+$-tree T or can be reached by using the recursive procedure SEARCHMDBTREE starting at the B$^+$-tree T. Consequently, the filial objects of the B$^+$-tree on the first level of the MDB-tree are the whole set of objects X. For example, in Figure 7.1, objects x_A, x_B, x_C and x_D are the filial objects of B$^+$-tree T_4 with tree identifier (0.0), i.e., objects x_A, x_B, x_C and x_D constitute the subset of objects X^{T_4}.

As we mentioned before, we do not need to estimate the score of obtained objects from the MDB-tree. Nevertheless, we can estimate the score of an object not yet known. Analogically to the NRA algorithm, we assume that aggregate function g is monotone. More specifically, we assume that g is nondecreasing (see Section 3.4.2). Furthermore, at the beginning of this chapter in the Section 7.1 we assumed that values of attributes A_1, ..., A_m are real numbers from the interval $[0, 1]$, i.e., each $D_{A_i}^{act} \subseteq [0, 1]$.

Based on these assumptions, we introduce the *best ranking of the B$^+$-tree* as a score estimation of filial objects of the B$^+$-tree by the following definition.

Definition 7.2b (best ranking of B$^+$-tree) *Let g be a nondecreasing aggregate function with m variables $p_1, ..., p_m$. Let T be the B$^+$-tree on the ith level of the MDB-tree with tree identifier $(k_1, ..., k_{i-1})$. Let X^T be a set of filial objects of B$^+$-tree T. Then we denote the best ranking of B$^+$-tree T as $Best(T)$, which is defined by g with a substitution as $Best(T) = g(p_1, ..., p_m)$, where $p_j = k_j$ for $j = 1, ..., i-1$ and $p_j = 1$ for $j = i, ..., m$, i.e.,*

$$Best(T) = g(k_1, ..., k_{i-1}, 1, ..., 1).$$

We understand the best ranking of B$^+$-tree T as the estimation of the upper bound score of each object $x \in X^T$ not yet known. More formally, the following statement expresses this relationship.

Statement 7.2c (upper bound score) *Let T be a B$^+$-tree of the MDB-tree. Let X^T be set of filial objects of B$^+$-tree T. Then the following holds:*

$$\forall x \in X^T : score(x) \leq Best(T).$$

Proof 7.2c *If we suppose B$^+$-tree T on the ith level of the MDB-tree with tree identifier $(k_1, ..., k_{i-1})$, then all the filial objects have values of the first $i-1$ attributes the same and equal to the values $k_1, ..., k_{i-1}$, respectively. Furthermore, if we assume the attribute domains as the unit interval, the values of the remaining attributes can be at most 1. Because we supposed that the aggregate function is nondecreasing, the Statement 7.2c holds.* □

We note that the restriction of the attribute domains into the unit interval is not crucial for Definition 7.2b and Statement 7.2c. In case of another attribute domain, the maximum value of the domain is used instead of 1.

During the execution of the recursive search of the MDB-tree (used in Algorithm 8), each B$^+$-tree of the MDB-tree is searched by the recursive procedure SEARCHMDBTREE. Let us suppose that the SEARCHMDBTREE procedure runs in B$^+$-tree T on the ith level of the MDB-tree. It obtains the keys from the leaf nodes of B$^+$-tree T step by step. Every key p obtained has the corresponding pointer $\rho(p)$, which refers to another node of the MDB-tree (see Notation 7.1b), where the SEARCHMDBTREE procedure should be run. This node is B$^+$-tree P on the $(i+1)$ level of the MDB-tree, which has a set of filial objects X^P or the object array P including a set of objects X^P in case of the last level of the MDB-tree, i.e., $i = m$.

According to Statement 7.2c, it is possible to estimate the upper bound score $Best(P)$ for each object $x \in X^P$. Let us suppose that the list *top-k* (depicted in Algorithm 9) already includes the best k objects and the score of the kth best object *min-k* is known. Then, for key p it is possible to decide whether or not it is desirable to continue to the other parts of the MDB-tree including set of objects X^P. More formally, the following statement expresses this relationship.

Statement 7.2d (score of filial objects) *Let p be the key in B$^+$-tree T with identifier $(k_1, ..., k_{i-1})$ chosen in a run of the* SEARCHMDBTREE *procedure. Pointer $\rho(p)$ of key p refers to B$^+$-tree P with identifier $(k_1, ..., k_{i-1}, p)$ on the next level of the MDB-tree (or to the object array). Let min-k be the ranking of the kth best object in list top-k. Then the following holds:*

$$Best(P) \leq \textit{min-k} \Rightarrow \forall x \in X^P : score(x) \leq \textit{min-k}.$$

Proof 7.2d *When pointer $\rho(p)$ refers to P, according to Statement 7.2c, then $\forall x \in X^P : score(x) \leq Best(P)$ holds. If list top-k already contains the best k objects, the min-k value is known. $Best(P) \leq$ min-k implies for each $x \in X^P$ that $score(x) \leq Best(P) \leq$ min-k, and thus $score(x) \leq$ min-k.* □

Consequence 7.2d *When $\rho(p)$ refers to P with identifier $(k_1, ..., k_{i-1}, p)$ and at the same time $Best(P) \leq$ min-k holds, none of the objects from X^P is going to have a better score than the kth object from top-k. So it is not meaningful to get objects from P, and it is not necessary to start the* SEARCHMDBTREE *procedure at the B$^+$-tree P (or at the object array P).*

If we search for the few top-k objects in the MDB-tree, these objects occur only in some parts of the MDB-tree. By using Consequence 7.2d, we can skip such parts of the MDB-tree where the objects included do not

have the chance to become members of the *top-k* list because they do not have higher estimation of the upper bound score than *min-k*. The recursive procedure SEARCHMDBTREE does not have to be run in those parts of the MDB-tree.

7.2.4 Utilization of monotonicity

Analogously to Fagin's algorithms, we suppose the use of a monotone aggregate function (see Section 4.1.3) for the top-k search. Moreover, during the execution of the SEARCHMDBTREE procedure in each B$^+$-tree of the MDB-tree, the keys are obtained gradually in descending order according to their values. This fact and the monotonicity we can utilize, and thus we can formulate a more powerful rule for skipping some parts of the MDB-tree than the rule from Consequence 7.2d.

Notice 7.2e (keys in descending order) *Let p be the key in B^+-tree T chosen in an iteration of the while-loop in the* SEARCHMDBTREE *procedure. Let q be the key chosen in the next iteration of the while-loop. Because the keys are obtained from T in descending order according to their values of the attribute indexed in B^+-tree T, then $p \geq q$ holds.*

Statement 7.2f *Let g be a monotone aggregate function with m variables $p_1, ..., p_m$. Let T be the B^+-tree with identifier $(k_1, ..., k_{i-1})$ on the ith level of the MDB-tree, where $i < m$. Let p and q be the keys obtained from T. The pointers $\rho(p)$ and $\rho(q)$ refer to P and Q, respectively, which are the B^+-trees situated on the $(i+1)$th level of the MDB-tree or object arrays when $i = m$. Then the following holds.*

$$p \geq q \Rightarrow Best(P) \geq Best(Q)$$

Proof 7.2f *If P and Q are B^+-trees, for the best ranking of P and Q, $Best(P) = g(k_1, ..., k_{i-1}, p, 1, ..., 1)$ and $Best(Q) = g(k_1, ..., k_{i-1}, q, 1, ..., 1)$ hold. From the monotonicity of the aggregate function follows*

$$p \geq q \Rightarrow g(k_1, ..., k_{i-1}, p, 1, ..., 1) \geq g(k_1, ..., k_{i-1}, q, 1, ..., 1).$$

Then we obtain $p \geq q \Rightarrow Best(P) \geq Best(Q)$.
If P and Q are object arrays, for upper bound score estimations of P and Q, $Best(P) = g(k_1, ..., k_{m-1}, p)$ and $Best(Q) = g(k_1, ..., k_{m-1}, q)$ hold. From the monotonicity of the aggregate function follows

$$p \geq q \Rightarrow g(k_1, ..., k_{m-1}, p) \geq g(k_1, ..., k_{m-1}, q).$$

Then we obtain $p \geq q \Rightarrow Best(P) \geq Best(Q)$. □

Finally, we provide the most important statement with its proof, which concludes the sequence of statements and consequences included in this section. The following statement, denoted as *exit rule*, constitutes the base of an efficient execution of the top-k search in the MDB-tree.

Statement 7.2g (exit rule) *Let T be a B^+-tree with identifier $(k_1, ..., k_{i-1})$ on the ith level of the MDB-tree. Let the keys be obtained one by one by a run of the* searchMDBtree *procedure in B^+-tree T. Let p be the key chosen in an iteration of the while-loop. The pointer $\rho(p)$ refers to B^+-tree P on the $(i+1)$th level of the MDB-tree or to the object array P. Let min-k be the ranking of the kth best object in the top-k list. If $Best(P) \leq$ min-k holds, then the* searchMDBtree *procedure can stop in T.*

Proof 7.2g *Since the keys are obtained from B^+-tree T on the ith level of the MDB-tree MDBT in descending order by the* searchMDBtree(*MDBT*, $(k_1, ..., k_{i-1})$) *procedure, according to Notice 7.2e, for each key p and the next key q obtained from B^+-tree T the expression $q \leq p$ holds. The pointers $\rho(p)$ and $\rho(q)$ refer to B^+-trees P and Q, respectively, in the next level of MDB-tree or they refer to the object arrays P and Q. According to Statement 7.2f, the expression $Best(P) \geq Best(Q)$ holds. If we suppose that $Best(P) \leq$ min-k then, according to Statement 7.2c, we receive the expression*

$$\forall x \in X^Q : score(x) \leq Best(Q) \leq Best(P) \leq \text{min-}k.$$

Thus, it is possible to estimate the upper bound score of the filial objects of Q despite the fact that these objects are not yet known. Their ranking is not higher than the min-k. Therefore, for all the following keys $q \leq p$ obtained from T it is not necessary to access B^+-tree Q (or object array Q) by the searchMDBtree(*MDBT*, $(k_1, ..., k_{i-1}, q)$) *procedure because no such object $x \in X^Q$ has the chance to become a member of the top-k list. It means that it is not necessary to obtain other keys from B^+-tree T and the* searchMDBtree(*MDBT*, $(k_1, ..., k_{i-1})$) *procedure executed in B^+-tree T can stop.* □

7.3 MD algorithm

In this section, we introduce the *MD algorithm* (multidimensional algorithm), which can efficiently search for the top-k objects in the MDB-tree with respect to a monotone aggregate function.

First, we focused on the advantages of the TA algorithm and the NRA algorithm, which can be utilized during the recursive search of the MDB-tree by the searchMDBtree procedure. Then we focused on the utilization

of the monotonicity of the aggregate function. Finally, we developed the MD algorithm, which is based on the recursive search of the MDB-tree and utilizes the exit rule for skipping some parts of the MDB-tree from Statement 7.2g. Therefore, the MD algorithm is able to find the top-k objects without searching some parts of the MDB-tree do not have to be searched by the MD algorithm. Thus, the MD algorithm does not need to obtain a significant part of the objects stored in the MDB-tree.

The pseudo-code of MD algorithm

The pseudo-code depicted in Algorithm 10 describes the MD algorithm. The main composition of the MD algorithm is adopted from the Algorithm 8, which describes depth-first search of the MDB-tree by the SEARCHMDB-TREE procedure. The MD algorithm uses the FINDTOPK procedure instead of the SEARCHMDBTREE procedure. Each instance of the FINDTOPK procedure is run in one particular B$^+$-tree T on the ith level of the MDB-tree with tree identifier $(k_1, ..., k_{i-1})$. The MD algorithm starts in the B$^+$-tree on the first level of the MDB-tree with tree identifier $(\emptyset)$.

First, B$^+$-tree T is initialized (or activated) by the ACTIVATEBPLUSTREE procedure, which finds the position of the key with the highest attribute value by only one execution of the random search (see Section 5.1), i.e., the rightmost key in the B$^+$-tree. Afterwards, the FINDTOPK procedure uses the GETNEXTKEY procedure, which sequentially obtains the keys of the B$^+$-tree in descending order according to their values and uses only the sequential search (see Section 5.1.2) of B$^+$-tree T. Each key is obtained by one iteration of the while-loop (see Algorithm 10, from line 3 to line 24 of the FINDTOPK procedure).

For each newly obtained key from B$^+$-tree T, the FINDTOPK procedure processes the part of the MDB-tree which is referred to by the pointer of this key. The FINDTOPK procedure includes the exit rule from Statement 7.2g (see Algorithm 10, line 7 of the FINDTOPK procedure). By the exit rule it is possible to skip the part of the MDB-tree.

If the part of the MDB-tree includes only objects which do not have the chance to become members of the *top-k* list, the part is skipped. If the pointer of the key refers to a B$^+$-tree on the next level of the MDB-tree, the FINDTOPK procedure is run in the B$^+$-tree. Otherwise, the pointer of the key refers to an object array.

We know immediately all the attribute values of each newly obtained object. Moreover, all the objects of the object array have the same ranking because all the keys constituting the part from the root of the MDB-tree to the object array are known. In other words, we can perceive the object

Algorithm 10 MD algorithm (the top-k search in the MDB-tree)

MD(MDB-tree *MDBT*, aggergate function g, number k)

1: *top-k* := $\emptyset$;
2: *min-k* := 0;
3: findTopK(*MDBT*, ($\emptyset$), g, k);
4: **return** *top-k*;

procedure findTopK(MDB-tree *MDBT*, tree identifier $(k_1, ..., k_{i-1})$, aggergate function g, number k)

1: T := B$^+$-tree with tree identifier $(k_1, ..., k_{i-1})$ on the ith level of *MDBT*;
2: activateBplusTree(T);
3: **while** $\exists$ the next key $k_i \in T$ **do**
4: p := getNextKey(T);
5: P := the node of *MDBT* referred to by $\rho(p)$;
6: $Best(P) := g(k_1, ..., k_{i-1}, p, 1, ..., 1)$;
7: **if** $Best(P) \leq$ *min-k* **then**
8: **return** ; // *according to Statement 7.2g*
9: **end if**
10: **if** P is B$^+$-tree **then**
11: findTopK(*MDBT*, $(k_1, ..., k_{i-1}, p)$, g, k);
12: **end if**
13: **if** P is object array **then**
14: // *Each* $x \in P$ *has the same* $score(x)$, *because* $i = m$.
15: $score(P) := g(k_1, ..., k_{m-1}, p)$;
16: **while** $\exists$ the next object $x \in P$ **and** $score(P) >$ *min-k* **do**
17: Obtain the next object $x \in P$;
18: $score(x) := score(P)$;
19: **if** *top-k*.size() = k **then** *top-k*.removeKthObject();
20: *top-k*.insertInTheRightPlace(x);
21: *min-k* := *top-k*.getMin-k();
22: **end while**
23: **end if**
24: **end while**
25: **return** ;

procedure activateBplusTree(B$^+$-tree T)

1: Find the position of key k_i with the highest value of A_i in the leaf nodes of T with tree identifier $(k_1, ..., k_{i-1})$ by the random search;
2: **return** ;

procedure getNextKey(B$^+$-tree T)

1: Choose the next key k_i with the highest value of A_i in the leaf nodes of T with tree identifier $(k_1, ..., k_{i-1})$ by the sequential search;
2: **return** p;

array as a B$^+$-tree on the $(m+1)$th level of the MDB-tree with tree identifier $(k_1, ..., k_m)$.

The score of all the objects in the object array, denoted as $score(P)$, is evaluated (line 15 of the FINDTOPK procedure). The objects included in P are processed as in the TA algorithm. In Section 7.2.2, we focused on the advantage of the TA algorithm which uses the *top-k* list and the *min-k* value for maintaining the top-k objects (see the ADDOBJECT procedure in Algorithm 9). By using the *top-k* list, each object is immediately inspected, whether or not it becomes a member of the *top-k* list.

The GETMIN-K procedure gets the value of the kth best object in the *top-k* list. The INSERTINTHERIGHTPLACE procedure and the REMOVEK-THOBJECT procedure represent operations, which maintain the *top-k* list sorted in descending order according to scores of included objects.

Example 7.2.1 Figure 7.2 depicts fourteen objects with values of two attributes stored in the MDB-tree with two levels. The MD algorithm is searching the best one object according to summation $g(A_1(x), A_2(x)) = A_1(x) + A_2(x)$ as aggregate function g. The MD algorithm starts in B$^+$-tree T_1 on the first level of the MDB-tree and it obtains key 1.0 with the highest value of attribute A_1, which refers to B$^+$-tree T_7 with tree identifier (1.0) on the second level of MDB-tree.

The MD algorithm obtains key 0.5 with the highest value of attribute A_2, which refers to the object array where it obtains object x_K with ranking $score(x_K) = 1.5$. The MD algorithm inserts object x_K the *top-k* list because list *top-k* is empty at the moment. The FINDTOPK procedure obtains the next key 0.4 in B$^+$-tree T_7 by the GETNEXTKEY procedure. This key refers to an object array which has the best ranking smaller than the ranking of the one object in the *top-k* list. The MD algorithm does not have to obtain any object from the object array and it can stop the execution of the FINDTOPK procedure in B$^+$-tree T_7 according to the exit rule from Statement 7.2g.

The MD algorithm continues in B$^+$-tree T_1 and obtains the next key 0.8, which refers to B$^+$-tree T_6 on the second level of the MDB-tree. In B$^+$-tree T_6 the FINDTOPK procedure is run because the best ranking of this B$^+$-tree is $Best(T_6) = 0.8 + 1 = 1.8$ and it is higher than the *min-k*. The MD algorithm obtains key 0.8 with the highest value, which refers to the object array where it obtains object x_M with ranking $score(x_M) = 1.6$. The best object x_K in the *top-k* list is replaced by the x_M because $score(x_M)$ is higher than $score(x_K)$ which is also equal to *min-k*. Then, in B$^+$-tree T_6, the MD algorithm obtains the next key 0.0, which refers to an object array with the best ranking smaller than *min-k*. Therefore the MD algorithm can stop the execution of the FINDTOPK procedure in B$^+$-tree T_6.

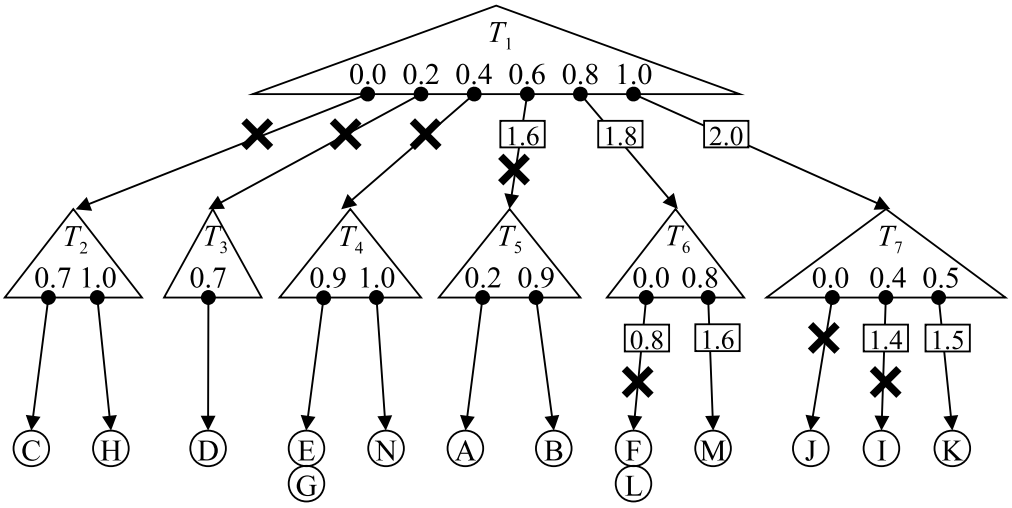


Figure 7.2: Example of searching for the best object in the MDB-tree.

The MD algorithm continues in B$^+$-tree T_1 and obtains the next key 0.6, which refers to a B$^+$-tree on the second level of the MDB-tree. The best ranking of the B$^+$-tree is $Best(T_6) = 0.6 + 1 = 1.6$ and it is not higher than the *min-k*. Therefore the MD algorithm can stop in B$^+$-tree T_1 on the first level of the MDB-tree. Thus, the top-k search in the MDB-tree finishes and the *top-k* list contains the best object x_M according to the aggregate function.

This example showed that the MD algorithm does not need to search all the objects in the MDB-tree. The crosses in Figure 7.2 depict the parts of the MDB-tree which do not have to be accessed by the MD algorithm.

7.3.1 Use of local preferences

The MD algorithm, which we introduced in Section 7.3, allows the user to find the top-k objects in the MDB-tree only according to a monotone aggregate function. In this thesis we focus on the top-k search with regard to the advanced model of user preferences. In addition to the monotone aggregate function, we consider the use of the user's local preference for each attribute (see Section 3.3).

More specifically, for a set of objects X with m attributes, user U expresses his/her preferences as a composition of m local preferences as m fuzzy functions $f_1^U, ..., f_m^U$ and the global preference as a monotone aggregate function g. Then the score of each object $x \in X$ is evaluated by substitution of each attribute value $A_i(x)$ by the values of fuzzy function $f_i^U(x)$ in the

aggregate function g (see Section 3.4.1), i.e.,

$$score(x) = g(f_1^U(x), ..., f_m^U(x)).$$

In Section 6.1, we proposed the model of the sorted list based on a B$^+$-tree which allows us to obtain the objects from the B$^+$-tree sequentially in descending order according to user's local preference. The model is desribed by the BTREELIST algorithm (see Algorithm 4, Section 6.1.3). The local preference for an attribute can be represented by an arbitrary fuzzy function. In fact, the model is based on the idea of obtaining the keys from the B$^+$-tree sequentially in descending order according to their values evaluated by the fuzzy function. To recapitulate, the domain of the attribute is divided according to local extrema of the fuzzy function into some disjoint intervals in which the fuzzy function is monotone. The keys are obtained concurrently from all these intervals step by step, the key with the highest value of fuzzy function being choosen in each step.

During the execution of the MD algorithm, the procedure FINDTOPK is run in one particular B$^+$-tree with identifier $(k_1, ..., k_{i-1})$ on the ith level of the MDB-tree. The keys situated on the leaf level of the B$^+$-tree are obtained sequentially in descending order according to their values of attribute A_i. In notation of the Fagin's algorithms, the keys are obtained as the key-value pairs $\{p, A_i(p)\}$. For the support of the local preferences, it is necessary to obtain the keys from the B$^+$-tree as the key-value pairs $\{p, f_i^U(p)\}$ in descending order according to their values $f_i^U(p)$ of local preference f_i^U.

Since the MD algorithm uses the MDB-tree composed of the B$^+$-trees, we can apply the local preferences during the execution of the top-k search in the MDB-tree by the utilization of the proposed model of sorted list based on the B$^+$-tree. Each B$^+$-tree on the ith level of the MDB-tree only contains keys with values of attribute A_i, and thus the keys have to be obtained in descending order according to arbitrary fuzzy function f_i^U.

We developed the MDLP algorithm as an extension of the MD algorithm, which is able to find the top-k objects in the MDB-tree with the support of local user preferences. The MDLP algorithm enables user U to find the top-k objects according to his/her global preference and local preferences assigned as aggregate function g and fuzzy functions $f_1^U, ..., f_m^U$.

Analogously to the MD algorithm, the MDLP algorithm is based on the recursive search of the MDB-tree and utilizes the rule from Statement 7.2g, and thus it is able to find the top-k objects without searching the whole MDB-tree. Compared to the MD algorithm, Definition 7.2b concerning the best ranking of an B$^+$-tree has to be slightly changed. Instead of the original expression $Best(T) = g(k_1, ..., k_{i-1}, 1, ..., 1)$, for B$^+$-tree T with tree identifier

with tree identifier $(k_1, ..., k_{i-1})$ on the ith level of the MDB-tree, we use expression

$$Best(T) = g(f_1^U(k_1), ..., f_{i-1}^U(k_{i-1}), 1, ..., 1),$$

by which we apply the local preferences. This change affects Statement 7.2d, by which the upper bound score estimation of the filial object of B$^+$-tree T is defined, and Statement 7.2g including the rule for skipping some parts of the MDB-tree. Despite the use of the local preferences, the formulation of the Statement 7.2d and Statement 7.2g does not have to be changed for the utilization in the MDLP algorithm, since the aggregate function has to be monotone too.

The pseudo-code of MDLP algorithm

The pseudo-code depicted in Algorithm 11 describes the MDLP algorithm. The main composition of the MDLP algorithm is adopted from the Algorithm 10. The MDLP algorithm has local user's preferences $f_1^U, ..., f_m^U$ as additional input parameters, compared to the MD algorithm. The MDLP algorithm uses the FINDTOPKLP procedure, which is run in one particular B$^+$-tree T on the ith level of the MDB-tree with tree identifier $(k_1, ..., k_{i-1})$.

First, the B$^+$-tree T is initialized by the CREATECANDIDATES procedure, which is adopted from the BTREELIST algorithm (see Algorithm 4, Section 6.1.3). The CREATECANDIDATES procedure creates the disjoint intervals according to the course of the local preference function and in each interval finds the key with the highest value of the local preference. These keys we call the candidate set and we denote them as C (see Section 6.1.3).

Afterwards, the FINDTOPKLP procedure uses the GETNEXTPAIR procedure, which is also adopted from the BTREELIST algorithm. The GETNEXTPAIR procedure obtains the keys of the B$^+$-tree T sequentially on the ith level of the MDB-tree in descending order according to their values of local preference for attribute A_i. Compared to the BTREELIST algorithm, the GETNEXTPAIR procedure in the FINDTOPKLP procedure obtains the keys from the leaf nodes of the B$^+$-tree as the key-value pairs $\{p, f_i^U(p)\}$ instead of the object-value pairs because each key of the B$^+$-tree refers to different parts of the MDB-tree on the next level of the MDB-tree and the objects (or object arrays) are only referred to from the last level of the MDB-tree. If the next key, as the key-value pair $\{p, f_i^U(p)\}$, is obtained by the GETNEXTPAIR procedure, this key is removed from candidate set C. If it is possible, a new key is added into C, which is the next best key in the same interval of leaf level of the B$^+$-tree, where key p was obtained. Each key-value pair is obtained by one iteration of the while-loop (see Algorithm 11, from line 3 to line 24 of the FINDTOPKLP procedure).

Algorithm 11 MDLP algorithm (MD algorithm with local preferences)

MDLP(MDB-tree *MDBT*, aggergation g, local preferences $f_1^U, ..., f_m^U$, number k)

1: *top-k* := $\emptyset$;
2: *min-k* := 0;
3: FINDTOPKLP(*MDBT*, ($\emptyset$), g, $f_1^U, ..., f_m^U$, k);
4: **return** *top-k*;

procedure FINDTOPKLP(MDB-tree *MDBT*, tree identifier $(k_1, ..., k_{i-1})$,
aggergation g, local preferences $f_1^U, ..., f_m^U$, number k)

1: T := B$^+$-tree with tree identifier $(k_1, ..., k_{i-1})$ on the ith level of *MDBT*;
2: C := CREATECANDIDATES(T, f_i^U);
3: **while** $C \neq \emptyset$ **do**
4: $\{p, f_i^U(p)\}$:= GETNEXTPAIR(T, f_i^U, C);
5: P := the node of *MDBT* referred to by $\rho(p)$;
6: $Best(P) := g(f_1^U(k_1), ..., f_{i-1}^U(k_{i-1}), f_i^U(p), 1, ..., 1)$;
7: **if** $Best(P) \leq$ *min-k* **then**
8: **return** ; // *according to Statement 7.2g*
9: **end if**
10: **if** P is B$^+$-tree **then**
11: FINDTOPKLP(*MDBT*, $(k_1, ..., k_{i-1}, p)$, g, $f_1^U, ..., f_m^U$, k);
12: **end if**
13: **if** P is object array **then**
14: // *Each* $x \in P$ *has the same* $score(x)$, *because* $i = m$.
15: $score(P) := g(f_1^U(k_1), ..., f_{m-1}^U(k_{m-1}), f_m^U(p))$;
16: **while** $\exists$ the next object $x \in P$ **and** $score(P) >$ *min-k* **do**
17: Obtain the next object $x \in P$;
18: $score(x) := score(P)$;
19: **if** *top-k*.SIZE() = k **then** *top-k*.REMOVEKTHOBJECT();
20: *top-k*.INSERTINTHERIGHTPLACE(x);
21: *min-k* := *top-k*.GETMIN-K();
22: **end while**
23: **end if**
24: **end while**
25: **return** ;

procedure CREATECANDIDATES(B$^+$-tree T, local preference f^U)

1: Create n disjoint intervals $I_1, ..., I_n$ in T according to the course of f^U;
2: On each interval find key p with the highest value of $f^U(p)$ and add p into C;
3: **return** C ;

procedure GETNEXTPAIR(B$^+$-tree T, local preference f^U, candidates C)

1: Choose the next key p with the highest value of $f^U(p)$ in C of T;
2: If possible, replace key p in C by the next key from the same interval in T;
3: **return** $\{p, f^U(p)\}$;

For each newly obtained key p from B$^+$-tree T, the FINDTOPKLP procedure processes the part P of the MDB-tree which is referred to by pointer $\rho(p)$ of key p. First, the best ranking $Best(P)$ is evaluated by using the local preferences (line 6 of the FINDTOPKLP procedure). Then the exit rule from Statement 7.2g is checked. It means that if a part of the MDB-tree includes only objects that do not have the chance to become members of the *top-k* list, the part is skipped. Otherwise, the part of the MDB-tree P is inspected.

If P is the B$^+$-tree in the next level of the MDB-tree, a new instance of the FINDTOPKLP procedure is run in the B$^+$-tree. When the the FINDTOPK procedure processes the B$^+$-tree of the mth (the last) level of the MDB-tree, P is an object array. All the objects of the object array have the same ranking because all the keys constituting the part from the root of the MDB-tree to the object array are known. It means that all attribute values of the objects are known. Then the score of all the objectsin the object array, denoted as $score(P)$, is evaluated by using the local preferences (line 15 of the FINDTOPKLP procedure). The objects included in an object array are processed as in the TA algorithm and each object is immediately inspected, whether or not it becomes a member of the *top-k* list.

The GETMIN-K procedure gets the value of the kth best object in the *top-k* list. The INSERTINTHERIGHTPLACE procedure and the REMOVEKTHOBJECT procedure represent operations which maintain the *top-k* list sorted in descending order according to scores of included objects. The rest of the FINDTOPKLP procedure is the same as the FINDTOPK procedure in the MD algorithm.

Example 7.2.2 To understand the MDLP algorithm better, we provide an example of searching for the best three objects according to the user preferences in the MDB-tree. Let us assume a set of objects X with three attributes A_1, A_2, A_3 indexed in the MDB-tree. User U sets up his/her local preferences for attributes A_1, A_2, A_3 by three fuzzy functions f_1^U, f_2^U, f_3^U, respectively, and his/her global preference by summation g, which is defined by expression $g(p_1, p_2, p_3) = p_1 + p_2 + p_3$ and it is naturally a monotone aggregate function. Figure 7.3 depicts the courses of fuzzy functions f_1^U, f_2^U, f_3^U.

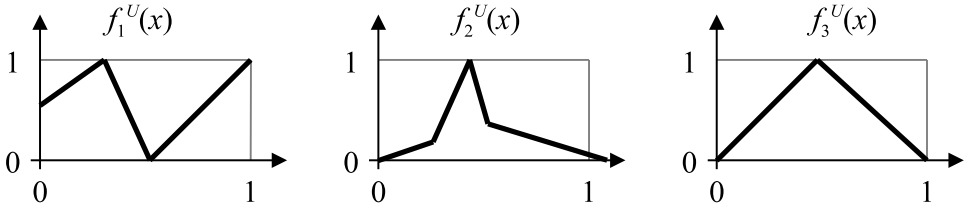


Figure 7.3: Local preferences of a user for three different attributes.

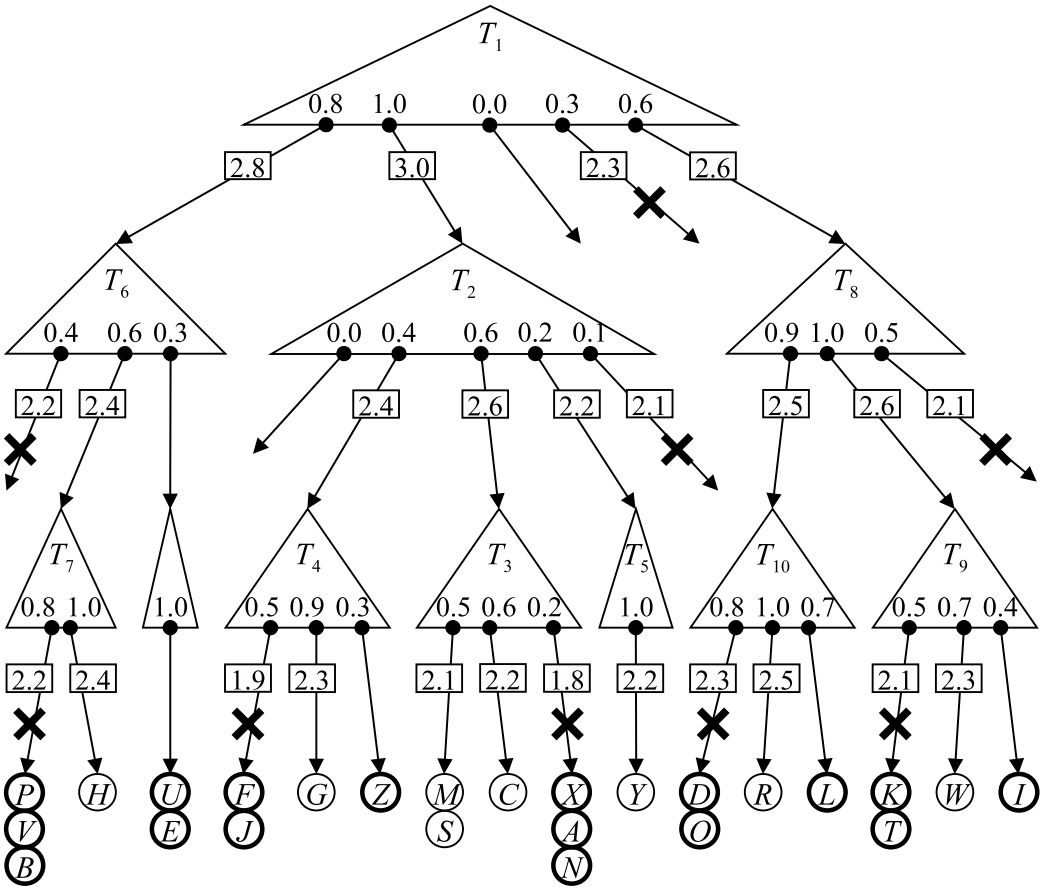


Figure 7.4: Example of searching for top three objects in the MDB-tree.

Figure 7.2 depicts a part of the MDB-tree with three levels. This depicted part allows us to explain the progress of the MDLP algorithm during the search for the best three objects $x \in X$ according to a ranking function defined as $score(x) = f_1^U(k_1) + f_2^U(k_2) + f_3^U(k_3)$, where p_1, p_2, p_3 are keys obtained from the B-trees on the first, the second and the third level of the MDB-tree. The ranking function expresses preferences of user U.

The MDLP algorithm starts the top-3 search in B$^+$-tree T_1 on the first level of the MDB-tree and it obtains the key with the highest value of f_1^U. In this case, it is a key k_1^{1st} with a value of attribute A_1 and with value $f_1^U(k_1^{1st}) = 1.0$, which refers to B$^+$-tree T_2 on the second level of MDB-tree (see Figure 7.4). The value of the attribute is not too important for us and it is only used as an item of the tree identifiers, which act as the navigation of the MDLP algorithm. Therefore only the values of the fuzzy functions are depicted in the figure. That is why the value 1.0 is not the rightmost value in the leaf level of the B$^+$-tree T_1 in the figure. The values depicted on the

leaf level of the B$^+$-trees on the ith level of the MDB-tree correspond to the course of the user's fuzzy function f_i^U depicted in Figure 7.3.

Analogously, as in the first level of the MDB-tree, the MDLP algorithm finds a key k_2^{1st} with value $f_2^U(k_2^{1st}) = 0.6$ in B$^+$-tree T_2 and a key k_3^{1st} with value $f_3^U(k_3^{1st}) = 0.6$ in B$^+$-tree T_3. Key k_3 refers to the object array, where object x_C is obtained. Then the next key k_3^{2nd} with value $f_3^U(k_3^{2nd}) = 0.5$ is obtained in T_3, and thus we get objects x_M and x_S. In that moment the *top-k* list contains three objects and the *min-k* value is equal to the $score(x_S) = 2.1$. Then the next key k_3^{3rd} with value $f_3^U(k_3^{3rd}) = 0.2$ is obtained in B$^+$-tree T_3 and it refers to an object array P. The value of $Best(P)$ is 1.8, which is depicted in the figure as a small rectangle with a number, and it is not higher than *min-k*. Therefore, according to the Statement 7.2g, the MDLP algorithm can stop in T_3. In the figure, it is represented as the highlighted cross over the pointer to the object array containing objects x_X, x_A and x_N, which are not obtained by the MDLP algorithm.

The MDLP algorithm continues in B$^+$-tree T_2, where the next key k_2^{2nd} with value $f_2^U(k_2^{2nd}) = 0.4$ is obtained. By using this key the B$^+$-tree T_4 is searched, and thus object x_G, as only one relevant object, is found and added in the right place into the *top-k* list. Concurrently, object x_S is removed from *top-k* list. Then the next key k_2^{3rd} with value $f_2^U(k_2^{3rd}) = 0.2$ is obtained in B$^+$-tree T_2. By using this key B$^+$-tree T_5 is searched and object x_Y is found and it replaces object x_M in the *top-k* list. The value of *min-k* is increased to 2.2. Then the next key k_2^{4th} with value $f_2^U(k_2^{4th}) = 0.1$ is obtained in B$^+$-tree T_2. This key refers to the B$^+$-tree in the third level of the MDB-tree which has the best ranking smaller than *min-k*. Therefore the MDLP algorithm can stop in T_2.

The MDLP algorithm continues in B$^+$-tree T_1, where the next key k_1^{2nd} with value $f_1^U(k_1^{2nd}) = 0.8$ is obtained. By using this key B$^+$-tree T_6 is searched, and thus only one relevant object, x_H, is found and added to the right place in the *top-k* list. After leaving B$^+$-tree T_6, the *top-k* list contains objects x_H, x_G, x_C, respectively, and $min\text{-}k = score(x_C) = 2.2$.

The MDLP algorithm continues in B$^+$-tree T_1, where the next key k_1^{3rd} with value $f_1^U(k_1^{3rd}) = 0.6$ is obtained. Then B$^+$-trees T_8 and T_9 are searched and object x_W is added into the *top-k* list, and thus x_C is removed. Afterwards, B$^+$-tree T_{10} is searched and object x_R is added into the *top-k* list, and thus x_W is removed. After leaving B$^+$-tree T_8, the *top-k* list contains objects x_R, x_H, x_G, respectively, and $min\text{-}k = score(x_G) = 2.3$.

The MDLP algorithm continues in B$^+$-tree T_1, where the next key k_1^{4th} with value $f_1^U(k_1^{3rd}) = 0.6$ is obtained. The best ranking of the B$^+$-tree, to which key k_1^{3rd} refers, is not higher than *min-k*. Therefore the MDLP algorithm can stop in T_1, and thus the top-3 search in the MDB-tree finishes.

The MDPL algorithm and the MD algorithm are able to find the top-k objects in the MDB-tree and they do not need to obtain a significant part of the objects stored in the MDB-tree. For illustration, the objects which were not obtained by the MDPL algorithm during the search for top-3 object for user U are depicted in Figure 7.4 as highlighted circles. Moreover, many other objects not obtained by the MDPL algorithm and also some parts of the MDB-tree are not depicted in the figure.

7.4 Experiments

We have tested the MD algorithm and compared it with the TA algorithm and the NRA algorithm. For the purpose of experimental evaluation, we have implemented the MDLP algorithm using of the local preferences and top-k algorithms TA and NRA using the lists based on B^+-trees, i.e., the TALP algorithm and the NRALP algorithm (see Algorithm 6 and Algorithm 5).

The implementation was developed in Java. The main research point was to estimate the number of accesses into data structures during calculation of algorithms. The corresponding data structures for the algorithms were kept in the memory, and thus independently of the disk environment. It was a sufficient for our experimental evaluation of the algorithms because then their computation is independent on the characteristic of a external storage.

7.4.1 Distribution of attribute values

For testing three sets of 100000 objects with 5 attribute values with normal, uniform, and exponential distribution of attribute values were used. Obtaining one attribute value of one object we perceive as *one access* into corresponding data structure to an algorithm. First, we used simple user preferences, because the local preferences can change the distribution of attribute values. As user's local preferences we used fuzzy functions whose course was based on an increasing linear function with respect to the attribute values. As the aggregate function an arithmetic average was used. Figure 7.5, Figure 7.6 and Figure 7.7 show results of the tests for normal, uniform and exponential distribution, respectively.

The best results were achieved with the MD algorithm for the set of objects with normal and uniform distribution of the attribute values. The test showed that for the uniform distribution, the MD algorithm achieved much better results than the other tested algorithms. Searching for the best object needed approximately 10 times fewer number of accesses than the TA algorithm and the NRA algorithm.

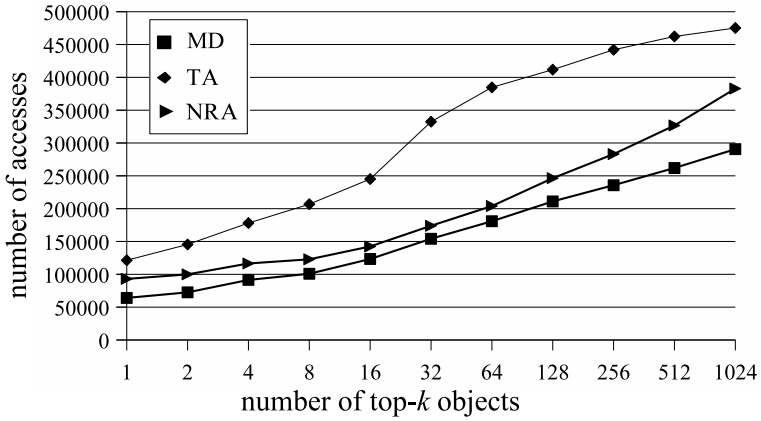


Figure 7.5: Normal distribution of the attribute values.

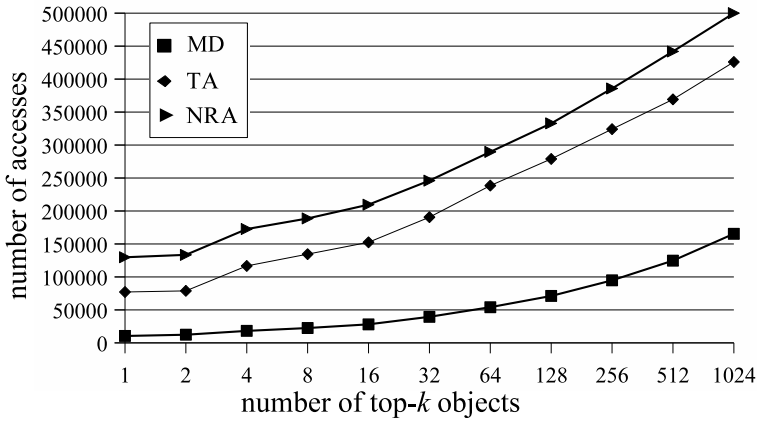


Figure 7.6: Uniform distribution of the attribute values.

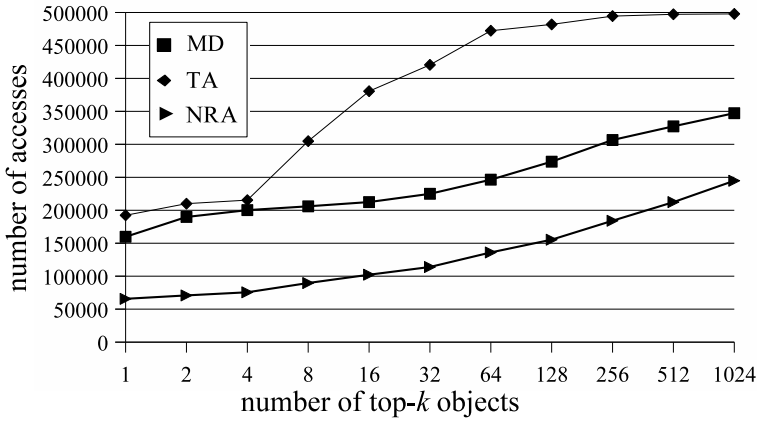


Figure 7.7: Exponential distribution of the attribute values.

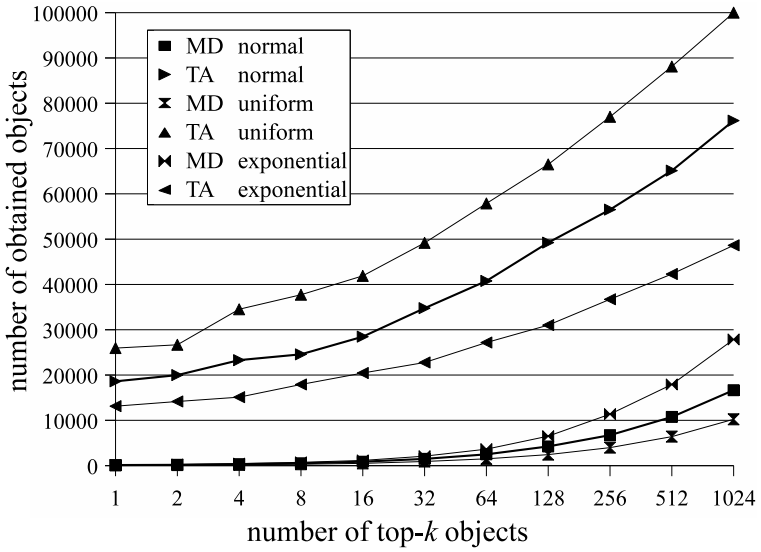


Figure 7.8: The number of obtained objects during the computation.

For the MD algorithm and the TA algorithm, it is also possible to determine the number of obtained objects from associated data structures. Figure 7.8 illustrates the results of this comparison. The most significant difference in the number of obtained objects between the TA algorithm and the MD algorithm has occurred for uniform distribution of the attribute values. For searching the top-k objects, the MD algorithm needed to obtain over 400 times fewer objects than the TA algorithm.

Figure 7.9 illustrates the execution time of the MD algorithm and the TA algorithm for different distributions of attribute values. For the TA algorithm and the MD algorithm with $k = 1024$ we achieved a time of less than one second. Testing has shown that the NRA algorithm is slower and its execution time took several seconds. Therefore we do not show the comparison with the NRA algorithm. This fact was caused by frequent recalculation of $Worst(x)$ ranking and $Best(x)$ ranking of the objects, which is the main disadvantage of the NRA algorithm implemented in memory, compared to the TA algorithm, which uses the random access.

These experiments have shown that the results of the MD algorithm are comparable with those achieved by Fagin's algorithms. We can conclude that the MD algorithm is more suitable in the case of uniform distribution of the attributes values as the tested Fagin's algorithms. Moreover, the MD algorithm achieves much better results than other tested algorithms for a small number of the top-k objects only if k is less than 32, which is the most common number of the best results required by a user.

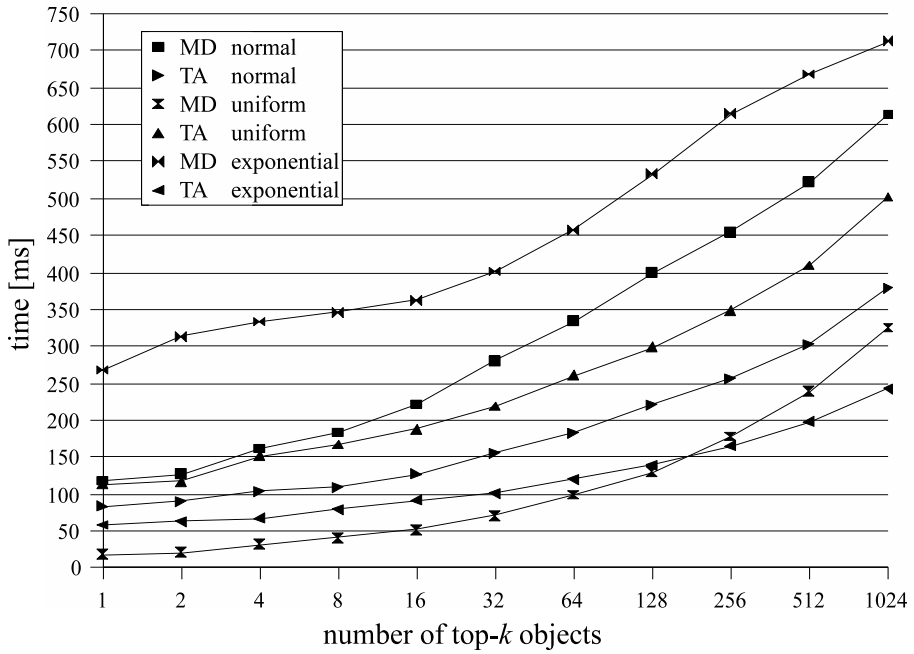


Figure 7.9: Computation time of the MD algorithm and TA algorithm.

7.4.2 Different domain sizes

In the previous test we used data sets, which had the same sizes of actual attribute domains. In this test, we used nine different data sets. We examined the MD algorithm for each data sets with different sizes of the actual attribute domains.

Because we used a set of 100 000 objects, we used five different sizes of the attribute domain sizes. We chosen the following sizes: 10, 100, 1000, 10 000 and 100 000. For five attributes and five domain sizes, it would be possible to create a $5^5 = 3125$ different data sets, but for the purposes of this test only the most representative data sets was chosen. Figure 7.10 shows the result of the test. In this figure we used the following notation: $a_b_c_d_e$, where for the first attribute A_1 the size of its actual domain is 10^a, for A_2 it is 10^b, for A_3 it is 10^c, for A_4 it is 10^d and for A_5 it is 10^b.

The test clearly showed that the size of actual domains of the attributes and their order in particular attributes significantly affect the efficiency of the MD algorithm. The MD algorithm achieves the best results when all the attributes have small-sized actual domain. The best result of the MD algorithm is depicted in Figure 7.10 as the graph of the "1_1_1_1_1" case, i.e., all five attributes had the size of actual domain equal to 10.

When the sizes of the actual attribute domains are given, the MD algo-

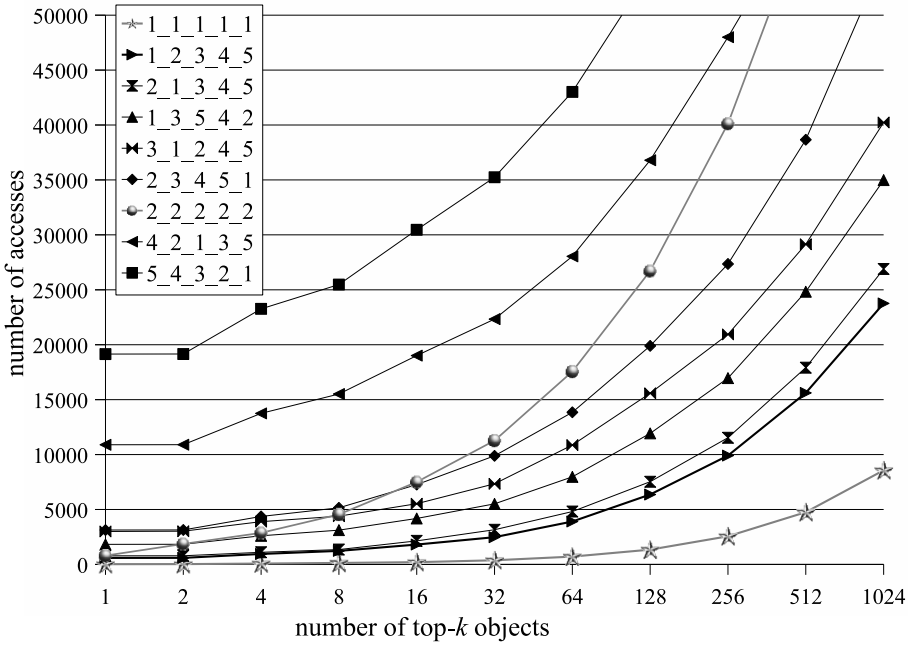


Figure 7.10: Computation time of the MD algorithm and TA algorithm.

rithm achieves the best results when the actual domain sizes are sorted in ascending order with respect to the order of attributes on particular levels in the MDB-tree. In such a case, the MD algorithm can more quickly decide that it is not necessary to obtain the next keys from some B^+-tree on a higher level of the MDB-tree and thus the MD algorithm can skip the bigger parts of the MDB-tree according to the exit rule from Statement 7.2g. The biggest difference in the results of the MD algorithm is between the "1_2_3_4_5" case and the "5_4_3_2_1" case depicted in Figure 7.10.

7.4.3 Various user preferences

We have tested the implemented top-k algorithms by using various settings of user preferences. We can conclude that the local preferences affect the execution of the top-k algorithms minimally. On the other hand, the aggregate function has a significant influence on their execution.

When we use the TALP algorithm and the NRALP algorithm (see Section 6.2), they use the B^+-trees instead of the sorted lists. Additionally to the original Fagin's algorithms TA and NRA, it is necessary to initialize the candidate sets in these B^+-trees according to fuzzy functions, as the local user preferences, before the computation of the algorithms. In case of the MDLP, it is necessary to initialize the candidate set in each B^+-tree, which

is searched by the FINDTOPKLP procedure.

These initializations of the candidate sets require some additional accesses. In most cases, the course of user fuzzy functions of real users is not too complicated. Therefore, the number of these additional accesses is much smaller than the number of subsequent accesses performed by the algorithms. Therefore, the use of local preferences affects the number of performed accesses or the obtained objects only slightly.

The more important fact is that local user preferences change the distribution of the attribute values because each local preference is a projection. Since the efficiency of the MD algorithm (and also Fagin's algorithms) depends on the distribution of attribute values, the efficiency of the MDLP (and also TALP and the NRALP) depends on a projection of the distribution to a new distribution. We can conclude that we do not know how local user preferences affect the efficiency of the algorithms because we cannot predict the local user preferences. On the other hand, if we know the local preferences and the distribution of the attribute values, we can guess how efficient the computation of the MDLP algorithm will be.

The MD algorithm and Fagin's algorithms search for the top-k objects according to a monotone aggregate function. The aggregate function is a function with m variables which correspond to the attributes of the object. Some of the variables can have higher weight or influence than others. The weighted average provides the most typical example.

Our experiments associated with different weights of attributes in the aggregate function have shown that the MD algorithm has the best results when the variables of the aggregate function with the highest weights correspond to the attributes stored on the first levels of the MDB-tree.

7.5 Multiple-ordered MDB-tree

We discovered that the efficiency of the MD algorithm depends on the weight of attributes in the aggregate function assigned by a user. Therefore, we propose the *multiple-ordered MDB-tree* as the useful index structure derived from the MDB-tree. Since we cannot guess particular aggregate function which will be set by the user this index structure method enables us to choose the most effective way of executing the MD algorithm according to given aggregate function with different weights of particular attributes.

According to the authors of the MDB-tree [SO82], the problem of assigning the attributes to the levels of the MDB-tree is itself a fairly complex optimization problem. The authors consider different probabilities of occurrence of the attributes in queries and also different probabilities of updating

operations with regard to a particular order of the attributes, and thus these considerations can cause several conflicting requirements for the optimal assignment of attributes to the levels on the MDB-tree. On the other hand, the ranking of attributes in decreasing order of the probability of their occurrence in a query can improve the efficiency of the search while answering a query.

7.5.1 Ordering of attributes in MDB-tree

In Section 5.3 and Section 5.4, we introduced the MDB-tree and its variant composed of B^+-trees. In this thesis, we perceive the multidimensional data as a set of objects with m attributes A_1, ..., A_m, and we suppose that the objects are indexed in the MDB-tree according to all the attributes. To introduce the MDB-tree, we simply assumed that attributes A_1, ..., A_m, respectively, are situated from the first to the last level of the MDB-tree, i.e., the ith attribute corresponds to the ith level of the MDB-tree.

In general, the order of attributes on particular levels of the MDB-tree can be different. For m attributes there are

$$m! = \prod_{i=1}^{m} m$$

different orders and it is equal to the number of permutations of attributes $A_1, ..., A_m$. It means that for m attributes we can construct $m!$ different MDB-trees, where each one is different with regard to the order of the attributes on particular levels of the MDB-tree.

7.5.2 Double-ordered MDB-tree

Now we introduce a case with two attributes. Let us suppose a set of objects with two attributes A_1 and A_2. The objects can be indexed by using two different MDB-trees. One MDB-tree with the order of attributes A_1 and A_2, respectively, has the values of attribute A_1 on its first level and the values of attribute A_2 on its second level. Another MDB-tree with the order of attributes A_2 and A_1, respectively, has the values of attribute A_2 on its first level and the values of attribute A_1 on its second level.

Nevertheless, both these MDB-trees index the same set of objects, and thus we can perceive these two MDB-trees as one data structure which we denote as *double-ordered MDB-tree*. Figure 7.11 depicts the double-ordered MDB-tree.

If we propose that the objects are indexed in such a way, we can choose the more suitable MDB-tree from the double-ordered MDB-tree for the execution

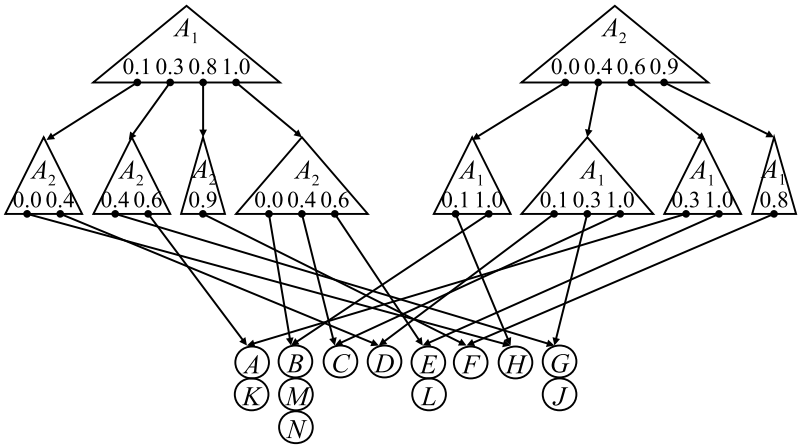


Figure 7.11: Double-ordered MDB-tree.

of the MD algorithm. If a user sets an aggregate function with the higher weight of attribute A_1, we choose the MDB-tree with the order of attributes A_1 and A_2, respectively. Otherwise we choose the other one.

7.5.3 Integration of all permutations of MDB-tree

A more interesting situation occurs when we propose a set of objects with more than two attributes indexed by using MDB-tree. Let us suppose a set of objects with three attributes A_1, A_2 and A_3. The objects can be indexed by using $3! = 6$ different MDB-trees, but it is not necessary to build six different MDB-trees.

Since each MDB-tree is composed of the B^+-trees, we focused on the B^+-trees. We discovered that some of these MDB-trees include the same B^+-trees. This fact is caused by the properties of the filial sets (see Definition 5.3b in Section 5.2.2) in multidimensional data sets. Each B^+-tree corresponds to a key on the previous level of the MDB-tree and this B^+-tree only contains keys which constitute the filial set of the key.

Therefore, it is possible to integrate these six different MDB-trees into one index structure which we denote as *triple-ordered MDB-tree*. The triple-ordered MDB-tree is depicted in Figure 7.12.

Let us suppose a set of objects with four attributes A_1, A_2, A_3 and A_4. The objects can be indexed by using $4! = 24$ different MDB-trees, but it is possible to integrate these 24 different MDB-trees into one index structure which we denote as *quadruple-ordered MDB-tree*. The quadruple-ordered MDB-tree is depicted in Figure 7.13.

Analogously, we can suppose a set of objects with more than four at-

tributes. Then it is possible to build such a corresponding index structure, which we call *multi-ordered MDB-tree*, based on integration of all the cases of the MDB-tree, for the specific number of attributes, i.e., for the number of dimensions.

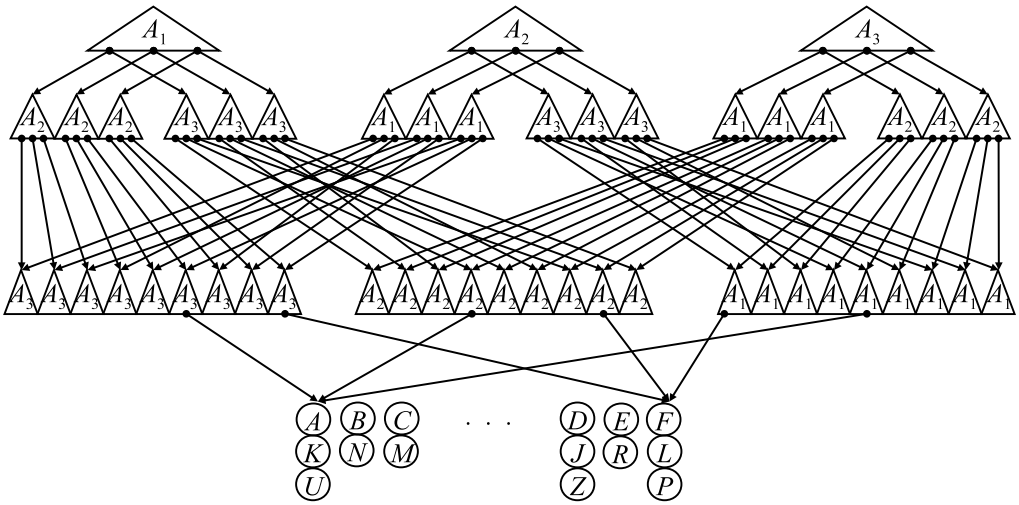


Figure 7.12: Triple-ordered MDB-tree.

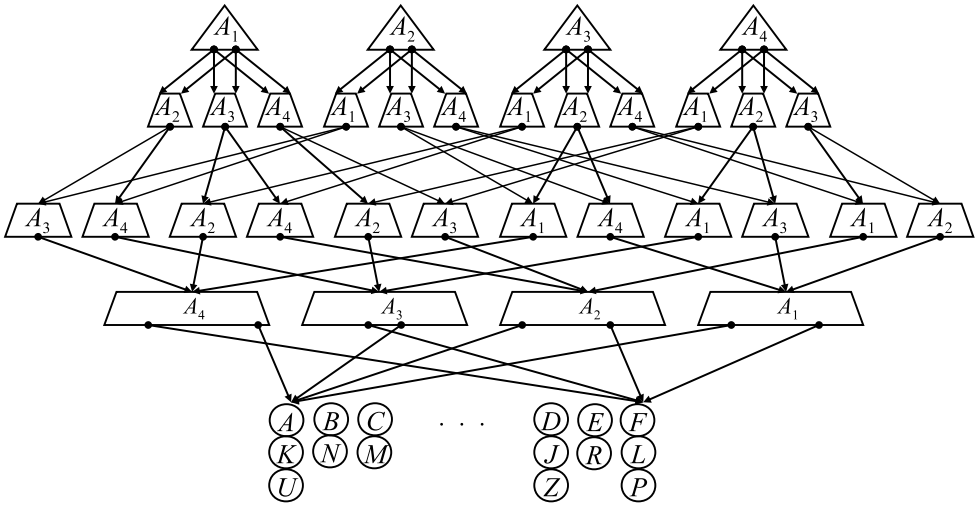


Figure 7.13: Quadruple-ordered MDB-tree.

7.5.4 Use of multiple-ordered MDB-tree

If a set of objects with m attributes is indexed by a multiple-ordered MDB-tree, we can choose the most suitable MDB-tree of the multiple-ordered MDB-tree for the execution of the MD algorithm. When a user sets his/her aggregate function, it is possible to determine the most suitable order of attributes, which is decreasing order according to their weights in the aggregate function. Afterwards, the MD algorithm is run in the MDB-tree of the multiple-ordered MDB-tree corresponding to this decreasing order. Thus, the MD algorithm is always executed in the most efficient way.

A little disadvantage of the multiple-ordered MDB-tree is that it can take a bigger space in the memory used. But nowadays, in the age of gigabytes, this disadvantage can be easily overcome. On the other hand, the multiple-ordered MDB-tree helps us to save space in the memory used. Instead of all the permutations of the MDB-tree we can only use one economical data structure.

Dimension	Ratio	Number of permutations
1	1.00	1
2	2.00	2
3	3.03	6
4	4.25	24
5	5.97	120
6	8.70	720
7	13.37	5 040
8	19.61	40 320
9	30.96	362 880
10	55.88	3 628 800
11	130.22	39 916 800
12	189.42	479 001 600
13	843.45	6 227 020 800
14	1362.42	87 178 291 200
15	2189.54	1 307 674 368 000

Table 7.1: The ratio of the size of the MDB-tree to the size of the multiple-ordered MDB-tree in a memory.

For example, for the number of dimensions from 1 to 15, we calculated the ratio of the size of one MDB-tree to the size of its corresponding multiple-ordered MDB-tree in a memory. Table 7.1 shows the results compared to the number of all the permutations for each dimension. The first column depicts the dimension. The second column depicts the size of the multiple-ordered MDB-tree compared to the original MDB-tree. The third column depicts

the size of all the permutations of the MDB-tree compared to the original MDB-tree, i.e., the number of all its permutations for the dimension.

7.6 Summary

Our experience with the implementation of the MD algorithm has shown that the efficiency of the MD algorithm depends on a particular set of objects and its distribution of the attribute values of the object stored in the MDB-tree. We also tested Fagin's algorithms and the MD algorithm for some real data sets. The achieved results were very similar to the results of the previous tests with artificial data sets because the results were also dependent on the particular distribution of attribute values.

Concerning other associated results, we discovered that the MD algorithm has the best results when the objects and their attribute values stored in the MDB-tree are distributed uniformly (or regularly), i.e., the MDB-tree is regular. In case of any special data sets, which cause that the MDB-tree is degenerated (see Section 5.4.1), the top-k search realized by the MD algorithm can be very inefficient.

Moreover, our experiences with the MD algorithm has shown that the MDB-tree should not be used in applications which result in a degenerated MDB-tree, i.e. one with very small filial set sizes on the last levels. Our results confirm the original idea of the MDB-tree [SO82], which was developed for range queries and for the minimization of the number of accesses into data structures.

If the attributes of a set of objects have different sizes and types of their actual attribute domains, the order of attributes in the MDB-tree is very important. We cannot influence the distribution of attribute values of a given data set, but we can choose the most advantageous variant of the MDB-tree for the set of objects. Another possibility is to use the multiple-ordered MDB-tree.

For the MD algorithm it is better to design the MDB-tree in such a way that the attributes with small-sized actual domains are situated on the first few levels of the MDB-tree and the attributes with big-sized actual domains are situated on the last few levels of the MDB-tree. When most attributes of a given data set have a big-sized actual domain, the use of the MDB-tree with the MD algorithm is not the most suitable solution to the top-k search. In such cases, the use of the TA algorithm or the NRA algorithm is more suitable than the use of the MD algorithm. We focus on this issue in more detail in Chapter 8.

Chapter 8

Top-k Search Based on MDB-tree with Lists

In this chapter, we focus on the use of the MDB-tree with lists for the top-k search. The MDB-tree with lists is a new tree-based data structure described in Section 5.5, which allows us to index a set of objects with more attributes. In the MDB-tree with lists, some attribute values are stored as in the MDB-tree and the others are stored as in Fagin's sorted lists. We developed the *MXT algorithm* [OP09, OP10], which can search for the top-k objects in the MDB-tree with lists according to a monotone aggregate function. The MXT algorithm is based on an integration of the MD algorithm and the TA algorithm since it uses the MDB-tree with lists, which is composed of the MDB-tree and Fagin's sorted lists.

Afterwards, we focus on the top-k search in the MDB-tree with lists according to the model of user preferences based on the local preferences, which enable us to use a non-monotone ranking function. Therefore, instead of the MDB-tree with lists, we propose to use the MDB-tree with groups of B^+-trees described in Section 5.5.1, which is composed solely of B^+-trees. In such a case, we can apply the model of sorted list based on B^+-tree, and thus it is possible to use the local preferences directly during the computation of the top-k search in the MDB-tree with groups of B^+-trees. We developed the MXTLP algorithm as an extension of the MXT algorithm, which is able to search for the top-k objects efficiently by using the local preferences.

The MXT algorithm and the MXTLP algorithm are efficient top-k algorithms because they are able to find the top-k objects without searching the whole corresponding data structure, and thus without obtaining all the objects. Moreover, the algorithms use tree-based data structures, which are independent of user preferences, and it is possible to update these data structures easily and quickly.

First, in Section 8.1, we explain why we use the MDB-tree with lists for the top-k search. In Section 8.2, we focus on the abilities the MDB-tree with lists, which help us to develop the MXT algorithm. In Section 8.3, we introduce how it is possible to use the MD algorithm, the TA algorithm and their mutual integration during the execution of the top-k search in the MDB-tree. In Section 8.4, we present the MXT algorithm and the MXTLP algorithm using the local preferences. Finally, in Section 8.5 we show experimental results of presented algorithms. In Section 8.6, we provide a summary and an analysis of the efficiency of the MXT algorithm.

8.1 Motivation

In this thesis, we differentiate between categorical attribute and numerical attribute as the two basic attribute types. We perceive the categorical attribute as an attribute with a small-sized actual domain and the numerical attribute as an attribute with a big-sized actual domain.

If we suppose a set of object indexed in the MDB-tree and the objects have more numerical attributes with actual attribute domains including large numbers of different attribute values, the MDB-tree can become degenerated (see Section 5.4.1). In such a case, using the MDB-tree is not the best choice because the use of the MD algorithm for the top-k search is not efficient (see Section 7.6). By using the MDB-tree with lists, we can partially solve the problem of the degenerated MDB-tree. The MDB-tree with lists is just designed to index the set of objects with more numerical attributes. In case of real data sets, the occurrence of the more numerical attributes is very common. Therefore, we focus on the top-k search by using the MDB-tree with lists.

8.2 Abilities of MDB-tree with lists

The MDB-tree with lists allows us to index a set of objects X with more attributes $A_1, ..., A_m$ in one data structure. For a set of objects with m attributes, which we want to index by the MDB-tree with lists, we suppose that the attributes $A_1, ..., A_n$ are categorical and attributes $A_{n+1}, ..., A_m$ are numerical. In the MDB-tree with lists, attributes $A_1, ..., A_n$ are stored as the MDB-tree with n levels and attributes $A_{n+1}, ..., A_m$ are stored as groups of $m - n$ Fagin's sorted lists.

The MDB-tree with n levels constitutes the base of the MDB-tree with lists. To explicitly identify each B^+-tree in the MDB-tree, we use again a

tree identifier of the B$^+$-tree. Each B$^+$-tree with tree identifier $(k_1, ..., k_{n-1})$ on the ith level of the MDB-tree includes some values of attribute A_i as the keys in its leaf nodes. For each key k_i, there is a corresponding pointer $\rho(k_i)$ which refers to another node of the MDB-tree.

Since the MDB-tree has n levels, each pointer of a key k_i in a B$^+$-tree on ith level of the MDB-tree, where $i < n$, refers by its pointer $\rho(k_i)$ to a B$^+$-tree on $(i+1)$th level of the MDB-tree. Each pointer of a key k_n in a B$^+$-tree on nth level of the MDB-tree refers by its pointer $\rho(k_n)$ to one group of sorted lists, instead of the associated data (an object or an object array). More exactly, it refers to sorted lists $L_{n+1}, ..., L_m$, and therefore the key k_n should have $m - n$ corresponding pointers $\rho_{n+1}(k_n), ..., \rho_m(k_n)$. For the sake of clarity, we will only assume that pointer $\rho(k_n)$ refers to the group of sorted lists $L_{n+1}, ..., L_m$.

For each key k_n in a B$^+$-tree on the nth level of the MDB-tree, there is a group of $m - n$ lists $L_{n+1}, ..., L_m$. These lists contain the filial set of objects of key k_n, denoted as X^{k_n} (see Definition 5.5a), which is a subset of all the objects stored in the MDB-tree with lists corresponding to key k_n. Each list L_i of this group, where $i = n+1, ..., m$, contains just one object-value pair $\{x, A_i(x)\}$ for each object $x \in X^{k_n}$ All the pairs in list L_i are sorted in descending order according to values $A_i(x)$.

In this section, for the sake of clarity, we assume that the values of attributes $A_1, ..., A_m$ are real numbers from the interval $[0, 1]$, i.e., each $D_{A_i}^{act} \subseteq [0, 1]$. This restriction is not crucial for the purposes of the top-k search in the MDB-tree with lists. Any other attribute types, which have an ordering defined on their domain, can be applied instead of the unit interval.

8.3 Top-k search in MDB-tree with lists

In this section, we first focus on the top-k search in the MDB-tree with lists according to monotone aggregate function g. We attempt to utilize the abilities of the MD algorithm and the TA algorithm because they can find the top-k objects efficiently in the MDB-tree and in the sorted lists, respectively, i.e., the number of objects obtained by these algorithms is much smaller than the number of all the objects stored in corresponding data structures.

8.3.1 Use of MD algorithm

Since the MDB-tree constitutes the top part of the MDB-tree with lists, we can apply the recursive procedure based on depth-first search (see Section 7.2.1). By this procedure the MD algorithm recursively searches for the

top-k objects in the MDB-tree starting at the B$^+$-tree on the first level of the MDB-tree.

In the moment when the recursive procedure obtains key k_n in the B$^+$-tree with tree identifier $(k_1, ..., k_{n-1})$ on nth level of the MDB-tree, the group of lists $L_{n+1}, ..., L_m$ can be reached by the pointer $\rho(k_n)$. This group of lists contains the filial objects X^{k_n} of the key k_n. In that moment, values of the first n attributes $A_1, ..., A_n$ of the filial objects X^{k_n} are known and they are equal to $k_1, ..., k_n$, respectively. The values $k_1, ..., k_{n-1}$ are known from the tree identifier and from the obtained key k_n.

The MD algorithm uses the exit rule from Statement 7.2g for skipping some parts of the MDB-tree, which include such objects that have no chance to become one of the top-k objects. In doing so, the MD algorithm uses the best ranking of the B$^+$-tree (see Definition 7.2b) as a score estimation of filial objects of the B$^+$-tree, which is based on the values included in the tree identifier of the B$^+$-tree. For this purpose, we can perceive group of lists P in the MDB-tree with lists as a node of the MDB-tree on its $(n+1)$ level. Then we can define the *tree identifier of the group of lists* as $(k_1, ..., k_n)$, where value k_n is the key to which the group corresponds and values $k_1, ..., k_{n-1}$ constitute the tree identifier of the B$^+$-tree on nth level of the MDB-tree, where the key k_n was obtained.

Now, we can introduce the *best ranking of the group of lists* as the upper bound score estimation of filial objects X^{k_n} (see Statement 7.2c) stored in the group of lists with tree identifier $(k_1, ..., k_n)$. We denote the best ranking of group of lists P as $Best(P)$. We define $Best(P)$ by using aggregate function g with a substitution as an expression $Best(T) = g(p_1, ..., p_m)$, where $p_j = k_j$ for $j = 1, ..., n$ and $p_j = 1$ for $j = n+1, ..., m$, i.e.,

$$Best(P) = g(k_1, ..., k_n, 1, ..., 1).$$

The MD algorithm uses a list *top-k*, in which it keeps the best current objects with their scores and a value *min-k* as the score of the kth best object in the *top-k* list. Let us consider the moment when the *top-k* includes k objects and the MD algorithm is repeatedly obtaining key p from B$^+$-tree T by the recursive procedure. The MD algorithm compares the *min-k* value and the $Best(P)$ value of B$^+$-tree P or group of lists P which is referred to by the $\rho(p)$. Then the MD algorithm can decide that it is beneficial to search for any other objects in P according to the exit rule. If the $Best(P)$ value is not higher than the *min-k* value, the recursive procedure stops in T. Otherwise, the MD algorithm searches for any other top-k objects in P.

If P is the group of lists, this group of lists has to be inspected whether or not it includes some objects which can become members of the top-k list. Since the group of lists is composed of Fagin's sorted lists, the TA algorithm

can be used, by which it is possible to search for the top-k objects in the lists.

8.3.2 Use of TA algorithm

The TA algorithm (see Section 4.2) is based on the idea of obtaining objects sequentially from the sorted lists by the sorted access and investigating them immediately by the random access whether they can or cannot be the ones of the top-k objects. In the moment when the group of sorted lists $L_{n+1}, ..., L_m$ is reached by the recursive procedure from key k_n, we can apply the TA algorithm on the group of the lists.

The TA algorithm searches for the top-k according to a monotone aggregate function g. The score $score(x)$ of each object x is evaluated by the aggregate function, i.e., $score(x) = g(A_1(x), ..., A_m(x))$, where each $A_i(x)$ is value of the attribute A_i of object x. The TA algorithm uses the *top-k* list, in which it keeps the best current objects with their scores and the *min-k* value as the score of the kth best object in the *top-k* list.

The TA algorithm repeatedly performs the sorted access to the lists $L_{n+1}, ..., L_m$ and it obtains pairs $\{x, A_i(x)\}$ from sorted lists $L_{n+1}, ..., L_m$ sequentially in round-robin style. For each newly obtained object x from one of the lists, its unknown attribute values are obtained by performing the random access to the lists $L_{n+1}, ..., L_m$ except list L_i, from which the pair $\{x, A_i(x)\}$ was obtained. Moreover, the values $A_1(x), ..., A_n(x)$ are known from the MDB-tree, i.e. from tree identifier $(k_1, ..., k_n)$ of the group of lists. Thus all the attribute values of newly obtained object x are known and $score(x)$ can be evaluated by the aggregate function, i.e.,

$$score(x) = g(k_1, ..., k_{n-1}, k_n, A_{n+1}(x), ..., A_m(x)).$$

Then object x is inspected whether or not it becomes a member of the *top-k* list by comparing its $score(x)$ and the *min-k* value.

The efficiency of the TA algorithm is based on the fact that it is capable of finding top-k objects much sooner than it reads the whole lists. The TA algorithm uses values $high_{n+1}, ..., high_m$ as all the last seen attribute values obtained by the sorted access from the sorted lists $L_{n+1}, ..., L_m$, respectively. The TA algorithm uses the *threshold* value to estimate the upper bound score of each object, which has not yet been seen in any of the lists. The *threshold* value is evaluated by the aggregate function, i.e.,

$$threshold = g(k_1, ..., k_{n-1}, k_n, high_{n+1}, ..., high_m).$$

If the *threshold* value is equal or lower than the *min-k* value, the TA algorithm can finish the search in the group of lists $L_{n+1}, ..., L_m$. The *top-k*

list contains the top-k objects found in the group of lists. If the *top-k* list contains less than k objects and all the objects from the lists were obtained, i.e. all the lists were read by the sorted access, the TA algorithm also can finish the search in the group of lists.

8.3.3 Integration of algorithms MD and TA

In Section 8.3.1 and Section 8.3.2, we described the use of the MD algorithm and the TA algorithm for the top-k search in the MDB-tree with lists. Both these algorithms use the *top-k* list, in which they keep the current top-k objects, and the *min-k* value.

In the moment when new object x is obtained by the TA algorithm in a group of lists, all the attribute values of object x are known and $score(x)$ can be evaluated. Therefore, it is possible to decide immediately whether or not object x is becoming a member of the *top-k* list. If $score(x)$ is higher than the *min-k* value, the *top-k* list is updated and the *min-k* value can be increased.

The main task of the top-k search in the MDB-tree with lists is to find the best k objects in the whole MDB-tree with lists according to the monotone aggregate function. Therefore, it is possible to use only one *top-k* list during the top-k search in the MDB-tree with lists. The use of the common *top-k* list integrates the MD algorithm searching for the top-k objects in the MDB-tree and the TA algorithm searching for the top-k in the groups of the lists. Thus this integration constitutes the base of the top-k search in the MDB-tree with lists.

Moreover, both algorithms can only use one common *min-k* value from the *top-k* list, which significantly affects their computation. For each B^+-tree P or the group of list P in the case of the MDB-tree with lists, the MD algorithm compares the best ranking of P and the *min-k* value, and thus it can skip some parts of the MDB-tree with lists. During the top-k search in the group of lists, the TA algorithm compares the *threshold* value and the *min-k* value, and thus it is able to stop before it reads the whole lists. These advantages can be used for the top-k in the MDB-tree with lists.

8.4 MXT algorithm

In this section, we introduce a new top-k algorithm, the MXT algorithm, which is able to find the top-k objects efficiently in the MDB-tree with lists according to a monotone aggregate function.

Arising from the results described in Section 8.3.3, the MXT algorithm will be based on the integration of the MD algorithm and the TA algorithm. This integration is possible because both the algorithms are able to search for the top-k objects according to a monotone aggregate function and we assume that the aggregate function is nondecreasing according to all variables. The MXT algorithm uses the *top-k* list, in which it keep the current top-k objects, and the *min-k* value as the score of the kth object in the *top-k* list.

The base of the MXT algorithm is constituted by the MD algorithm based on the recursive procedure (see Section 7.2.1). Since the values of the first n attributes $A_1, ..., A_n$ are stored as in the MDB-tree, the values are searched in the same way as in the MD algorithm. In each B^+-tree of the MDB-tree reached by the recursive procedure of the MD algorithm, its keys are obtained sequentially in descending order according to their values. The exit rule is checked for each obtained key. Each B^+-tree on the n-th level of the MDB-tree contains keys with pointers, which refer to the groups of $m-n$ Fagin's sorted lists, and these contain objects with values of attributes $A_{n+1}, ..., A_m$.

Each group of lists reached by the MD algorithm is searched by a new instance of the TA algorithm. This instance of the TA algorithm uses its own *threshold* value and the common *top-k* list with the MD algorithm. The instance of the TA algorithm search for the top-k objects in the lists until its *threshold* value is higher than the *min-k* value or the whole lists are read. When the instance of the TA algorithm finishes, computation of the MXT algorithm continues again as in the MD algorithm.

The efficiency of the MXT algorithm is based on the fact that it does not needed to obtain the best k objects from each group of lists. The MXT algorithm obtains only the objects with the score higher than the *min-k* value. Compared to the original TA algorithm, each instance of the TA algorithm can stop earlier in a group of lists, and thus it can perform a smaller number of sorted accesses and random accesses into the lists.

The pseudo-code of MXT algorithm

The pseudo-code depicted in Algorithm 12 describes the MXT algorithm. The main composition of the MXT algorithm is adopted from the MD algorithm. The FINDTOPK procedure is adopted from the Algorithm 10, but only the part from line 1 to line 12 is the same. In the MXT algorithm, the FINDTOPK procedure uses the ACTIVATEBPLUSTREE and the GETNEXTKEY procedure from the Algorithm 10. Each instance of the FINDTOPK procedure is run in one particular B^+-tree of the MDB-tree in which its keys are obtained sequentially in descending order according to their values.

Algorithm 12 MXT algorithm (the top-k search in MDB-tree with lists)

MXT(MDB-tree with lists *MDBT*, aggergate function g, number k)

1: *top-k* := $\emptyset$; *min-k* := 0;
2: findTopK(*MDBT*, ($\emptyset$), g, k);
3: **return** *top-k*;

procedure findTopK(MDB-tree *MDBT*, tree identifier $(k_1, ..., k_{i-1})$,
aggergate function g, number k)

1: T := B$^+$-tree with tree identifier $(k_1, ..., k_{i-1})$ on the ith level of *MDBT*;
2: activateBplusTree(T);
3: **while** $\exists$ the next key $k_i \in T$ **do**
4: p := getNextKey(T);
5: P := the node of *MDBT* referred to by $\rho(p)$;
6: $Best(P) := g(k_1, ..., k_{i-1}, p, 1, ..., 1)$;
7: **if** $Best(P) \leq$ *min-k* **then**
8: **return** ; // *according to Statement 7.2g*
9: **end if**
10: **if** P is B$^+$-tree **then**
11: findTopK(*MDBT*, $(k_1, ..., k_{i-1}, p)$, g, k);
12: **end if**
13: // *If T is on the nth level of MDBT (n is number of categorical attributes).*
14: **if** P is group of lists **then**
15: // *An instance of the TA algorithm is run on $m - n$ lists $L_{n+1}, ..., L_m$.*
16: **for** $n + 1 \leq e \leq m$ **do** $high_e := 1$;
17: **repeat**
18: **for** $n + 1 \leq e \leq m$ **do**
19: $\{x, A_e(x)\}$:= getNextPair(L_e); // *sorted access*
20: $high_e := A_e(x)$;
21: $threshold := g(k_1, ..., k_{i-1}, p, high_{n+1}, ..., high_m)$;
22: **for** $n + 1 \leq j \leq m$ **do**
23: **if** $j \neq e$ **then** $A_j(x)$:= getValue(L_j, x); // *random access*
24: **end for**
25: $score(x) := g(k_1, ..., k_{i-1}, p, A_{n+1}(x), ..., A_m(x))$;
26: **if** $score(x) >$ *min-k* **then**
27: **if** *top-k*.size() = k **then** *top-k*.removeKthObject();
28: *top-k*.insertInTheRightPlace(x);
29: *min-k* := *top-k*.getMin-k();
30: **end if**
31: **end for**
32: **until** $threshold \leq$ *min-k* **and** $\forall e$: $\nexists$ the next pair $\{x, A_e(x)\} \in L_e$
33: **end if**
34: **end while**
35: **return** ;

The MXT algorithm starts in the B$^+$-tree on the first level of the MDB-tree with tree identifier ($\emptyset$). If the pointer of a key refers to B$^+$-tree P on the next level of the MDB-tree, the FINDTOPK procedure is run in the B$^+$-tree. Otherwise, in case of the nth level of the MDB-tree, the pointer of the key refers to a group of lists P. If P only includes objects which do not have the chance to become members of the *top-k* list, the part is skipped according to the exit rule (see line 7 of the FINDTOPK procedure).

Each group of lists P reached by the MD algorithm is searched by an instance of the TA algorithm. The instance of the TA algorithm is run on $m - n$ lists $L_{n+1}, ..., L_m$. The lines from 16 to 12 of the FINDTOPK procedure are adopted from the original TA algorithm (see Algorithm 1). First, this instance of the TA algorithm uses its own *threshold* value and its own $high_{n+1}, ..., high_m$ values as the last seen attribute values obtained by the sorted access (see line 19 of the FINDTOPK procedure) in lists $L_{n+1}, ..., L_m$, respectively. For each newly obtained object x, its unknown values are obtained by the random access (see line 23 of the FINDTOPK procedure) and the ranking of the object, i.e., $score(x)$, is evaluated (see line 25 of the FINDTOPK procedure). If the ranking of the object is sufficient, the *top-k* list is updated by the procedure REMOVEKTHOBJECT and the INSERTINTHERIGHTPLACE procedure. The instance of the TA algorithm finishes, if its *threshold* value is equal or lower than the *min-k* value or all the lists were read.

Then the computation of the MXT algorithm continues by the next iteration of the while-loop of the FINDTOPK procedure as in the original MD algorithm.

Example 8.4 Figure 8.1 depicts an MDB-tree with lists, in which a set of objects with two nominal attributes and two ordinal attributes are stored. Two nominal attributes A_1 and A_2 are stored as the MDB-tree and two ordinal attributes A_3 and A_4 are stored as groups of sorted lists. Each group of lists contains two sorted lists. Figure 8.1 depicts sixteen objects of the set, which are stored in four groups of two sorted lists. In the MDB-tree with lists, the MXT algorithm searches the best three object according to summation $g(A_1(x), A_2(x), A_3(x), A_4(x)) = A_1(x) + A_2(x) + A_3(x) + A_4(x)$ as monotone aggregate function g.

The MXT algorithm starts in B$^+$-tree T_1 on the first level of the MDB-tree and it obtains key 0.9 with the highest value of attribute A_1, which refers to B$^+$-tree T_2, in which key 1.0 is obtained. This key refers to group of L_1 and L_2 lists, in which an instance of the TA algorithm is run. This instance obtains three pairs from both L_1 and L_2 lists until its $threshold = 2.6$ is not higher than $min\text{-}k = score(x_N) = 2.8$. The *top-k* list includes the current best three objects x_M, x_O and x_N.

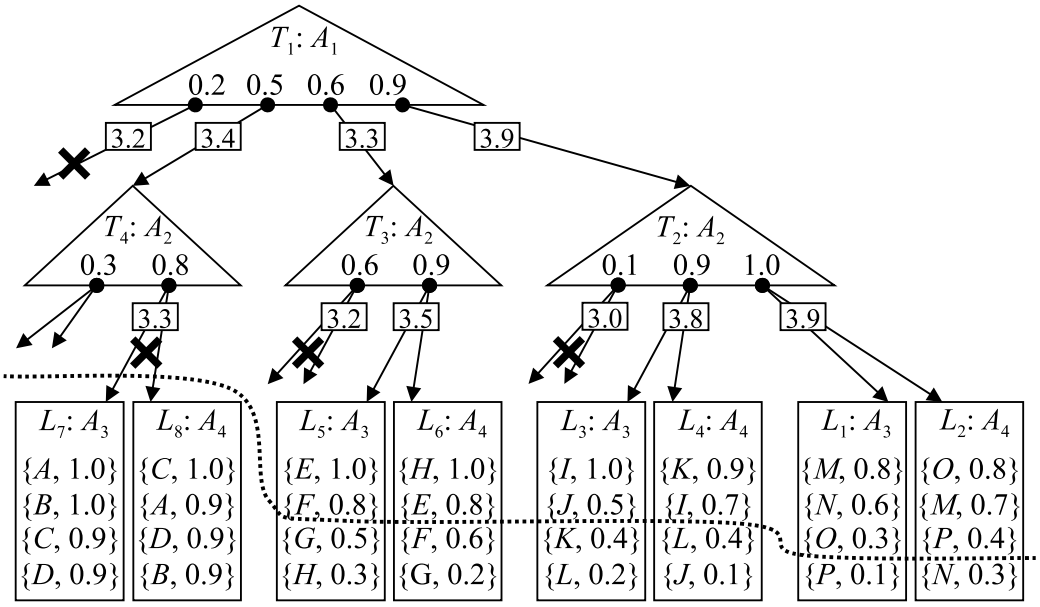


Figure 8.1: Example of searching for the top-3 object in the MDB-tree with lists with two nominal attributes and two ordinal attributes.

The MXT algorithm continues in B$^+$-tree T_2, where the next key 0.9 is obtained. This key refers to group of L_3 and L_4 lists, in which a new instance of the TA algorithm is run because the best ranking (depicted in the figure as a small rectangle with a number) of this group of lists is higher than the current *min-k* value. This instance obtains two pairs from both L_3 and L_4 lists until its *threshold* = 3.0 is not higher than *min-k* = $score(x_K)$ = 3.1. The *top-k* list includes the current best objects x_I, x_M and x_K.

The MXT algorithm continues in B$^+$-tree T_2, where the next key 0.1 is obtained. This key refers to a group of lists whose best ranking 3.0 is not higher than the current *min-k* value. Therefore, the MXT algorithm finishes in B$^+$-tree T_2 according to the exit rule.

The MXT algorithm continues in B$^+$-tree T_1, where the next key 0.6 is obtained. This key refers to B$^+$-tree T_3, in which key 0.9 is obtained. This key refers to group of L_5 and L_6 lists, in which a new instance of the TA algorithm is run because the value of its best ranking 3.5 is high enough. This instance obtains two pairs from both L_5 and L_6 lists until its *threshold* = 3.1 is not higher than *min-k* = $score(x_E)$ = 3.3. The *top-k* list includes the current best objects x_I, x_M and x_E.

The MXT algorithm continues in B$^+$-tree T_3, where the next key 0.6 is obtained. This key refers to a group of lists whose best ranking 3.2 is not higher than the current *min-k* value. Therefore, the MXT algorithm finishes

in B$^+$-tree T_3 according to the exit rule.

The MXT algorithm continues in B$^+$-tree T_1, where the next key 0.5 is obtained. This key which refers to B$^+$-tree T_4, in which key 0.8 is obtained. This key refers to group of L_7 and L_8 lists whose best ranking 3.3 is not higher than the current $min\text{-}k = score(x_E) = 3.3$ value. Therefore, the MXT algorithm finishes in B$^+$-tree T_3 according to the exit rule.

The MXT algorithm continues in B$^+$-tree T_1, where the next key 0.2 is obtained. This key which refers to the B$^+$-tree whose best ranking 3.3 is not higher than the current *min-k* value. Therefore, the MXT algorithm finishes the top-3 search in the MDB-tree with lists.

The efficiency of the MXT algorithm is illustrated by the dotted line in Figure 8.1. Under the dotted line, the part of the MDB-tree with lists which does not have to be accessed by the MXT algorithm is situated. The dotted line separates the sorted lists that the object-value pairs which were not read by particular instances of the TA algorithm are situated under. Instead of the best tree objects, only the best two objects x_I and x_K were added into the *top-k* list from group of L_3 and L_4 lists, and only the best object x_E was added into the *top-k* list from group of L_5 and L_6 lists.

8.4.1 Use of local preferences

The MXT algorithm allows a user to find the top-k objects in the MDB-tree with lists only according to a monotone aggregate function g. In this thesis, we focus on the top-k search with regard to the advanced model of user preferences which considers the use of the user's local preference for each attribute. More specifically, for a set of objects X with m attributes, user U expresses his/her local preferences as m fuzzy functions $f_1^U, ..., f_m^U$.

Since the MXT algorithm is based on integration of the MD algorithm and the TA algorithm, we focus on the variants of the MD algorithm and the TA algorithm, which enable us to apply the local preferences. In Section 7.3.1, we introduced the MDLP algorithm (see Algorithm 11) as an extension of the MD algorithm, which is able to find the top-k objects in the MDB-tree with the support of local user preferences. In Section 6.2, we introduced the TALP algorithm (see Algorithm 6) as an extension of the TA algorithm, which is able to use the local preferences. In contrast to the original TA algorithm, the TALP algorithm searches for the top-k objects in the B$^+$-trees instead of Fagin's sorted lists because it is possible to obtain the objects from the B$^+$-trees sequentially in descending order according to arbitrary user's local preferences (see Section 6.1).

Therefore, we designed the MDB-tree with groups of B$^+$-trees (see Section 5.5.1) as an extension of the MDB-tree with lists, where each Fagin's

sorted list is substituted by the B$^+$-tree. Let us propose a set of objects with m attributes, where n attributes $A_1, ..., A_n$ are categorical and the $m - n$ attributes $A_{n+1}, ..., A_m$ are numerical, is stored in the MDB-tree with groups of B$^+$-trees. Then the values of attributes $A_1, ..., A_n$ are stored like in the MDB-tree and the objects with values of attributes $A_{n+1}, ..., A_m$ are stored as groups of $m - n$ B$^+$-trees.

We developed the MXTLP algorithm as an extension of the MXT algorithm, which is able to search for the top-k objects efficiently according to monotone aggregate function g and by using the local preferences in the MDB-tree with groups of B$^+$-trees. The MXTLP algorithm is based on integration of the MDLP algorithm and the TALP algorithm because the MDB-tree with groups of B$^+$-trees is composed solely of B$^+$-trees. Therefore, it is possible to apply the local preferences directly during the computation of the top-k search in the MDB-tree with groups of B$^+$-trees.

In contrast to the MXT algorithm, the MXTLP algorithm obtains the keys in each B$^+$-tree of the MDB-tree with groups of B$^+$-trees sequentially in descending order according to values of corresponding fuzzy functions, i.e., in the MDB-tree according to fuzzy functions $f_1^U, ..., f_n^U$ depending on the level of the MDB-tree and in a group of B$^+$-trees $T_{n+1}, ..., T_m$ according to fuzzy functions $f_{n+1}^U, ..., f_m^U$, respectively. Thus, the MXTLP algorithm operates with the values of fuzzy functions instead of the values of attributes.

Analogously, like the MDLP algorithm, the MXTLP algorithm evaluates the best ranking of B$^+$-tree P on the ith level of the MDB-tree with tree identifier $(k_1, ..., k_{i-1})$ by using aggregate function g with a substitution as an expression

$$Best(P) = g(f_1^U(k_1), ..., f_{i-1}^U(k_{i-1}), 1, ..., 1)$$

and the best ranking of group of B$^+$-trees P with tree identifier $(k_1, ..., k_n)$ as an expression

$$Best(P) = g(f_1^U(k_1), ..., f_{i-1}^U(k_n), 1, ..., 1).$$

Analogously, like the TALP algorithm, the MXTLP algorithm during its computation in group of B$^+$-trees P with tree identifier $(k_1, ..., k_n)$ evaluates the ranking of each obtained object x by using aggregate function g with a substitution as an expression

$$score(x) = g(f_1^U(k_1), ..., f_n^U(k_n), f_{n+1}^U(x), ..., f_m^U(x)$$

and its own *threshold* value as an expression

$$threshold = g(f_1^U(k_1), ..., f_n^U(k_n), high_{n+1}, ..., high_m),$$

where values $high_{n+1}, ..., high_m$ are the last seen values of fuzzy functions $f_{n+1}^U, ..., f_m^U$ obtained by the sorted access from $m - n$ B$^+$-trees of P.

Algorithm 13 MXTLP algorithm (MXT algorithm with local preferences)

MXTLP(MDB-tree with lists *MDBT*, agg. g, local pref. $f_1^U, ..., f_m^U$, number k)

```
1: top-k := ∅; min-k := 0;
2: findTopKlp(MDBT, (∅), g, f_1^U, ..., f_m^U, k);
3: return top-k;
```

procedure findTopKlp(MDB-tree *MDBT*, tree identifier $(k_1, ..., k_{i-1})$, aggergation g, local preferences $f_1^U, ..., f_m^U$, number k)

```
1: T := B+-tree with tree identifier (k_1, ..., k_{i-1}) on the ith level of MDBT;
2: C := createCandidates(T, f_i^U);
3: while C ≠ ∅ do
4:    {p, f_i^U(p)} := getNextPair(T, f_i^U, C);
5:    P := the node of MDBT referred to by ρ(p);
6:    Best(P) := g(f_1^U(k_1), ..., f_{i-1}^U(k_{i-1}), f_i^U(p), 1, ..., 1);
7:    if Best(P) ≤ min-k then
8:       return ;                          // according to Statement 7.2g
9:    end if
10:   if P is B+-tree then
11:      findTopKlp(MDBT, (k_1, ..., k_{i-1}, p), g, f_1^U, ..., f_m^U, k);
12:   end if
13:   if P is group of B+-trees then
14:      // An instance of the TA algorithm is run on m−n B+-trees T_{n+1}, ..., T_m.
15:      for n + 1 ≤ e ≤ m do high_e := 1;
16:      for n + 1 ≤ e ≤ m do C_e := createCandidates(T_e, f_e^U);
17:      repeat
18:         for n + 1 ≤ e ≤ m do
19:            {x, f_e^U(x)} := getNextPair(T_e, f_e^U, C_e); // sorted access
20:            high_e := f_e^U(x);
21:            threshold := g(f_1^U(k_1), ..., f_{n-1}^U(k_{i-1}), f_n^U(p), high_{n+1}, ..., high_m);
22:            for n + 1 ≤ j ≤ m do
23:               if j ≠ e then f_j^U(x) := f_j^U(getValue(T_j, x)); // random access
24:            end for
25:            score(x) := g(f_1^U(k_1), ..., f_{n-1}^U(k_{i-1}), f_n^U(p), f_{n+1}^U(x), ..., f_m^U(x));
26:            if score(x) > min-k then
27:               if top-k.size() = k then top-k.removeKthObject();
28:               top-k.insertInTheRightPlace(x);
29:               min-k := top-k.getMin-k();
30:            end if
31:         end for
32:      until threshold ≤ min-k and ∀e : C_e = ∅
33:   end if
34: end while
35: return ;
```

The pseudo-code of MXTLP algorithm

The pseudo-code depicted in Algorithm 13 describes the MXTLP algorithm. The main composition of the MXTLP algorithm is adopted from the MXT algorithm (see Algorithm 12). Aditionally to the MXT algorithm, the MXTLP algorithm uses local user's preferences $f_1^U, ..., f_m^U$ as additional input parameters.

The part of the FINDTOPKLP procedure from line 1 to line 12 is the same as in the pseudo-code of the MDLP algorithm (see Algorithm 11). The part of the FINDTOPKLP procedure from line 14 to line 32 is adopted from the pseudo-code of the TALP algorithm (see Algorithm 6 in Section 6.2).

Since the MDB-tree with groups of B^+-trees is composed solely of B^+-trees, the MXTLP algorithm obtains the keys from each B^+-tree of the MDB-tree with groups of B^+-trees sequentially in descending order according to the values of corresponding fuzzy functions. Therefore, in contrast to the MXT algorithm, the MXTLP algorithm uses the CREATECANDIDATES procedure and the GETNEXTPAIR procedure depicted in the pseudo-code of the MDLP algorithm (see Algorithm 11).

Each B^+-tree which is reached by the MXTLP algorithm in the MDB-tree or in a group of B^+-trees has to be initialized by the CREATECANDIDATES procedure, which was originally adopted from the BTREELIST algorithm (see Algorithm 4 in Section 6.1.3). Afterwards, the MXTLP algorithm obtains the keys from the B^+-tree sequentially by using the GETNEXTPAIR procedure, which was originally adopted from the BTREELIST algorithm. Therefore, the MXTLP algorithm tests whether or not the set of candidates (see Section 6.1.3) for a B^+-tree is empty instead of testing the existence of the next key in the B^+-tree or the existence of the next object-value pair in a sorted list as in the case of the MXT algorithm.

8.5 Experiments

We have tested the MXT algorithm and compared it with the MD algorithm and the TA algorithm. For this purpose, we have implemented variants of the algorithms which use the local preferences. Namely, we implemented the MXTLP algorithm using the MDB-tree with groups of B^+-trees, the MDLP algorithm using the MDB-tree (see Algorithm 11) and the TALP algorithm by using the B^+-trees instead of the sorted lists (see Algorithm 6).

The implementation was developed in Java. Analogously as in Section 7.4, the main research point was to estimate the number of accesses into data structures during the execution of algorithms. During the basic experiments,

we used simple user preferences. We used fuzzy functions whose courses were based on an increasing linear function with respect to the attribute values as user's local preferences. We used the arithmetic average as the aggregate function.

8.5.1 Distribution of attribute values

First, we tested two sets of 100 000 objects with 5 attributes with uniform and normal distribution of attribute values. We used the TA algorithm, the MD algorithm and three variants of the MXT algorithm, namely MXT_3, MXT_2 and MXT_1. Each variant proposes a different number of categorical attributes in the sets of objects.

The MXT_3 algorithm uses the first three attributes of the set of objects as categorical attributes, which are stored as an MDB-tree with three levels, and the other two attributes are stored as groups of two Fagin's sorted lists. The MXT_2 algorithm uses two attributes as a categorical attributes and the MXT_1 algorithm uses one attribute as categorical attribute. Figure 8.2 and Figure 8.3 show results of this test.

The best results were achieved by the MXT_3 algorithm and the MD algorithm for the set of objects with the uniform distribution of the attributes values. These results confirmed that the algorithms which we developed are capable of exceeding the Fagin's algorithms for suitable multidimensional data sets. The test for normal distribution of attribute values showed that the MXT algorithm can also achieve the worst results in some cases. It means that for the MXT algorithm it is very important how many attributes of a particular set of objects are perceived as categorical attributes.

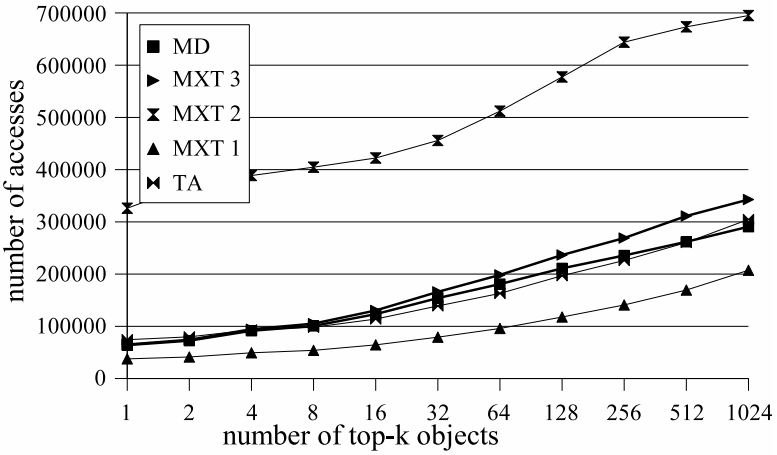


Figure 8.2: Normal distribution of attribute values.

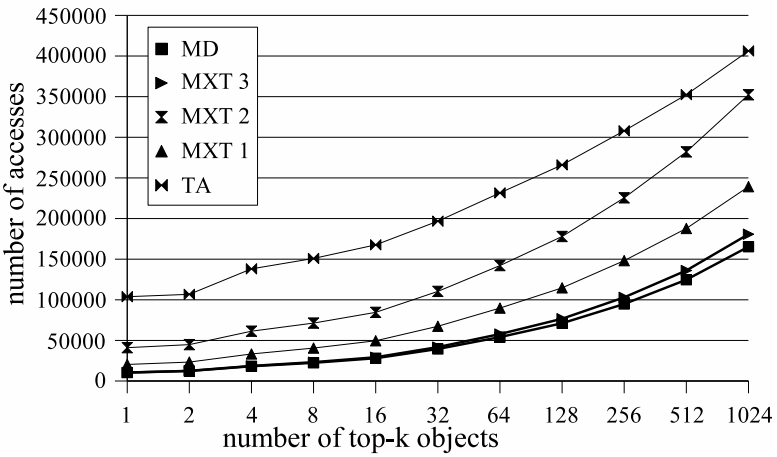


Figure 8.3: Uniform distribution of attributes values.

8.5.2 Real data set

We tested the algorithms by using several real data sets. We obtained the most representative result for a set of 8 822 flats for rent in Prague (see Example 5.5 in Section 5.5). These flats have four relevant attributes. Two attributes are categorical with a small-sized domain and two attributes are numerical attributes with a big-sized domain. We used the TA algorithm, the MD algorithm and the most suitable variant of the MXT algorithm for two categorical attributes. Figure 8.4 shows the results of this test.

The best result was achieved by the MXT algorithm. This test showed that the MXT algorithm provides the most efficient execution of the top-k search in this case.

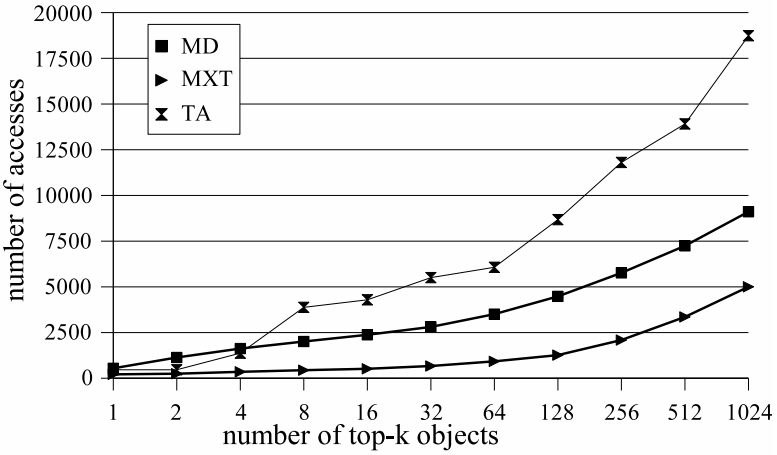


Figure 8.4: Finding the best flats in Prague.

In practice, the majority of the sets of objects searched by users have more categorical attributes and more numerical attributes. In this case, it is the most suitable solution to use the MXT algorithm and the MDB-tree with lists (MDB-tree with groups of B^+-trees) as the index.

8.5.3 Various user preferences

In general, it is difficult to test the efficiency of top-k algorithms in dependence on local user preferences. For various settings of local user preferences and for various distribution of attribute values, top-k algorithms achieve different results. The use of local user preferences changes the distribution of attribute values, and the efficiency of proposed algorithms depends on the distribution.

On the other hand, the use of local preferences affects the number of performed accesses only slightly (see Section 7.4.3) because we use the support of local user preferences based on the B^+-tree and the number of the additional accesses is many time smaller than the number of subsequent accesses performed by the algorithms.

Concerning other associated results we discovered that the aggregate function used as global user preference has a significant influence on the execution of proposed top-k algorithms. Therefore we focus on the experimental evaluation of the MXT algorithm, the MD algorithm and the TA algorithm in dependence on the aggregate function.

For this purpose we used the weighted average (see Section 3.4.1) as the most representative case of the nondecreasing aggregate function according to all variables (see Section 3.4.2). We used a set of 100 000 objects with two nominal attributes and three ordinal attributes. The distribution of attribute values was uniform. Figure 8.5 shows the results of this test, where the weights of all the attributes were the same.

The MD algorithm and the MXT algorithm achieved almost the same and the best results. In this test, the worst results were achieved by the TA algorithm. To find the top-k objects, the MXT algorithm and the MD algorithm needed to perform over 400 times fewer accesses than the TA algorithm. This test showed that using of the MXT algorithm and the MD algorithm is the most efficient solution for a set of objects with more categorical attributes.

Figure 8.6 shows the results of a test, where the weights of the first two categorical attributes were equal to 0. In this case, the result of the top-k search is independent of the values of nominal attributes. The computations of the MXT algorithm and the MD algorithm were affected because they could not utilize their advantage based on the reduction of dimensionality (see Section 5.2).

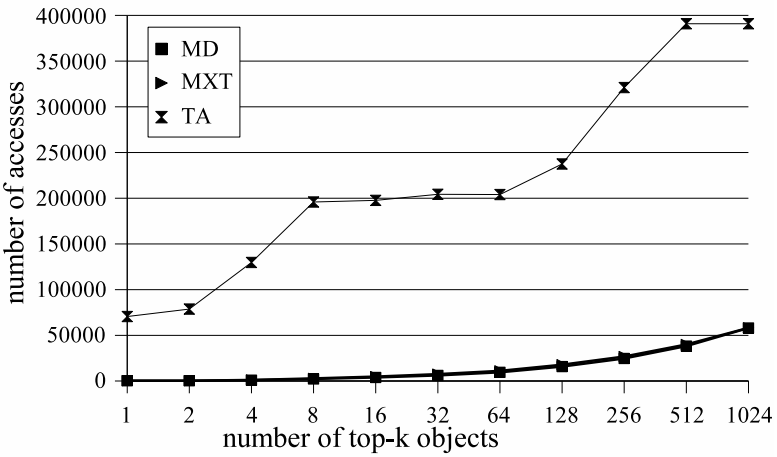


Figure 8.5: Weights of all the attributes were the same.

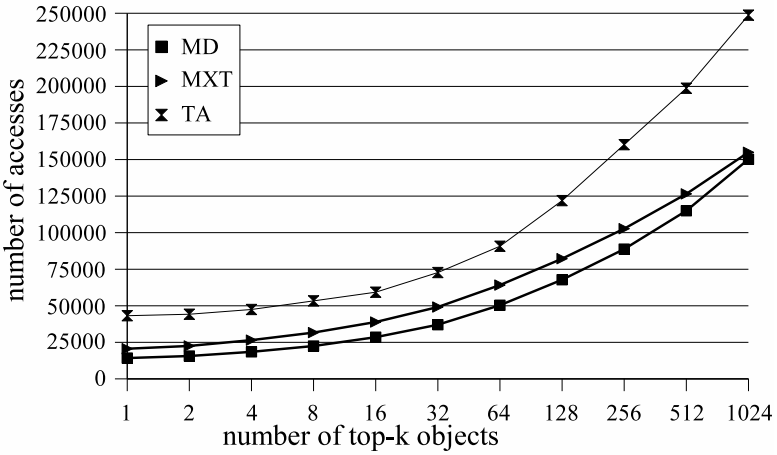


Figure 8.6: Weights of categorical attributes were equal to 0.

For the set of objects used, the MDB-tree with two levels constitutes the top part of corresponding data structures and the depth-first search of the MDB-tree is changed into the breadth-first search (see Section 7.2.1). The TA algorithm achieved better results than in the previous test depicted in Figure 8.5 because in fact, the TA algorithm was searching for the top-k objects only according to three attributes and it is a less complex problem than according to five attributes.

Figure 8.7 shows the results of a test, in which the MXT algorithm achieved the best results in number of accesses. In this test, the weight of the first numerical attribute was equal to 0, i.e., the top-k search was independent on the values of the third attribute. The results of the MD algorithm

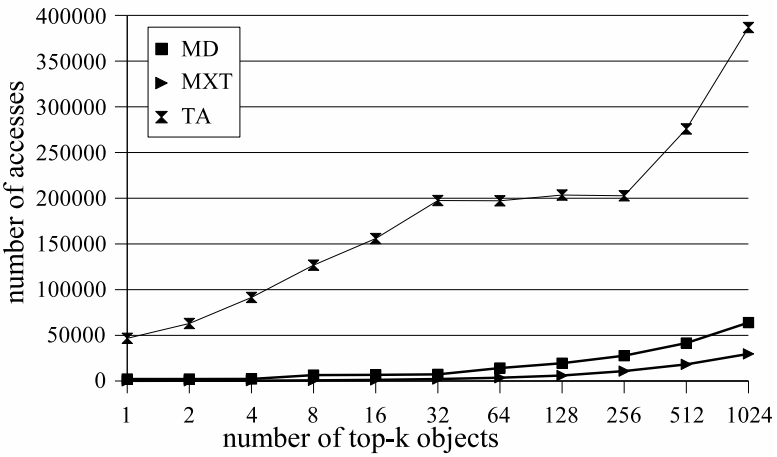


Figure 8.7: Weight of the first numerical attribute was 0.

and TA algorithm were very similar to those depicted in Figure 8.5.

The MD algorithm achieved worse results than the MXT algorithm because the weight of the attribute on the third level of the corresponding MDB-tree was 0. Therefore, the MD algorithm was exhaustively searching the keys in B^+-trees on the third level of the MDB-tree without progressively decreasing of the best ranking of the B^+-trees on the next level of the MDB-tree. Thus, all the parts of the MDB-tree referred to by keys of relevant B^+-trees on the third level of the MDB-tree were accessed by the recursive procedure of the MD algorithm. The exit rule could not be used on the third level of the MDB-tree.

In this case, the MXT algorithm utilized the advantages of the MD algorithm and the TA algorithm, of which it is composed. The instance of MD algorithm in the MXT algorithm searched for the best groups in the MDB-tree with two levels and each instance of the TA algorithm in the MXT algorithm searched for the top-k objects in a group of list (in a group of B^+-trees) only according to two attributes and it is a less complex problem than for three attributes. This case for three attributes is depicted as the result of the MXT algorithm in Figure 8.5.

8.6 Summary

These and other tests which we performed showed that the MXT algorithm achieves results comparable with the results of the other proposed top-k algorithms. In some cases, the MXT algorithm provides the most efficient execution of top-k search. The MXT algorithm is efficient for a set of objects

which has several nominal attributes with small-sized domains and several ordinal attributes with big-sized domains, especially if the user prefers the first few attributes to the other ones.

According to the types of object attributes and the distribution of attribute values, it is possible to choose how to index a set of objects. Because we use the local preferences, we index the set of objects in the MDB-tree, in the B^+-trees or in the MDB-tree with groups of B^+-trees. Each of the implemented top-k algorithms searches for the top-k objects in one of these data structures.

According to the properties of a set of objects, we can guess how efficient the computation of a particular algorithm will be. Therefore, if we know the properties of a set of objects, we can choose a data structure, in which the set of objects should be indexed and thus the most efficient top-k algorithm which will be used. The process of choosing the best data structure can be automated in real applications according to analyzing attribute domains, i.e., distribution of attribute values, sizes of actual attribute domains or information about the attribute types.

In general, the complexity of the MD algorithm and the MXT algorithm is a difficult question because it depends on many factors. Therefore, it is difficult to describe the complexity in a presentable way.

A crucial factor affecting the complexity of the algorithms is how the corresponding data structures are fulfilled. Let us suppose an example of the MD algorithm using the MDB-tree. Let us suppose a set of objects with 5 attributes, where the domain of each attribute has 10 different values. This set of objects can include 100 000 objects with at least one different attribute value. We can suppose a uniform distribution of objects with regard to the space of attribute domains.

If the set contains a few (from 10 to 1 000) objects which are stored in the MDB-tree, the B^+-tree on the first level of the MDB-tree is probably full, i.e., it includes all 10 values of the first attribute. However, the B^+-trees on the other levels are degenerated, and thus the whole MDB-tree is degenerated (see Section 5.4.1). In this case, the MD algorithm is not efficient (see Section 7.6) in comparison to the other top-k algorithms.

On the other hand, if the set contains about 100 000 or more (a few million) objects which are stored in the MDB-tree, then the MDB-tree is very regular. In this case, each B^+-tree on every level includes probably almost all the possible values of attributes. In this case, the MD-algorithm is very efficient because it finds the top-k objects in a very small part of the MDB-tree. In the best case, all the top-k objects are obtained from one object array.

Chapter 9

Conclusion

In this thesis, we have focused on the top-k search according to user preferences by using the B$^+$-tree, the MDB-tree and novel tree-based data structures based on the multidimensional B-tree. These data structures provide a dynamic environment for the top-k search and enable us to search for the top-k objects according to a non-monotone ranking function.

First, we focused on the top-k search with support of non-monotone ranking functions by the model of sorted list based on the B$^+$-tree. We have enabled Fagin's algorithms to search for the top-k objects according to a non-monotone ranking function. Moreover, we have applied the top-k search with the support of non-monotone ranking functions in the Web environment with data on different remote servers.

Then we focused on the top-k search in various multidimensional tree-based data structures based on reduction of dimensionality by using B$^+$-trees. In this thesis we designed several novel data structures, namely, the MDB-tree composed of B$^+$-trees, the MDB-tree with lists, the MDB-tree with groups of B$^+$-trees and the multiple-ordered MDB-tree. Concurrently, we have developed several novel top-k algorithms, namely, MD, MDLP, MXT, MXTLP, TALP and NRALP. These algorithms are efficient top-k algorithms because they are able to find the top-k objects without the need to obtain the majority of the objects. In some cases, the MXT algorithm and the MD algorithm needed to perform over 400 times fewer accesses than the TA algorithm.

All the algorithms proposed were experimentally evaluated by using various artificial and real data sets. Our results have shown that our top-k algorithms achieve results comparable with the results of the other top-k algorithms proposed in related works. Moreover, in the majority of cases, our top-k algorithms have the best results, especially for a small number of the top-k objects, and it is very convenient for the purposes of the top-k search.

In addition, based on the experimental results, we can guess how efficient the computation of a particular algorithm will be according to the properties of a set of objects as sizes and types of its actual attribute domains. Therefore, if we know the properties of a set of objects, we can choose the most suitable of the proposed data structures, in which the set of objects should be indexed and thus the most efficient of the proposed top-k algorithms which will be used.

At the same time, we are developing the TREETOPK software system [On11a, On11b], which includes the implementation of our developed top-k algorithms and new multidimensional tree-based data structures. The system also includes several user interfaces, in which the user can choose his/her local and global preferences and thus to set a non-monotone ranking function according to which the system searches for the top-k object for the user. This system was used for the experimental evaluation of the proposed top-k algorithms.

9.1 Contribution

This thesis have brought various contributions into the wide area of top-k search. These contributions were published in [OP08, OP09, Ond09, OP10, HOP11, On11a, On11b]. Our publications were also cited by several different authors in [WMD10, SG12, Cao12, SWC13]. In this section, we discuss the main contributions as follows.

We introduced the model of a sorted list based on the B^+-tree in detail, which enables the Fagin's algorithm to search for the top-k objects according to the non-monotone ranking function, which is composed of the arbitrary local user preferences and monotone aggregate function.

Initially, we adopted concept of similarity search from [FKS03] and the approach from [EPV07, GVP08]. Subsequently, in this thesis and in [OP08, OP09, OP10, HOP11], we redefined the theoretical background of the model. Thus, we were able to apply Fagin's NRA algorithm using the model in Web environment [HOP11], where the values of each attribute are stored on different remote servers on the Internet and, moreover, each server can be stateless, i.e. independent of any previous request.

We applied the multidimensional B-tree in the top-k search. We developed the MD algorithm as a new efficient top-k algorithm, which is able to find the top-k objects in the MDB-tree according to a monotone ranking function [OP08].

Moreover, we developed the MDLP algorithm as a variant of the MD algorithm which is able to find the top-k objects in the MDB-tree composed

of B^+-trees according to a non-monotone ranking function [OP08, OP10]. In addition, we propose the multiple-ordered MDB-tree as a new multidimensional index structure, which guarantees the most efficient execution of the MD algorithm and the MDLP algorithm.

We developed the MDB-tree with lists as a new multidimensional tree-based data structure and the MXT algorithm as a new efficient top-k algorithm, which is able to find the top-k objects in the MDB-tree with lists according to a monotone ranking function [OP09, OP10].

Moreover, we developed the MDB-tree with groups of B^+-trees as a new multidimensional tree-based data structure. In addition, we developed the MXTLP algorithm as a variant of the MD algorithm which is able to find the top-k objects in the MDB-tree with groups of B^+-trees according to a non-monotone ranking function.

9.2 Future work

There are many issues that can be further investigated. These issues can provide suggestions, motivations and directions of our future research. Some of them were already mentioned in Section 6.5, Section 7.6 and Section 8.6 in the summaries of the corresponding chapters.

Since the MXT algorithm is based on the integration of the MD algorithm and the TA algorithm, we can perceive the MD algorithm and the TA algorithm as extreme cases of the MXT algorithm. According to the properties of a set of objects we can choose the most suitable of the proposed tree-based data structures, in which it will be stored. The process of choosing the best data structure can be automated according to an analysis of attribute domains, e.g. the types and sizes of attribute domains.

Because of the MXT algorithm construction, it could be interesting that some instances of the TA algorithm would be computed concurrently. In this way, we can involve parallel computing techniques in the MXT algorithm and thus improve its efficiency. Moreover, by using the parallel computing, we presuppose an extension of the MXT algorithm that the instances of the NRA algorithm or the CA algorithm can be used instead of the instances of the TA algorithm. Analogously, the MD algorithm would be improved by parallel computing that several filial sets of the MDB-tree (several sub-trees) can be inspected concurrently whether or not they contain some objects which can become members of the top-k result.

In that case, it would be useful to apply the improvements developed for Fagin's top-k algorithms as the access scheduling [BMT06, GBK00, GV08] or the probabilistic approach [FLN01, TSW04] allowing the approximate result

(see Section 2.2.2). Thus, the efficiency of the MD algorithm and the MXT algorithm can be improved.

Nowadays, in the age of gigabytes, we assume that the idea of preprocessed data in the top-k search [HKP01, DGK06] is becoming an interesting direction in the context of the so-called Big Data problem. In this context, we propose the use of the multiple-ordered MDB-tree (see Section 7.5). As it is common in information retrieval, we can use a number of preprocessed multiple-ordered MDB-trees to improve the subsequent search for the top-k objects by our top-k algorithms.

For example, for a huge number of attributes (dimensions) of the objects, we can use the double-ordered MDB-tree for each attribute pair or the triple-ordered MDB-tree for each attribute triplet and so on. Then, according to user preferences, we can choose the most suitable MDB-tree included in the corresponding multiple-ordered MDB-tree for the execution of the MD algorithm. Moreover, some of such multiple-ordered MDB-trees including few common attributes can be perceived as one MDB-tree with lists and the MD algorithm can be used.

Another direction in our future research can be focusing on the multi-keywords search. For example, the authors in [SWC13] used the MD algorithm and the MDB-tree, which improve the efficiency of their multi-keyword search engine. To support multi-keyword search and search result ranking, the authors propose to build the MDB-tree based on term frequency and the vector space model with cosine similarity measure to achieve a higher search result accuracy.

In our research, we used a model of user preferences based on local and global user preferences. It would be interesting to focus on different models of user preferences. For example, if the values of two attributes are mutaly dependent, the user can set one local preference for both attributes. In this case, we should evolve some modifications of our top-k algorithms or we should develop some new top-k algorithms, which would be based on completely different models of preferences. Moreover, two attributes can be incomparable, and then we do not know which one is better for the user because neither of them dominates the other one in both attributes as in the skyline.

We can also focus on relevance feedback in the case of repeated top-k queries from the user. After running first top-k query, the user receives the top-k objects. Then the user can tune his/her preferences and execute the next top-k query. In that case, it is possible to help the user with tuning his/her preferences and reuse some data from the previous computations of the top-k algorithm.

Bibliography

[ARS03] Amato, G., Rabitti, F., Savino, P., Zezula, P.: Region proximity in metric spaces and its use for approximate similarity search. In: Proc. of ACM Transactions on Information Systems, Vol. 21, No. 2, pp. 192–227, 2003.

[BBK01] Böhm, C., Berchtold, S., Keim, D. A.: Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. In: Proc. of ACM Computing Surveys, Vol. 33, No. 3, pp. 322–373, 2001.

[BCG02] Bruno, N., Chaudhuri, S., Gravano, L.: Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation. In: Proc. of ACM Transactions on Database Systems, Vol. 27, No. 2, pp. 153-187, 2002.

[Ben75] Bentley, J. L.: Multidimensional Binary Search Trees Used for Associative Searching. Communications of the ACM, Vol. 18, No. 19, pp. 509-517, 1975.

[BGM02] Bruno, N., L. Gravano, L., Marian, A.: Evaluating top-k queries over web-accessible databases. In: Proc. of the 18th International Conference on Data Engineering (ICDE'02), pp. 369-380, 2002.

[BKS01] Borzsonyi, S., Kossmann, D., and Stocker, K.: The Skyline Operator. In: Proc. of the 17th International Conference on Data Engineering (ICDE'01), pp. 421–430, 2001.

[BL85] Buckley, C., Lewit, A. F.: Optimization of inverted vector searches. In: Proc. of the 8th International Conference on Research and Development in Information Retrieval (SIGIR'85), Washington: ACM, pp. 97-110, 1985.

[BM72] Bayer, R., McCreight, E.: Organization and Maintenance of Large Ordered Indices. Acta Informatica, Vol. 1, Fasc. 3, pp. 173-189, 1972.

[BMT06] Bast, H., Majumdar, D., Theobald, M., Schenkel, R., Weikum, G.: IO-Top-k: Index-optimized top-k query processing. In: Proc. of the 32th International Conference on Very Large Data Bases (VLDB'06), pp. 475–486, 2006.

[BP98] Brin, S., Page, L.: The anatomy of a large-scale hypertextual Web search engine. Computer Networks and ISDN Systems, Vol. 30, No. 4, pp. 107–117, 1998.

[Cao12] Cao, N.: Secure and Reliable Data Outsourcing in Cloud Computing. Dissertation Thesis, Faculty of the Worcester Polytechnic Institute, Worcester, Massachusetts, United States, July 2012.

[CDH04] Chaudhuri, S., Das, G., Hristidis, V., Weikum, G.: Probabilistic ranking of database query results. In: Proc. of the 30th International Conference on Very Large Data Bases (VLDB'04), Vol. 30, pp. 888-899, 2004.

[CDN01] Chaudhuri, S., Das, G., Narasayya, V.: A robust, optimization-based approach for approximate answering of aggregate queries. In: Proc. of the 2001 ACM International Conference on Management of Data (SIGMOD'01), ACM, New York, NY, USA, pp. 295-306, 2001.

[CG99] Chaudhuri, S., Gravano, L.: Evaluating top-k selection queries. In: Proc. of the 25th International Conference on Very Large Data Bases (VLDB'99), Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 397-410, 1999.

[CGM04] Chaudhuri, S., Gravano, L., Marian, M.: Optimizing Top-k Selection Queries over Multimedia Repositories. IEEE Trans. On Knowledge and Data Engineering (TKDE'04), Vol. 16, No. 8, pp. 992-1009, 2004.

[Com79] Comer, D.: The Ubiquitous B-Tree. ACM Computing Surveys, Vol. 2, No. 11, pp. 121-138, 1979.

[CP00] Ciaccia P., Patella, M.: PAC nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In: Proc. of the 16th International Conference on Data Engineering (ICDE'00), pp. 244–255, 2000.

[CPZ97] Ciaccia, P., Patella, M., Zezula, P.: M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In: Proc. of the 23th International Conference on Very Large Data Bases (VLDB'97), pp. 426-435, 1997.

[CH02] Chang, K. C. C., Hwang, S. W.: Minimal probing: Supporting expensive predicates for top-k queries. In: Proc. of the 2002 ACM International Conference on Management of Data (SIGMOD'02), ACM, New York, NY, USA, pp. 346-357, 2002.

[DGK06] Das, G., Gunopulos, D., Koudas, N., Tsirogiannis, D.: Answering top-k queries using views. In: Proc. of the 32th International Conference on Very Large Data Bases (VLDB'06), pp. 451–462, 2006.

[DPK08] Deshpande, P. M., P, D., and Kummamuru, K.: Efficient online top-K retrieval with arbitrary similarity measures. In: Proc. of the 11th International Conference on Extending Database Technology (EDBT'08), ACM, New York, NY, pp. 319-330, 2008

[EPV07] Eckhardt, A., Pokorný, J., Vojtáš, P.: A system recommending top-k objects for multiple users preference. In Proc. of 2007 IEEE International Conference on Fuzzy Systems, July 24-26, 2007, London, England, pp. 1101-1106, 2007.

[Fag96] Fagin, R.: Combining fuzzy information from multiple systems (extended abstract). In Proc. of Symposium on Principles of Database Systems (PODS '96), Montreal, Quebec, Canada, June 04 - 06, 1996, ACM, New York, NY, pp. 216-226, 1996.

[Fag99] Fagin. R.: Combining Fuzzy information from multiple systems. Journal of Computer and System Sciences, Academic Press, Inc. Orlando, FL, USA, Vol. 58, No. 1, pp. 83-99, 1999.

[Fag02] Fagin. R.: Combining fuzzy information: an overview. In: Proc. of the 2002 ACM International Conference on Management of Data (SIGMOD'02), ACM, New York, NY, USA, Vol. 31, No. 2, pp. 109–118, 2002.

[FKS03] Fagin, R., Kumar, R., Sivakumar, D.: Efficient similarity search and classification via rank aggregation. In: Proc. of the 2003 ACM International Conference on Management of Data (SIGMOD'03), ACM, New York, NY, USA, pp. 301-312, 2003.

[FLN01] Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: Proc. of 2001 ACM Symposium on Principles of Database Systems (PODS'01), pp. 102-113, 2001.

[FLN03] Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. Journal of Computer and System Sciences, Vol. 66, No. 4, pp. 614-656, 2003.

[GBK00] Güntzer, U., Balke, W. -T., Kießling, W.: Optimizing multi-feature queries for image databases. In: Proc. of the 26th International Conference on Very Large Data Bases (VLDB'00), pp. 419–428, 2000.

[GLV05] Gurský, P., Lencses, R., Vojtáš, P.: Algorithms for user dependent integration of ranked distributed information. In: Proc. of TED Conference on e-Government (TCGOV'05), LNCS (LNAI), Vol. 3416, pp. 123-130, 2005.

[GM80] Gopalakrishna, V., Veni Madhavan, C. E.: Performance evaluation of attribute-based tree organization. In: Proc. of ACM Transactions on Database Systems, Vol. 5, No. 1, pp. 69-87, 1980.

[Gut84] Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proc. of the 1984 ACM International Conference on Management of data (SIGMOD'84), ACM, New York, NY, USA, pp. 47-57, 1984.

[GV08] Gurský, P., Vojtáš, P.: Speeding Up the NRA Algorithm. In: Proc. of the 6th International Conference on Scalable Uncertainty Management (SUM'08), LNCS, Springer, Vol. 5291, pp. 243-255, 2008.

[GVP08] Gurský, P., Vaneková, V., Pribolová, J.: Fuzzy User Preference Model for Top-k Search. In: Proc. of Proceedings of IEEE World Congress on Computational Intelligence (WCCI'08), pp. 1606-1612, 2008.

[HKP01] Hristidis, V., Koudas, N., Papakonstantinou, Y.: PREFER: a system for the efficient execution of multi-parametric ranked queries. In: Proc. of the 2001 ACM International Conference on Management of Data (SIGMOD'01), ACM, New York, NY, USA, pp. 259-270, 2001

[HOP11] Horničák, E., Ondreička, M., Pokorný, J., Vojtáš, P.: Multi-user Searching of Top-k Objects with Data on Remote Servers. In: Proc. II of the 15th International Conference on Advances in Databases and Information Systems (ADBIS'11), CEUR Workshop Proceedings, Vol. 789, pp. 200-211, 2011.

[HS99] Hjaltason, G. R., Samet, H.: Distance browsing in spatial databases. In: Proc. of ACM Transactions on Database Systems, Vol. 24, No. 2, pp. 265–318, 1999.

[HS03] Hjaltason, G. R., Samet, H.: Index-driven similarity search in metric spaces. In: Proc. of ACM Transactions on Database Systems, Vol. 28, No. 4, pp. 517-580, 2003.

[IAE02] Ilyas, I. F., Aref, W. G., Elmagarmid, A. K.: Joining ranked inputs in practice. In: Proc. of the 28th International Conference on Very Large Data Bases (VLDB'02), pp. 950-961, 2002.

[IAE04] Ilyas, I. F., Aref, W. G., Elmagarmid, A. K.: Supporting top-k join queries in relational databases. In: Proc. of the 30th International Conference on Very Large Data Bases (VLDB'04), pp. 207-221, 2004.

[IBS08] Ilyas, I. F., Beskales, G., Soliman, M. A.: A survey of top-k query processing techniques in relational database systems. ACM Computing Surveys (CSUR), Vol. 40, No. 4, October 2008, pp. 1-58, 2008.

[LCI05] Li, C., Chang, K. C., Ilyas, I. F., Song, S.: Ranksql: Query algebra and optimization for relational top-k queries. In: Proc. of the 2005 ACM International Conference on Management of Data (SIGMOD'05), ACM, New York, NY, USA, pp. 131-142, 2001

[LCI06] Li, C., Chang, K. C., Ilyas, I. F.: Supporting ad-hoc ranking aggregates. In: Proc. of the 2006 ACM International Conference on Management of Data (SIGMOD'06), ACM, New York, NY, USA, pp. 61-72, 2006.

[MAK05] Marian, A., Amer-Yahia, S., Koudas, N., Srivastava, D.: Adaptive Processing of Top-k Queries in XML. In: Proc. of the 21th International Conference on Data Engineering (ICDE'05), pp. 162-173, 2005.

[MTW05] Michel, S., Triantafillou, P., Weikum, G.: KLEE: a framework for distributed top-k query algorithms. In: Proc. of the 31th International Conference on Very Large Data Bases (VLDB'05), VLDB Endowment, pp. 637–648, 2005.

[NCS01] Natsev, A., Chang, Y. -C., Smith, J. R., Li, C. -S., Vitter, J. S.: Supporting incremental join queries on ranked inputs. In: Proc. of the 27th International Conference on Very Large Data Bases (VLDB'01), pp. 637–648, 2005.

[NHS84] Nievergelt, J., Hinterberger, H., Sevcik, K. C.: The Grid File: An Adaptable, Symmetric Multikey File Structure. In: Proc. of ACM Transactions on Database Systems, Vol. 9, No. 1, pp. 38-71, 1984.

[NR99] Nepal, S., Ramakrishna, M. V.: Query Processing Issues in Image (Multimedia) Databases. In: Proc. of the 15th International Conference on Data Engineering (ICDE'99), IEEE Computer Society, Washington, DC, USA, pp. 22–29, 1999.

[On11a] Ondreička, M.: Efficient Top-k Searching According to User Preferences Based on Fuzzy Functions with Usage of Tree-Oriented Data Structures. In: Proc. of the 35th International Conference on Very Large Data Bases (VLDB'11), PhD Workshop, Seattle, Washington, USA, Sep 1, VLDB Endowment, pp. 13-18, 2011.

[On11b] Ondreička, M.: Top-k Searching for More Users in Multidimensional B-tree with Lists. In: Proc. II of the 15th International Conference on Advances in Databases and Information Systems (ADBIS'11), Doctoral Consortium, CEUR Workshop Proceedings, Vol. 789, pp. 242-247, 2011.

[Ond09] Ondreička, M.: Searching of Top-K Objects: A Survey. In: Proc. the 18th Annual Conference of Doctoral Students (WDS'09), Proceedings of Contributed Papers: Part I - Mathematics and Computer Sciences, Prague, Matfyzpress, pp. 51–55, 2009.

[OP08] Ondreička, M., Pokorný, J.: Extending Fagin's algorithm for more users based on multidimensional B-tree. In: Proc. II of the 12th International Conference on Advances in Databases and Information Systems (ADBIS'08), P. Atzeni, A. Caplinskas, and H. Jaakkola (Eds.), LNCS 5207, Springer-Verlag Berlin Heidelberg, pp. 199-214, 2008.

[OP09] Ondreička, M., Pokorný, J.: Combination of TA- and MD-algorithm for efficient solving of top-K problem according to user's preferences. In: Proc. of the 9th Annual International Workshop on Databases, Texts, Specifications, and Objects (DATESO'09), Eds. Richta, K., Pokorny, J., Snasel, V., CEUR Workshop Proceedings, Vol. 471, pp. 142-153, 2009.

[OP10] Ondreička, M., Pokorný, J.: Efficient Top-K Problem Solvings for More Users in Tree-Oriented Data Structures. In: Proc. of the 5th International Conference on Signal Image Technology and Internet Based Systems (SITIS'09), Marrakech, Morocco, IEEE Computer Society Press, pp. 345-354, 2010.

[OS81] Ouksel, A. M., Scheuermann, P.: Multidimensional B-Trees: Analysis of Dynamic Behavior. In: Proc. of BIT 1981, Vol. 21, No. 4, pp. 401-418, 1981.

[PF95] Pfeifer, U., Fuhr, N.: Efficient processing of vague queries using a data stream approach. In: Proc. of the 18th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR'95), ACM Press, pp. 189–197, 1995.

[RDS07] Ré, C., Dalvi, N., Suciu, D.: Efficient top-k query evaluation on probabilistic data. In: Proc. of the 23th International Conference on Data Engineering (ICDE'07), pp. 886–895, 2007.

[RIM85] Rao, N. S. V., Iyengar, S. S., Madhavan, C. E. V.: A comparative study of multiple attribute tree and inverted file structures for large bibliographic files. In: Proc. of Information Processing and Management, Vol. 21, No. 5, pp. 433-442, 1985.

[Rob81] Robinson, J. T.: The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In: Proc. of the 1981 ACM International Conference on Management of data (SIGMOD'81), ACM, New York, NY, USA, pp. 10-18, 1981.

[SG11] Šumák, M., Gurský, P.: Top-k search in product catalogues. In: Proc. of the 11th Annual International Workshop on Databases, Texts, Specifications, and Objects (DATESO'11), Eds. Snasel, V., Pokorny, J., Richta, K., CEUR Workshop Proceedings, Vol. 706, pp. 1-12, 2011.

[SG12] Šumák, M., Gurský, P.: Top-k search over grid file. In: Proc. of the 12th Annual International Workshop on Databases, Texts, Specifications, and Objects (DATESO'12), Eds. Pokorny, J., Snasel, V., Richta, K., CEUR Workshop Proceedings, Vol. 837, pp. 115-126, 2009.

[Sko06] Skopal, T.: On Fast Non-Metric Similarity Search by Metric Access Methods. In: Proc. of the 10th International Conference on Advances in Database Technology (EDBT'06), Springer-Verlag, LNCS 3896, pp. 718-736, 2006.

[SL08] T. Skopal, J. Lokoc. NM-tree: Flexible Approximate Similarity Search in Metric and Non-metric Spaces. In: Proc. of the 19th International Conference on Database and Expert Systems Applications (DEXA'08), Springer-Verlag, LNCS 5181, pp. 312-325, 2008.

[SO82] Scheuerman, P., Ouksel, M.: Multidimensional B-trees for associative searching in database systems. In: Information Systems Journal, Vol. 7, No. 2, pp. 123-137, 1982.

[SWC13] Sun, W., Wang, B., Cao N., Li, M., Lou, W., Hou, Y. T., Li, H.: Privacy-Preserving Multi-keyword Text Search in the Cloud Supporting Similarity-based Ranking. In: Proc. of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS'13), Hangzhou, China, May 7-10, 2013.

[TBM08] Theobald, M., Bast, H., Majumdar, D., Schenkel, R., Weikum, G.: TopX: efficient and versatile top-k query processing for semistructured data. In: The VLDB Journal, Vol. 17, No. 1, pp. 81-115, 2008.

[The06] Theobald, M.: Efficient Top-k Query Processing for Text, Semistructured, and Structured Data. Dissertation Thesis, Max-Planck-Institut für Informatik, Universität des Saarlandes, Germany, 2006.

[TPK03] Tsaparas, P., Palpanas, T., Kotidis, Y., Koudas, N., Srivastava, D.: Ranked Join Indices. In: Proc. of the 19th International Conference on Data Engineering (ICDE'03), pp. 277-288, 2003.

[TSW04] Theobald, M., Schenkel, R., Wiekum, G.: Top-k Query Processing with Probabilistic Guarantees. In: Proc. of the 30th International Conference on Very Large Data Bases (VLDB'04), pp. 648–659, 2004.

[TSW05] Theobald, M., Schenkel, R., Wiekum, G.: An efficient and versatile query engine for TopX search. In: Proc. of the 31th International Conference on Very Large Data Bases (VLDB'05), pp. 625–636, 2005.

[Voj06] Vojtáš, P.: Fuzzy logic aggregation for semantic web search for the best (top-k) answer. In: Capturing Intelligence, Elsevier, Amsterdam, Chapter 17, Vol. 1, pp. 341-359, 2006.

[WMD10] Wojnar, A., Mlýnková, I., Dokulil, J.: Structural and Semantic Aspects of Similarity of Document Type Definitions and XML Schemas. In: Special Issue on Intelligent Distributed Information Systems, Information Sciences, Elsevier, Vol. 180, No. 10, pp. 1817-1836, 2010.

[XHC07] Xin, D., Han, J., and Chang, K. C.: Progressive and selective merge: computing top-k with ad-hoc ranking functions. In: Proc. of the 2007 ACM International Conference on Management of data (SIGMOD'07), ACM, New York, NY, USA, pp. 103-114, 2007.

[YM07] Yiu, M. L., Mamoulis, N.: Efficient processing of top-k dominating queries on multi-dimensional data. In: Proc. of the 33th International Conference on Very Large Data Bases (VLDB'07), pp. 483–494, 2007.

[ZHC06] Zhang, Z., Hwang, S., Chang, K. C., Wang, M., Lang, C. A., Chang, Y.: Boolean + ranking: querying a database by k-constrained optimization. In: Proc. of the 2006 ACM International Conference on Management of data (SIGMOD'06), ACM, New York, NY, USA, pp. 359-370, 2006.