Composite Recommendation: Semantics and Efficiency

by

Min Xie

B.Eng., Renmin University of China, 2005
M.Eng., Renmin University of China, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

March 2015

Abstract

Classical recommender systems provide users with a list of recommendations where each recommendation consists of a single item, e.g., a book or DVD. However, many applications can benefit from a system which is capable of recommending packages of items. Sample applications include travel planning, e-commerce, and course recommendation. In these contexts, there is a need for a system that can recommend the most relevant packages for the user to choose from.

In this thesis we highlight our research achievements for the composite recommendation problem. We first consider the problem of composite recommendation under hard constraint, e.g., budget. It is clear that this is a very common paradigm for the composite recommendation problem. In Chapter 3, we first discuss how given a fixed package schema, we can efficiently find the top-k most relevant packages with hard constraints. The proposed algorithm is shown to be instance optimal, which means that no algorithm in a reasonable class can perform more than a constant times better, for some fixed constant. And we also propose relaxed solutions based on probabilistic reasoning. In Chapter 4, we lift the constraint on the package schema, and discuss how efficient algorithms can be derived to solve the more general problem with a flexible package schema. For this problem, again we propose both instance optimal algorithm and heuristics-based solution which have been verified to be effective and efficient through our extensive empirical study. Then in Chapter 5, motivated by the fact that hard constraints sometimes might lead to unfavorable results, and following the recent paradigm on "softening" the constraints, we study the problem of how to handle top-k query processing with soft constraints. Finally, in Chapter 6, we discuss a general performance tuning solution based on cached views which can be leveraged to further optimize the various algorithms proposed in this thesis.

Preface

This dissertation is the result of collaboration with several researchers. The most influential among them is my supervisor Laks V.S. Lakshmanan from the University of British Columbia, Canada. Most of the studies included in this dissertation are published in peer-reviewed venues.

Algorithms for composite recommendation with hard constraints and fixed package schema are presented in Chapter 3 and are based on our publication in VLDB 2011 [122]. In Chapter 4, we present algorithms which consider composite recommendation with hard constraints and flexible package schema. This chapter is based on our papers in RecSys 2010 [120] and ICDE 2011 [121]. In Chapter 5, we consider how soft constraints can be employed to handle limitations of hard constraints. And this chapter is based on our paper in VLDB 2013 [125] and VLDB 14 [126]. Finally, we consider how various algorithms for composite recommendation presented in this thesis can be optimized using cached views in Chapter 6, this work being based on our paper in EDBT 2013 [124].

All of the published works [120–122, 124–126] are in collaboration with Peter Wood from University of London, Birkbeck, U.K., and my supervisor Laks V.S. Lakshmanan from the University of British Columbia, Canada.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to express sincere gratitude to my supervisor, Dr. Laks V.S. Lakshmanan, for all the inspiring discussions, helpful suggestions, and constant encouragement throughout the course of my Ph.D. program. Without his support, I would not be able to make the achievement today, and it is my honor to have the chance to work with him.

I highly appreciate the help from Dr. Peter Wood from University of London, Birkbeck on most of my research work here in UBC, without the fruitful discussions with him and his excellent advices, those research would not be possible.

I'm also extremely thankful for Dr. Francesco Bonchi from Yahoo! Research Barcelona. His provoking thoughts and insights have always inspired me to keep exploring interesting and challenging research problems.

I would also like to thank my Ph.D. supervisory committee members, Dr. Raymond Ng, Dr. Alan Mackworth, and Dr. Giuseppe Carenini. Whose challenging questions and comments have helped a lot on improving this thesis.

Also I want to thank other faculty members in the DMM lab, Rachel Pottinger and Ed Knorr, from whom I have learned a lot through conversations, meetings, and classes.

I would like to thank my colleagues in the lab, Michael Lawrence, Hongrae Lee, Shaofeng Bu, Jian Xu, Solmaz Kolahi, Amit Goyal, Mohammad Khabbaz, Pooya Esfandiar, Ali Moosavi, Dibesh Shakya, Zhaohong (Charles) Chen, Xueyao (Sophia) Liang, Tianyu Li, Naresh Kumar Kolloju, Wei Lu, Pei Li, Rui Chen, Smriti Bhagat, Lan Wei, Shanshan Chen, Arni Thrastarson, Vignesh V.S., Zainab Zolaktaf, Yidan Liu, Glenn Bevilacqua, Keqian Li, Weicong Liao, you guys have made my five years in UBC so joyful.

Finally, I want to thank my family especially my beloved wife Mengya, who sacrificed so much to support me pursuing this career. Without her by my side, I wouldn't have come this far.

Chapter 1

Introduction

Recommender systems (RecSys) have become very popular of late and have become an essential driver of many applications. However, classical RecSys provide recommendations consisting of single items, e.g., books or DVDs. Several applications can benefit from a system capable of doing *composite recommendation*, or recommending packages of items, in the form of sets. For example, in trip planning, a user is interested in suggestions for places to visit, or points of interest (POI). There may be a cost associated with each visiting place (time, price, etc.). Optionally, there may be a notion of compatibility among items in a set, modeled in the form of constraints: e.g., "no more than 3 museums in a package", "not more than two parks", "the total distance covered in visiting all POIs in a package should be ≤ 10 km". The user may have a limited budget and may be interested in suggestions of compatible sets of POIs such that each set has a cost that is under a budget and has a value (as judged from ratings) that is as high as possible. In these applications, there is a natural need for the top-k recommendation packages for the user to choose from. Some so-called "third generation" travel planning web sites, such as NileGuide[1] and YourTour[2], are starting to provide certain of these features, although in a limited form.

Another application arises in social networks, like twitter, where one of the important challenges is helping users with recommendations for tweeters to follow, based on their topics of interest[3]. Tweeters are ranked based on how influential they are [119] and currently any new user is presented with a list of influential tweeters on each topic from which they manually choose tweeters they would like to follow[3]. To automate tweeter recommendation, a tweeter's influence score can be treated as their value and the frequency with which they tweet as their cost. Compatibility may correspond to the constraint that a given set of topics should be covered. Given a user's topics of interest as well as a budget representing the number of tweets the user can deal with in a day, it would be useful to select compatible sets of tweeters to

[1] `http://www.nileguide.com` (visited on 03/16/2015)
[2] `http://www.yourtour.com` (visited on 03/16/2015)
[3] `https://blog.twitter.com/2010/power-suggestions` (visited on 03/16/2015)

follow such that their total influence score is maximized and the total cost is within budget. Once again, it would be beneficial to give the user choice by presenting them with the top-k sets of recommended tweeters to follow. We note that some newly founded startups like Followformation[4] are beginning to provide services on recommending to users the top-k influential tweeters in a specific domain.

As a third application, consider that a university student who wishes to choose courses that are both highly rated by past students and satisfy certain degree requirements. Assume each course is assigned a level, a category, and a number of credits. In order to obtain an MSc degree, students must take 8 modules, subject to the following further constraints: (i) at least 75 credits must come from courses in the "database" category, (ii) the minimum level of any course taken is 6, and (iii) the maximum number of credits taken at level 6 is 30. The requirements above can be expressed as a conjunction of aggregation constraints. And based on popularity and other information available, different sets of courses which satisfy all aggregation constraints can be ranked to find the top most interseting ones to the student.

From these examples, we can easily infer that a standard recommendation engine which generates lists of items can be quite overwhelming for the user since the user needs to manually figure out the package of items, and the potential number of underlying items is huge. Thus we need a novel system which is capable of capturing users' preferences and recommending high quality packages of items.

1.1 Challenges

1.1.1 Semantics

One of the core issues in the composite recommendation problem is how we can determine the utility of a specific package for a user. To answer this question, we note that there are usually two most important criteria for determining the value of a package for a user: 1. the quality of the package, e.g., the sum of the ratings of items within the package; 2. the constraints specified by the user, e.g., no more than $500 for the cost.

For user-specified constraints, there are again two popular paradigms of handling them. First, we can treat these users' preferences as hard constraints, e.g., if a user specifies a cost budget of $500, only packages of which the cost is within the budget will be considered. This paradigm is very useful

[4] `http://followformation.com (visited on 03/16/2015)`

for the cases where users know their preferences exactly.

On the other hand, we can also treat these users' preferences as soft constraints. For example, suppose we have a budget constraint on cost. Then we can assign a score to each package based on its cost budget: the higher the cost budget, the lower the score. Then this means we do not necessarily rule out those packages which do not satisfy the constraint. And this will be particularly useful for cases where users are not 100% sure of their preferences. For example, though a user set a cost budget of $500 on a trip to New York, he might be well interested in a package which cost slightly higher than $500, but includes many high quality places of interest and good restaurants.

Given a specific paradigm of handling user-specified constraints, there are still multiple ways of determining the utility of a specific package to a user. E.g., considering hard constraints based approach, given individual item's utility to the user, how to determine the utility of a set of items to the user? Or for the soft constraints based approach, given the score based on individual item's utility, and also a score based on the constraints, how to estimate the overall utility of a package for the user? As shown by many previous works, we can leverage an additive utility function f which can be used to rank packages [27, 34]. However, the challenge of this approach lies in the fact that we need to determine the parameters associated with function f. For example, in the database community [63], usually it is assumed that f is given by the user. This assumption might be too strong considering the fact that users many often not know their preference exactly. Thus a more promising way for determining the utility function is through interaction with the users using preference elicitation, as demonstrated in [27, 34].

1.1.2 Efficiency

Given different semantics of the composite recommendation problem as discussed in the previous section, another core issue is the efficiency of the algorithm for finding the most relevant packages.

Consider the user-specified constraints, usually hard constraints will lead to NP-hard optimization problems such as Knapsack [120], and Orienteering [123], which render the underlying problem computationally difficult to scale to a large dataset. And on the other hand, for soft constraints, even if we can sort and find the top-k packages by an efficient algorithm, determining the utility function which will be used for ranking might itself be a challenging problem. E.g., as demonstrated in [27, 34], under a reasonable uncertainty setting, the candidate utility function can range over an unlimited set of

possibilities.

1.2 Key Contributions

As we shall see from later chapters of this thesis, different composite recommendations problems might have significantly different types of properties, thus instead of proposing a universal one-size-fits-all solution, we believe the more optimal way is to exploit underlying different properties of different problem settings. So in this thesis, depending on structure of the underlying composite recommendation problem, we propose a portfolio of solutions (Chapter 3,4,and 5) which can be selected from and tailored to satisfy different needs of the underlying application, and we also provide toolkits such as cached views (Chapter 6) which can be leveraged to optimize various proposed composite recommendation algorithms.

1.2.1 Composite Recommendation under Hard Constraints

In [123], we consider the problem of performing composite recommendation under hard constraints and having a fixed package schema (E.g., each package has exactly one hotel, and one restaurant). We consider a simple additive utility function, and connect this problem to existing solutions on rank join [86] by extending these algorithms with aggregation constraints. By analyzing their properties, we developed deterministic and probabilistic algorithms for their efficient processing. In addition to showing that the deterministic algorithm retains the minimum number of accessed tuples in memory at each iteration, we empirically showed both our deterministic and probabilistic algorithms significantly outperform the obvious alternative of rank join followed by post-filtering in many cases and that the probabilistic algorithm produces results of high quality.

In [120] and [121], we consider the more general case of composite recommendation by lifting the constraint on package schema. We proposed the problem of generating top-k package recommendations that are compatible and are under a cost budget, where a cost is incurred by visiting each recommended item and the budget and compatibility constraints are user specified. We identify the problem of finding the top package as being intractable since it is a variant of the Knapsack problem, with the restriction that items need to be accessed in value-sorted order. So we developed two 2-approximation algorithms that are designed to minimize the number of items accessed based on simple statistics (e.g., minimum value) about item costs. The first of these, InsOpt-CR-Topk, is instance optimal in a strong

sense: every 2-approximation algorithm for the problem must access at least as many items as this algorithm. The second of these, Greedy-CR-Topk, is not guaranteed to be instance optimal, but is much faster. We experimentally evaluated the performance of the algorithms and showed that in terms of the quality of the top-k packages returned both algorithms are close to each other and deliver high quality packages; in terms of the number of items accessed Greedy-CR-Topk is very close to InsOpt-CR-Topk, but in terms of running time, Greedy-CR-Topk is much faster. We also showed that using histogram-based information about item costs, rather than simply knowledge of the minimum item cost, further reduces the number of items accessed by the algorithms and improves their running time.

1.2.2 Composite Recommendation under Soft Constraints

We also study how composite recommendation is possible using soft constraints [125, 126]. Following [27, 34], we assume the system does not have the complete information about user's utility function, and we leverage the existing preference elicitation frameworks for eliciting preferences from users. However the challenge here is how can we perform the elicitation efficiently, especially considering the fact that we are reasoning about utilities of combinations of items. We propose several sampling-based methods which, given user feedback, can capture the updated knowledge of the underlying utility function. Finally, we also study various package ranking semantics for finding top-k packages, using the learned utility function.

1.2.3 Efficiency Optimization

A key component in the composite recommendation problem is the searching of the top-k packages under a given semantics. In classical database query optimization, use of materialized views is a popular technique for speeding up query processing. Recently, it was extended to top-k queries [45]. In [124] we consider a general optimization procedure based on cached views which can be leveraged to further reduce the computational cost of processing top-k queries. We show that the performance of the state-of-the-art top-k query answering using view algorithm LPTA [45] suffers because of iterative calls to a linear programming sub-procedure, which can be especially problematic when the number of views is large or if the dimensionality of the dataset is high. By observing an interesting characteristic of the LPTA framework, we proposed LPTA$^+$, an improved algorithm for using cached top-k views for efficient query processing, which has greatly outperformed

LPTA. Furthermore, LPTA is not directly applicable in our setting of top-k query processing with cached views, where views are not complete tuple rankings and base views are not available. Thereto, we adapted both algorithms so that they can overcome such limiting assumptions. Finally, we proposed an index structure, called IV-Index, which stores the contents of all cached views in a central data structure in memory, and we can leverage IV-Index to answer a new top-k query much more efficiently compared with LPTA and LPTA$^+$. Using comprehensive experiments, we showed LPTA$^+$ substantially improves the performance of LPTA while the algorithms based on IV-Index outperform both these algorithms by a significant margin. We discuss in this thesis how the proposed optimization framework can be integrated into various composite recommendation algorithms proposed in this thesis.

1.3 Thesis Outline

The rest of this dissertation is organized as follows. In Chapter 2, we provide a brief background and review related work. In Chapter 3, we consider the first problem in composite recommendation which deals with fixed package schema and hard constraints. In Chapter 4, we lift the constraint on the schema of the package, and consider the problem of composite recommendation with flexible package schema and hard constraint. In Chapter 5, we consider how soft constraints can be considered in the composite recommendation framework. We also discuss how to elicit user's preference using implicit feedback. Finally, in Chapter 6, we discuss a general tuning framework based on cached views which can be leveraged to improve performance of various proposed top-k package searching algorithms. In Chapter 7, we summarize this dissertation and list directions for future research in composite recommendation.

Chapter 2

Related Work

2.1 Composite Recommendation

2.1.1 Application in Trip Planning

Our composite recommendation problem is most related to recent studies on *travel package recommendation*. In [19], the authors are interested in finding the top-k tuples of travel entities. Examples of entities include cities, hotels and airlines, while packages are tuples of entities. Instead of querying recommender systems, they query documents using keywords in order to determine entity scores. Similar to our work [123], a package in their framework is of fixed schema, e.g., one city, one hotel, and one airline, with fixed associations among the entities essentially indicating all possible valid packages.

Obviously in many real world scenarios, we would like to have flexible packages schema, thus frameworks which allow flexible package schema configuration were proposed by several researchers. A representative work in this category is [47], in which the authors propose a novel framework to automatically generate travel itineraries from online user-generated data like picture uploads and formulate the problem of recommending travel itineraries of high quality where the travel time is under a given time budget. However, the value of each POI in [47] is determined by the number of times it is mentioned by users, whereas in our work [120, 121], item value is a personalized score which comes from an underlying recommender system, and we consider the very practical setting where accessing these items is constrained to be in value-sorted order. Similar settings of [47] are also explored in [92]. In [39], the authors extend the previous works by considering how multi-day trips can be planned. As discussed in [47] and [121], when travel time between POIs is taken into consideration, the underlying problem is closely related to the classical *Orienteering problem* [38], which seeks a maximum value walk on a graph subject to a budget constraint on the cost. However, unlike [120, 121], the orienteering problem does not take POI access cost into consideration which can be less desirable since the number of POIs in

the database might be huge.

In the database community, researchers have considered the travel package recommendation problem from the query processing perspective. E.g., in [111], the authors considered the *optimal sequenced route* (OSR) query which returns a sequence of POIs which satisfy the following two properties: 1. the sequence of POIs follow exactly a given "POI type template" which specifies order and type of each POI; 2. the total travel distance of the returned POIs is minimized. In [40], the authors extend OSR by considering partial sequence-based POI type template. In [88], the authors propose *trip planning queries* (TPQ), with which the user specifies a subset (not a sequence) of location types and asks for the optimal route from a given starting location to a specified destination which passes through at least one POI of each type specified. In [69, 83], the authors consider a similar query type as OSR with which POI type templates are specified using keyword-based query. Clearly, the query oriented travel package search problem requires users to know exactly or at least roughly what they want in a travel package, which may not be practical in real world travel applications. In this thesis, we are more interested in generating travel packages by leveraging users' previous travel behavior and minimizing the amount of information that needs to be provided by the user, which is similar as the state-or-the-art recommender systems [16].

2.1.2 Application in Course Planning

In [74, 99, 102], the authors study various composite recommendation problem related to *course planning*. The resulting product of these works, CourseRank [102], is a project motivated by a Stanford course planning application for students, where constraints are of the form "take k_i from S_i," where k_i is a non-negative integer and S_i is a set of courses. Similar to our work [120], each course in this system is associated with a score which is calculated using an underlying recommendation engine. Given a number of constraints of the form above (and others), the system finds a minimal set of courses that satisfies the requirements and has the highest score. In [99, 101], the authors extend CourseRank with prerequisite constraints, and proposes various algorithms that return high-quality course recommendations which satisfy all the prerequisites. Similar to [120], such recommendations need not be of fixed size. However, [99, 101, 102] do not consider the *cost* of items (cf. courses) which can be important for many composite recommendation applications.

2.1.3 Application in E-commerce

Finally, *E-commerce* represents another promising application in which composite recommendation engines can be very useful. In [106], the authors study the problem of recommending "satellite items" related to a given "central item" subject to a cost budget. Every package found by the proposed algorithm is composed of one specific central item and several satellite items. Clearly, the target of this work is a dedicated engine tailored for a specific type of recommendation, which is different from our works such as [120, 121, 123] which targets the more general composite recommendation problem without a specific shape in mind for the recommendation package. However, we note that we could easily extend our algorithm to make such recommendations by post-filtering as discussed in [120]. In [37], the authors propose a framework which can help users search items of fixed pair-wise relationships. The problem studied in [37] is similar to [19], thus the proposed framework is not intended to search packages with flexible schema.

In [56], the authors design a shopping tool which can help users find existing deals about bundles of items from various E-commerce sources. Compared with our work [120, 121, 123, 124], the shopbot proposed in [56] cannot create new packages which are tailored to users' interest, but instead focus on finding pricing and other information of *existing* packages which are provided from different sources. The general framework of pricing strategies of item bundles in marketing science has been discussed in [23].

The composite recommendation is also related to *Combinatorial Auction* in E-commerce [48], since the underlying objects in combinatorial auction are also packages of items. However, combinatorial auction focuses on determining item or package allocation under a multi-participant scenario, whereas in the composite recommendation problem studied in this thesis, we focus on making personalized recommendation where there exists no competition for packages among users, which is in alignment with the state-or-the-art recommender systems [16].

2.1.4 Other Applications

In addition to travel planning, course planning, and E-commerce, there also exist other applications where composite recommendation can be extremely beneficial. E.g., in [32], the authors proposed a framework CARD which provides the top-k recommendations of composite products or services. Fine-grained control over specifying user requirements, as well as how atomic costs are combined, is provided by an SQL-like language extended with

features for decision support. Each composite recommendation is of fixed schema, making the problem simpler; and CARD returns only exact, not approximate, solutions.

As another example, the problem of team formation studied in [78] can also be considered as a composite recommendation problem. Here each person has a set of skills and pairs of people have a collaboration cost associated with them (lower cost indicates better collaboration). Given a task requiring a set of skills, the problem is to find a set of people whose skills cover those required and who have a low aggregated collaboration cost. The notion of compatibility in [120] can model their collaboration cost. Similar to CourseRank, in [78], the people (items) themselves are not rated. A further difference with [120] is that in [120] we wish to maximize the aggregate item (people) ratings subject to item and compatibility costs, rather than minimize compatibility cost.

We note that although we do not include in our system complex constraints such as those in [78, 99, 101, 102, 116], for applications where complex constraints exist, we can leverage existing work to post-process each composite recommendation generated by our algorithms to ensure that the constraints are satisfied.

2.2 Preference Handling and Elicitation

There has been much investigation into the handling of preferences over items, e.g., general preference frameworks [72] [41], skyline queries [25, 95, 96], and top-k queries [63]. While different approaches for reasoning about preferences might be suitable for different applications, as discussed in [70], a more popular and practical approach is to leverage a general utility function which succinctly characterizes users' trade-offs among different properties of the items under consideration. Among various forms of utility function which have been studied, the most popular one is the simple linear utility function or additive utility function [63, 70].

While item preference reasoning has been popular for a long time, only recently have researchers started considering preference handling for packages, or sets of items. This calls for exploring a much larger candidate space, and usually has an aggregation-based feature space (e.g., total cost budget or average rating), which further complicates the underlying problem. Existing set preference works by researchers in artificial intelligence usually focus on the formal aspects of this problem, e.g., expressiveness of the preference language. These works include [31] which considers generaliz-

ing item preferences to set preferences through *order lifting* and *incremental improvements*, and [29, 30] which considers extensions of CP-nets proposed in [28] to sets of items. But the proposed models in these works are often not practical for applications with large amount of items. As commented in [29], the reason is because a language for specifying preferences between sets of items of arbitrary size, to be understood ceteris paribus, there is a inherent high complexity. E.g., comparing two sets of items under the preference language proposed in [29] is PSPACE-complete, and algorithms proposed in [30] has a worst-case exponential time complexity. In [51] and [104], the authors study preferences over sets with an emphasize on diversity of the underlying sets.

In [129] and [87], the authors study skyline packages of fixed cardinality, which finding packages of fixed cardinality and are better than or equal to other packages on all attributes under consideration. However, a severe drawback of this approach is that the number of skyline packages is usually prohibitive. For example, in [129], the number of skyline packages can be in the hundreds or even thousands for a reasonably-sized dataset.

In this dissertation, following the popular paradigm of reasoning preference about individual items [63, 70], we take a quantitative and utility function-based approach for reasoning about preference under the composite recommendation framework. Specifically, we consider the utility of a package to be a linear weighted combination of various properties of the package (e.g., quality and cost), and the package property can be further described by aggregations over attribute values of items within the package.

However, though linear utility function can be leveraged to model tradeoffs among different package properties, still weights of the utility function need to be determined. A simple approach for reasoning about preference is to assume weights of the utility function are given by the user, e.g., as we have discussed in [120, 121, 123]. However, this may not be practical since users usually cannot know the weights for sure. For example, users would not be able to tell the system that they are 0.8 interested in the overall cost, and 0.2 interested in the overall quality of a package. Thus in Chapter 5, we propose to extend our works in Chapter 3 and Chapter 4, and study how weights of the utility function can be learned through implicit feedback using *preference elicitation* techniques [27, 34].

For eliciting preferences, most existing works have been focusing on items instead of packages. In [130], the authors propose an interactive way to elicit preferred items. The paper does not consider preferences for packages, and more importantly, they assume that the weight parameters of the underlying utility function follow a uniform distribution, and do not discuss how

user feedback can be leveraged to update the utility function. In [95] and [67], the authors have a similar setting of inferring preferences given some partial comparisons of items. However, these two papers focus on inferring a most desirable order directly using these given partial comparisons; in our work, we took a different Bayesian-based approach by modeling the parameters of the utility function using a distribution. The most desirable order of packages depends directly on the uncertain utility function following different ranking semantics. Feedback received only affects the posterior of the parameter distribution.

In [107], the authors consider an interactive way of ranking travel packages. However, the user feedback model in [107] is defined in such a way that for each iteration, the user is asked to rank a set of items instead of a set of packages. Thus every decision the user makes is "local" in the sense that the user is not able to personalize her/his preference over packages as a whole; it is possible that items favored in one iteration might become less desirable after seeing some additional items. Also unlike our framework in which the elicitation of preferences is implicit, [107] requires several iterations of explicit preference elicitation before the system would show the user any recommended package.

Compared with existing work on interactive preference elicitation for items [95, 96], our search space of candidate packages is much larger, and we consider features which are based on aggregations of item attribute values, thus the problem becomes more challenging.

2.3 Top-k Query Processing

Given a linear utility function, for both hard constraints based composite recommendation (as in Chapter 3 and Chapter 4), and soft constraints based composite recommendation (as in Chapter 5), the core of the composite recommendation algorithms is in finding the most promising packages given the linear utility function [120, 121, 123, 125]. As we will demonstrate in this dissertation, this problem can be cast as a variation of the classical problem of top-k query processing [53].

For general top-k query processing, the most popular approach is the family of algorithms embodied by *Threshold Algorithm* (TA) / *No Random Access Algorithm* (NRA) as proposed by Fagin et al. in [53]. While TA and NRA differ in whether random access to the database is allowed, this family of algorithms usually share a similar query processing framework which is outlined in Algorithm 1. In this algorithm framework, given a database R

of items, a query vector w for the linear utility function, and the number of items required k, we first sort items into a set of lists w.r.t. each attribute under consideration. Then these lists are accessed following a certain order (usually round robin). For each item t accessed, we can put t into the result queue if it is better than the kth item in the queue. Because items are sorted in each list w.r.t. w or the desirable order, we could determine an upperbound value τ on the possible value which can be achieved by any unseen items. If the current kth item in R has value larger than or equal to τ, we know we have already found the top-k items w.r.t. w.

Algorithm 1: TA(R, w, k)

```
1  L ← Lists of items in R which are sorted w.r.t. every attribute;
2  O ← Priority queue for the top-k results;
3  τ ← ∞;
4  while |O| < k ∨ value(O.kthResult) ≤ τ do
5      L_i ← Next attribute following round robin;
6      t ← L_i.next();
7      if |O| < k ∨ value(O.kthResult) < value(t) then
8          O.insert(t);
9      τ ← Upperbound value given current access positions in each list;
10 return O
```

Recently, various improvements to the original algorithms such as the *Best Position Algorithm* [18] have been proposed, while variations of top-k queries such as *Rank Join* [62] and *Continuous Top-k Queries* [128] have been studied. Finally, Li et al. study how top-k algorithms might be implemented in a relational database [86]. An excellent survey on top-k query processing can be found in [63].

2.3.1 Domination-based Pruning and Variable Elimination

As discussed in the main TA algorithm framework, the key in top-k query processing is to maintain in the memory a set of promising top-k result candidates, and iteratively check whether there exist a set of k candidates which can "beat" any of the remaining items w.r.t. the query, thus becoming the optimal top-k results. As we shall see in later chapters of this thesis, the performances of many variations of the top-k algorithms depends on how many candidate items we need to maintain in the memory, thus a common

way to facilitate top-k query processing is to prune away items which are not able to make into the top-k results as early as possible.

A popular approach for this purpose is the *domination-based pruning*, of which the idea can be described as follows. Consider a linear utility function whose parameters are described by w. For an item t, we say it dominates another item t', $t \succ t'$, if t is better than or equal to t' on every criterion w.r.t. w, and is better than t' on at least one criterion. Then obviously w.r.t. w, there is no hope for t' to make into the top-1 result given the existence of t, thus t' can be eliminated from consideration. This domination-based pruning can be easily extended to the top-k case.

We discuss in [123] the properties of aggregation constraints in the composite recommendation framework and develop an efficient algorithm for processing rank joins with aggregation constraints, based on two strategies for domination-based pruning. We note that similar approaches were also explored in [129] and [87] in the context of skyline sets.

The domination-based pruning is also related to *variable elimination* (VE) in solving constraint satisfaction problem (CSP) [76]. The key idea here is to transform the original CSP into a new reduced CSP which is equivalent to the original CSP. To enable such transformation, we consider variables one by one. For each variable X under consideration, we first take all constraints which involve X and generate intermediate partial solutions which satisfy each constraint under investigation. A join operation is performed to merge all intermediate partial solutions, and then variable X can be removed from consideration by generating new constraints which do not involve X. E.g., as discussed in [76], consider a CSP that contains the variables A, B, and C. Suppose the only constraints under consideration are $A < B$ and $B < C$, then we could remove B by joining partial solutions of the CSP which satisfy each of the above two constraints, and consider a new constraint $A < C$ for the join result. Clearly, for each join result satisfying $A < C$, there must exist an assignment to variable B which extends it to a solution of the original problem.

Clearly, both domination-based pruning and variable elimination correspond to approaches which aim at removing variables/items which do not contribute to the optimal solution of the underlying problem. However, unlike VE which aims at removing variables by "marginalizing" their effects on the underlying problem, domination-based pruning aims at removing variables/items which are not promising under a given query. Also most existing works on VE does not consider aggregation effects over variables, which is a key in the composite recommendation problem, as constraints are usually specified as aggregations over item attribute values.

2.4 Constraint Optimization

In *constraint optimization* the task is to find a solution that optimizes some cost function and satisfies the specified constraints [49, 76]. The constraint optimization problem finds applications in various domains such as planning, scheduling, and auction.

As discussed in [49], the general constraint optimization problems are usually solved by branch-and-bound search algorithms or dynamic programming. For the search-based algorithm, the efficiency of the algorithm depends on its ability to cut the branches which do not lead to the optimal solution during the search process. This dead branch detection is done usually with a heuristic function which computes a lower bound of the current subproblem at the branch under consideration. E.g., we can use either weighted CSP local consistency [79] or mini-bucket elimination [50]. Dynamic programming was proposed as an alternative to the branch-and-bound search [49], and the algorithm was introduced in the context of sequential decision making.

When the underlying constraints and cost functions in a constraint optimization problem are all linear, the constraint optimization problem can also be solved through general integer programing or linear programming solvers such as CPLEX [5].

In the composite recommendation problem studied in this thesis, each underlying constraint optimization problem usually takes a specific restricted form. Thus instead of leveraging a general purpose solver, we can exploit the property of the underlying problem and consider solvers of simpler problems such as Knapsack [71] for the composite recommendation problem with hard constraint and flexible schema (Chapter 4), and RankJoin [62] for the composite recommendation problem with hard constraint and fixed schema (Chapter 3).

[5] `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/` (visited on 03/16/2015)

Chapter 3

Composite Recommendation with Hard Constraints: Fixed Package Schema

3.1 Introduction

In this chapter, we first discuss how composite recommendation problem with hard constraint and fixed package schema can be efficiently solved.

In the last several years, there has been tremendous interest in rank join queries and their efficient processing [54, 62, 110]. In a rank join query, you are given a number of relations, each containing one or more *value* attributes, a monotone score aggregation function that combines the individual values, and a number k. The objective is to find the top-k join results, i.e., the join results with the k highest overall scores. Clearly, composite recommendation with fixed package schema can be directly casted to a rank join problem. E.g., additive utility function can be considered as a specific instance of the montone score aggregation function, and the only thing missing in rank join is how to handle the user specified constraints, which will be addressed in this chapter.

Rank join can be seen as a generalization of classic top-k queries where one searches for the top-k objects w.r.t. a number of criteria or features [53]. For classic top-k queries, assuming that objects are stored in score-sorted inverted lists for each feature, the top-k objects w.r.t. a monotone score aggregation function can be computed efficiently using algorithms such as TA, NRA and their variants [53]. These algorithms satisfy a property called *instance optimality*, which intuitively says that no algorithm in a reasonable class can perform more than a constant times better, for some fixed constant.

Ilyas et al. [62] were the first to develop an instance-optimal algorithm for rank join queries involving the join of two relations. Their algorithm employs the so-called corner-bounding scheme. Polyzotis et al. [110] showed that whenever more than two relations are joined or relations are allowed to

contain multiple value-attributes, the corner bounding scheme is no longer instance optimal. They proposed a tight bounding scheme based on maintaining a "cover set" for each relation, and using this bounding scheme results in instance optimal algorithms [54, 110].

For the composite recommendation problem, as we have discussed before, constraints can add considerable value [85, 97, 103, 105] in two ways. First, they allow the relevant application semantics to be abstracted and allow users to impose their application-specific preferences on the query (or mining task) at hand. Second, constraints can often be leveraged in optimizing the query or mining task at hand. In this chapter, we argue that *aggregation constraints* can enrich the framework of rank join queries by including such application semantics.

We should highlight the fact that, in our constraints, aggregation is applied to values appearing in *each* tuple resulting from a join, rather than in the traditional sense where aggregation is over *sets* of tuples. In this sense, aggregation constraints exhibit some similarity to selections applied to a join.

A natural question is how to process rank joins with aggregation constraints efficiently. A naive approach is to perform the rank join, then apply post-filtering, dropping all results that violate the constraints, and finally report the top-k among the remaining results. We show that rank joins with aggregate constraints can be processed much faster than this post-filtering approach. First, we develop techniques for pushing constraint processing within the rank join framework, allowing irrelevant and "unpromising" tuples to be pruned as early as possible. As a result, we show that tuples that will not contribute to the top-k answers can be detected and avoided. Second, based on the observation that such an optimized algorithm still needs to access many tuples, we propose a probabilistic algorithm which accesses far fewer tuples while guaranteeing the quality of the results returned.

Specifically, we make the following contributions in this work:

- we introduce the problem of efficient processing of rank join queries with aggregation constraints (Sec. 6.2), showing the limitations of the post-filtering approach (Sec. 3.3);
- we analyze the properties of aggregation constraints and develop an efficient algorithm for processing rank joins with aggregation constraints, based on two strategies for pruning tuples (Sec. 3.4);
- we also develop a probabilistic algorithm that terminates processing in a more aggressive manner than the deterministic approach while

guaranteeing high quality answers (Sec. 5.2);

- we report on a detailed set of experiments which show that the execution times of our algorithms can be orders of magnitude better than those of the post-filtering approach (Sec. 3.6).

3.2 Problem Definition

Consider a set $\mathbf{R}$ of n relations $\{R_1, R_2, \ldots, R_n\}$, with R_i having the schema $schema(R_i)$, $1 \leq i \leq n$. For each tuple $t \in R_i$, the set of attributes over which t_i is defined is $schema(t) = schema(R_i)$. We assume each relation has a single *value* attribute V, and (for simplicity) a single join attribute J.[6] Given a tuple $t \in R_i$ and an attribute $A \in schema(t)$, $t.A$ denotes t's value on A. We typically consider join conditions jc corresponding to equi-joins, i.e., $J = J$.

Let $\mathbf{R}' = \{R_{j_1}, R_{j_2}, \ldots, R_{j_m}\} \subseteq \mathbf{R}$. Given a join condition jc, we define $s = \{t_1, \ldots, t_m\}$ to be a *joinable set* (JS) if $t_i \in R_{j_i}$, $i = 1, \ldots, m$, and $\bowtie_{jc} {}_{i=1}^{m}\{t_i\} \neq \emptyset$. If $m = n$, we call s a *full joinable set* (FJS), while if $m < n$ we call s a *partial joinable set* (PJS). We denote by **JS** the set of all possible (partial) joinable sets. Furthermore, for a JS s which comes from $\mathbf{R}'$, we define $Rel(s) = \mathbf{R}'$.

3.2.1 Language for Aggregation Constraints

Aggregation Constraints can be defined over joinable sets. Let $AGG \in \{MIN, MAX, SUM, COUNT, AVG\}$ be an aggregation function, and let the binary operator θ be $\leq$, $\geq$ or $=$.[7] Let $p ::= A\ \theta\ \lambda$ be an *attribute value predicate*, where A is an attribute of some relation, θ is as above, and λ is a constant. We say tuple t satisfies p, $t \models p$, if $A \in schema(t)$ and $t.A\ \theta\ \lambda$ is *true*. An attribute value predicate p can be the constant *true* in which case every tuple satisfies it. A set of tuples s satisfies p, $s \models p$, if $\forall t \in s$, $t \models p$.

We now consider aggregation constraints which are applied to tuples resulting from a join. A *primitive aggregation constraint* (PAC) is of the form $pc ::= \mathrm{AGG}(A, p)\ \theta\ \lambda$, where AGG is an aggregation function, A is an attribute (called the *aggregated attribute*), p is an attribute value predicate (called the *selection predicate*) as defined above, and θ and λ are defined as

[6] Our results and algorithms easily extend to more general cases of multiple value-attributes and/or multiple join attributes, following previous work such as [54].

[7] Operators $<$ and $>$ can be treated similarly to $\leq$ and $\geq$.

above. Given a joinable set s, we define

$$Eval_{pc}(s) = \text{AGG}([t.A \mid t \in s \wedge t \models p])$$

where we use $[\ldots]$ to denote a multiset. Then we say s satisfies the primitive aggregation constraint pc, $s \models pc$, if $Eval_{pc}(s)\ \theta\ \lambda$ holds.

The language for (full) aggregation constraints can now be defined as follows:

$$\begin{array}{ll} \text{Predicates:} & p ::= true \mid A\ \theta\ \lambda \mid p \wedge p \\ \text{Aggregation Constraints:} & ac ::= pc \mid pc \wedge ac \\ & pc ::= \text{AGG}(A, p)\ \theta\ \lambda \end{array}$$

The meaning of a full aggregation constraint ac is defined in the obvious way, as are the notions of joinable sets satisfying ac and the satisfying subset R^{ac} of a relation R resulting from a join.

Let R be a relation resulting from a (multi-way) join $R_1 \bowtie_{jc} \cdots \bowtie_{jc} R_m$. Each tuple $t \in R$ can also be viewed as a joinable set s_t of tuples from the relations R_i. Given an aggregation constraint ac, we define R^{ac} as $\{t \mid t \in R \wedge s_t \models ac\}$.

Note that by adding a special attribute C to each relation and setting the value of each tuple on C to be 1, $COUNT$ can be simulated by SUM. Similarly, when the number of relations under consideration is fixed, AVG can also simulated by SUM. So to simplify the presentation, we will not discuss $COUNT$ and AVG further.

3.2.2 Problem Studied

We assume the domain of each attribute is normalized to $[0, 1]$. Let $\mathbb{R}$ denote the set of reals and $S : \mathbb{R}^n \to \mathbb{R}$ be the score function, defined over the value attributes of the joined relations. Following common practice, we assume S is *monotone*, which means $S(x_1, ..., x_n) \leq S(y_1, ..., y_n)$ whenever $\forall i,\ x_i \leq y_i$. To simplify the presentation, we will mostly focus on S being SUM, so given a joinable set s, the overall value of s, denoted as $v(s)$, can be calculated as $v(s) = \sum_{t \in s} t.V$. Furthermore, in this chapter we assume that the join condition jc is equi-join, which means that given two tuples t_1 and t_2 from two relations, $\{t_1\} \bowtie_{jc} \{t_2\} \neq \emptyset$ iff $t_1.J = t_2.J$. For brevity we will omit the join condition jc from the join operator when there is no ambiguity.

Let ac be a user-specified aggregation constraint (which may be a conjunction of PACs) and jc be the join condition. We study the problem of *Rank Join with Aggregation Constraints* (RJAC):

Definition 1 Rank Join with Aggregation Constraints: Given a set of relations $\mathbf{R} = \{R_1, \ldots, R_n\}$ and a join condition jc, let RS denote $\bowtie_{i=1}^{n} R_i$. Now given a score function S and an aggregation constraint ac, find the top-k join results $RS_k^{ac} \subseteq RS^{ac}$, that is, $\forall s \in RS_k^{ac}$ and $\forall s' \in RS^{ac} - RS_k^{ac}$, we have $v(s) \geq v(s')$.

We denote an instance of the RJAC problem by a 5-tuple $\mathbf{I} = (\mathbf{R}, S, jc, ac, k)$. Because we are usually only interested in exactly k join results, we will discard potential join results which have the same value as the k^{th} join result in RS_k^{ac}; however, the proposed technique can be easily modified to return these as well if needed. Our goal is to devise algorithms for finding the top-k answers to RJAC as efficiently as possible.

3.3 Related Work

3.3.1 Rank Join and Post-Filtering

The standard rank join algorithm with no aggregation constraints works as follows [62, 110]. Given a set of relations $\mathbf{R} = \{R_1, \ldots, R_n\}$, assume the tuples of each relation are sorted in the non-increasing order of their value. The algorithm iteratively picks some relation $R_i \in \mathbf{R}$ and retrieves the next tuple t from R_i. Each seen tuple $t \in R_i$ is stored in a corresponding buffer HR_i, and t is joined with tuples seen from $HR_j, j \neq i$. The join result is placed in an output buffer O which is organized as a priority queue. To allow the algorithm to stop early, the value of t is used to update a stopping threshold τ, which is an upperbound on the value that can be achieved using any unseen tuple. It can be shown that if there are at least k join results in the output buffer O which have value no less than τ, the algorithm can stop, and the first k join results in O are guaranteed to be the top-k results.

To characterize the efficiency of a rank join algorithm, previous work has used the notion of *instance optimalilty*, proposed by Fagin et al. [53]. The basic idea is that, given a cost function *cost* (which is a monotone function of the total number of tuples retrieved), with respect to a class $\mathcal{A}$ of algorithms and a class $\mathcal{D}$ of data instances, a top-k algorithm A is instance optimal if, for some constants c_0 and c_1, for all algorithms $B \in \mathcal{A}$ and data instances $D \in \mathcal{D}$, we have $cost(A, D) \leq c_0 \times cost(B, D) + c_1$.

Instance optimality of a rank join algorithm is closely related to the *bounding scheme* of the algorithm, which derives the stopping threshold at each iteration. It has been shown in [110] that an algorithm using the *corner-bounding* scheme [62] is instance optimal if and only if the underlying

join is a binary join and each relation contains one value attribute. To ensure instance optimality in the case of multiple value attributes per relation and multi-way rank join, Schnaitter et al. [110] proposed the *feasible region* (FR) bounding scheme. This FR bound was later improved by Finger and Polyzotis [54] using the *fast feasible region* (FR*) bounding scheme.

Suppose each relation has m value attributes, then the basic idea of FR/FR* bounding scheme is to maintain a *cover set* CR_i for each relation R_i. CR_i stores a set of points that represents the m-dimensional boundary of the values of all unseen tuples in R_i. Given an n-way rank join over $\mathbf{R} = \{R_1, \ldots, R_n\}$, to derive the stopping threshold τ, we first enumerate all possible subsets of $\mathbf{R}$. Then for each subset $\mathbf{R}'$, we derive the maximum possible join result value by joining the HRs of relations in $\mathbf{R}'$ with the CRs of relations in $\mathbf{R} - \mathbf{R}'$. The threshold τ is the maximum of all such values. We note that although FR/FR* bounding scheme is tight, its complexity grows exponentially with the number of relations involved [110]. Indeed, following Finger and Polyzotis [54], we mainly consider rank joins with a small number of relations.

In addition to the bounding scheme, the *accessing strategy* (which determines which relation to explore next) may also affect the performance of the rank join algorithm. For example, a simple accessing strategy such as *round-robin* often results in accessing more tuples than necessary. More efficient accessing strategies include the *corner-bound-adaptive* strategy [62] for binary, single value-attribute rank join and the *potential adaptive* strategy [54] for multi-way, multiple value-attribute rank join.

As shown in the introduction, there are many situations where it is very natural to have aggregation constraints along with rank join. While previous work on rank join algorithms has devoted much effort to optimizing the bounding scheme and accessing strategy, little work has been done on opportunities for improving runtime efficiency by using constraints that may be present in a query.

One way to handle aggregation constraints in the standard rank join algorithm is by *post-filtering* each join result using the aggregation constraints. It can be shown that an algorithm based on post-filtering remains instance optimal. However, as we will demonstrate in the next section, this naïve algorithm misses many optimization opportunities by not taking full advantage of the properties of the aggregation constraints, and, as we will show in Sec. 3.6, can have poor empirical performance as a result. This observation coincides with recent findings that instance optimal algorithms are not always computationally the most efficient [54].

3.3.2 Other Related Work

As described in the introduction, rank join can be seen as a generalization of classic top-k querying where one searches for the top-k objects w.r.t. a number of criteria or features [53]. Ilyas et al. [64] discussed how to incorporate binary rank join operator into relational query engines. The query optimization framework used in [64] follows System R's dynamic programming-based approach, and in order to estimate the cost of the rank join operator, a novel probabilistic model is proposed. In [86], Li et al. extended [64] by providing a systematic algebraic support for relational ranking queries. Tsaparas et al. proposed in [117] a novel indexing structure for answering rank join queries. In this work, various tuple pruning techniques are studied to reduce the size of the index structure. In [94], Martinenghi et al. proposed a novel proximity rank join operator in which the join condition can be based on a nontrivial proximity score between different tuples. A more detailed survey of top-k query processing and rank join can be found in [63], We note that *no previous work on rank join has considered aggregation constraints.*

Our work is also closely related to recent efforts on *package recommendation* [19, 47, 100, 106, 120]. Though some of these works [47, 120] discuss finding high-quality packages under certain aggregation constraints such as budgetary constraints, none of them provide a systematic study of aggregation constraints. In [19], the authors propose a rank join-based algorithm for finding travel packages of a fixed schema, however, in this work, the authors do not consider aggregation constraints which can be very useful in practice.

3.4 Deterministic Algorithm

We begin by illustrating rank joins with aggregation constraints.

Example 1 *[Rank Join with Aggregation Constraints]* Consider two relations, `Museum` and `Restaurant`, each with three attributes, `Location`, `Cost` and `Rating`, where `Rating` is the value attribute and `Location` is the join attribute (see Fig. 3.1). Assume we are looking for the top-2 results subject to the aggregation constraint $SUM(\texttt{Cost}, true) \leq 20$. Under the corner bounding scheme and round-robin accessing strategy, the algorithm will stop after accessing 5 tuples in `Museum` and 4 tuples in `Restaurant`. Note that even though the joinable set $\{t_3, t_7\}$ has a high value, it is not a top-2 result because it does not satisfy the constraint.

Our motivation in this section is to develop efficient pruning techniques for computing rank joins with aggregation constraints fast. Thereto, we

Museum

	Location	Cost	Rating
t_1:	a	13.5	5
t_2:	a	15	5
t_3:	b	10	4.5
t_4:	a	15	4.5
t_5:	b	5	3.5

Restaurant

	Location	Cost	Rating
t_6:	c	50	4.5
t_7:	b	20	4.5
t_8:	b	10	4.5
t_9:	a	5	3
t_{10}:	a	10	3

Constraint

SUM(Cost,*true*) ≤ 20

Top-2 results

{ t_3, t_8 }

{ t_1, t_9 }

Figure 3.1: Post-filtering rank join with aggregation constraints.

first present a number of properties of aggregation constraints and show how these properties can be leveraged to prune seen tuples from the in-memory buffers. We then propose an efficient rank join algorithm supporting aggregation constraints that minimizes the number of tuples that are kept in the in-memory buffers, which in turn helps cut down on useless joins.

3.4.1 Properties of Aggregation Constraints

Let $pc ::= \text{AGG}(A, p)\ \theta\ \lambda$ be a primitive aggregation constraint (PAC). In order to use pc to prune seen tuples, we first study properties of the various forms of pc, i.e., for $\text{AGG} \in \{MIN, MAX, SUM\}$ and $\theta \in \{\leq, \geq, =\}$.

First consider the cases when AGG is MIN and θ is $\geq$, or AGG is MAX and θ is $\leq$. These cases are the simplest because pc need only be evaluated on each seen tuple individually rather than on a full joinable set. When accessing a new tuple t, if $A \in schema(t)$ and t satisfies p, we can simply check whether $t.A\ \theta\ \lambda$ holds. If not, we can prune t from future consideration as pc will not be satisfied by any join result including t. After this filtering process, all join results obtained by the algorithm must satisfy the constraint pc. We name this property the *direct-pruning* property.

When $\text{AGG} \in \{MAX, SUM\}$ and θ is $\geq$, or AGG is MIN and θ is $\leq$, the corresponding aggregation constraint pc is *monotone*.

Definition 2 (**Monotone Aggregation Constraint**) A PAC pc is monotone if $\forall t \in R$, $\forall s \in \mathbf{JS}$, where $R \notin Rel(s)$: if $\{t\} \models pc$ and $\{t\} \bowtie s \neq \emptyset$, then $\{t\} \bowtie s \models pc$.

For the case when AGG is SUM and θ is $\leq$, the PAC is *anti-monotone*. This means that if a tuple t does not satisfy pc, no join result of t with any partial joinable set will satisfy PAC either.[8]

[8]The cases where AGG is MAX and θ is $\leq$ and AGG is MIN and θ is $\geq$ are also anti-monotone, but they can be handled using direct pruning, discussed above.

Definition 3 (**Anti-Monotone Aggregation Constraint**) A PAC pc is anti-monotone if $\forall t \in R$, $\forall s \in \mathbf{JS}$, where $R \notin Rel(s)$: if $\{t\} \not\models pc$, then either $\{t\} \bowtie s = \emptyset$ or $\{t\} \bowtie s \not\models pc$.

As a special case, when AGG $\in \{MIN, MAX\}$ and θ is $=$, we can efficiently check whether all the joinable sets considered satisfy $AGG(A, p) \geq \lambda$ and $AGG(A, p) \leq \lambda$, using a combination of direct pruning and anti-monotonicity pruning.

Finally, for the case when AGG is SUM and θ is $=$, it is easy to see that pc is neither monotone nor anti-monotone. However as discussed in [97], pc can be treated as a special constraint in which the evaluation value of a tuple t on pc, $Eval_{pc}(\{t\})$, determines whether or not the anti-monotonic property holds. For example, let $pc ::= SUM(A, p) = \lambda$ and t be a tuple. If $t \models p$ and $\{t\} \not\models pc$, then either $Eval_{pc}(\{t\}) > \lambda$ or $Eval_{pc}(\{t\}) < \lambda$. In the first case, the anti-monotonic property still holds. We call this conditional anti-monotonic property *c-anti-monotone*. Table 3.1 summarizes these properties.

AGG\θ	$\leq$	$\geq$	$=$
MIN	monotone	direct-pruning	monotone after pruning
MAX	direct-pruning	monotone	monotone after pruning
SUM	anti-monotone	monotone	c-anti-monotone

Table 3.1: Properties of primitive aggregation constraints.

Properties like direct-pruning, anti-monotonicity and c-anti-monotonicity can be used to filter out tuples that do not need to be maintained in buffers. However, this pruning considers each tuple individually. In the next subsection, we develop techniques for determining when tuples are "dominated" by other tuples. This helps in pruning even more tuples.

3.4.2 Subsumption-based Pruning

Consider Example 1 again. After accessing four tuples from `Museum` and three tuples from `Restaurant` (see Figure 3.2), the algorithm cannot stop as it has found only one join result. Furthermore we cannot prune any seen `Museum` tuple since each satisfies the constraint. However, it turns out that we can safely prune t_4 (from `Museum`) because, for any unseen tuple t' from `Restaurant`, if t' could join with t_4 to become a top-2 result, t' could also join with t_1 and t_2 without violating the constraint and giving a larger score.

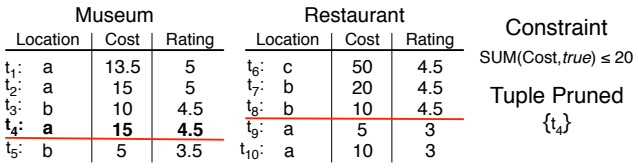


Figure 3.2: Tuple pruning using aggregation constraints.

The above example shows that, in addition to the pruning that is directly induced by the properties of the aggregation constraints, we can also prune a tuple by comparing it to other seen tuples from the same relation. As we discuss in the next section, this pruning can help to reduce the number of in-memory join operations. The key intuition behind pruning a tuple $t \in R$ in this way is the following. Call a join result *feasible* if it satisfies all applicable aggregation constraints. To prune a seen tuple $t \in R$, we should establish that whenever t joins with tuples (joinable set) s from other relations to produce a feasible join result ρ, then there is another seen tuple $t' \in R$ that joins with s and produces a feasible result whose overall value is more than that of ρ. Whenever this condition holds for a seen tuple $t \in R$, we say t' *beats* t. If there are k distinct seen tuples $t'_1, ..., t'_k \in R$ such that each of them beats t, then we call t *beaten*. Clearly, a seen tuple that is beaten is useless and can be safely pruned. In the rest of this section, we establish necessary and sufficient conditions for detecting (and pruning) beaten tuples among those seen. Thereto, we need the following notion of tuple domination.

Definition 4 *(pc-***Dominance Relationship***)* Given two tuples $t_1, t_2 \in R$, t_1 *pc*-dominates t_2, denoted $t_1 \succeq_{pc} t_2$, if for all $s \in \mathbf{JS}$, s.t. $R \notin Rel(s)$, $\{t_2\} \bowtie s \neq \emptyset$ and $\{t_2\} \bowtie s \models pc$, we have $\{t_1\} \bowtie s \neq \emptyset$ and $\{t_1\} \bowtie s \models pc$.

Intuitively, a tuple t_1 *pc*-dominates another tuple t_2 from the same relation (for some given PAC *pc*) if for any possible partial joinable set s which can join with t_2 and satisfy *pc*, s can also join with t_1 without violating *pc*.

Note that the *pc*-dominance relationship defines a *quasi-order* over tuples from the same relation since it is reflexive and transitive but not anti-symmetric: there may exist two tuples t_1 and t_2, such that $t_1 \succeq_{pc} t_2$, $t_2 \succeq_{pc} t_1$, but $t_1 \neq t_2$.

For the various PACs studied in this chapter, we can characterize precisely when the *pc*-dominance relationship holds between tuples. The conditions depend on the type of the PAC.

First of all, consider a monotone PAC *pc*. Because *pc* is monotone, given a tuple t, if $t \not\models pc$, then the join result of t with any other joinable set

will also satisfy pc, as long as t is joinable with s. So we have the following lemma in the case where $pc ::= SUM(A, p) \geq \lambda$.

Lemma 1 Let $pc ::= SUM\ (A, p) \geq \lambda$ be a primitive aggregation constraint and t_1, t_2 be tuples in R. Then $t_1 \succeq_{pc} t_2$ iff $t_1.J = t_2.J$ and either $t_1 \models pc$ or $t_1.A \geq t_2.A$.

We can prove a similar lemma for the other monotonic aggregation constraints, where AGG is MIN and $\theta \in \{\leq, =\}$, or AGG is MAX and $\theta \in \{\geq, =\}$.

Lemma 2 Let pc be a primitive aggregation constraint in which AGG is MIN and $\theta \in \{\leq, =\}$, or AGG is MAX and $\theta \in \{\geq, =\}$. Given two tuples t_1 and t_2, $t_1 \succeq_{pc} t_2$ iff $t_1.J = t_2.J$ and either $t_1 \models pc$ or $t_2 \not\models pc$.

For the anti-monotone constraint $pc ::= SUM(A, p) \leq \lambda$, we can directly prune any tuple t such that $t \not\models pc$; however, for tuples that do satisfy pc, we have the following lemma.

Lemma 3 Let $pc ::= SUM\ (A, p) \leq \lambda$ be a primitive aggregation constraint and t_1, t_2 be two tuples such that $t_1 \models pc$ and $t_2 \models pc$. Then $t_1 \succeq_{pc} t_2$ iff $t_1.J = t_2.J$ and $t_1.A \leq t_2.A$.

Similarly, for the c-anti-monotone constraint $SUM(A, p) = \lambda$, we have the following lemma.

Lemma 4 Let $pc ::= SUM\ (A, p) = \lambda$ be a primitive aggregation constraint and t_1, t_2 be two tuples such that $t_1 \models SUM(A, p) \leq \lambda$ and $t_2 \models SUM(A, p) \leq \lambda$. Then $t_1 \succeq_{pc} t_2$ iff $t_1.J = t_2.J$ and $t_1.A = t_2.A$.

Given the pc-dominance relationship for each individual aggregation constraint, we can now define an overall subsumption relationship between two tuples.

Definition 5 *(***Tuple Subsumption***)*Let t_1, t_2 be seen tuples in R and $ac ::= pc_1 \wedge \cdots \wedge pc_m$ be an aggregation constraint. We say that t_1 *subsumes* t_2, denoted $t_1 \succeq t_2$, if $t_1.J = t_2.J$, $t_1.V \geq t_2.V$ and, for all $pc \in \{pc_1, \ldots, pc_m\}$, $t_1 \succeq_{pc} t_2$[9].

[9]Let r_k be the k^{th} join result in RS_k^{ac}. To handle the case where all join results which have the same score as r_k need to be returned, we can change the condition $t_1.V \geq t_2.V$ in Definition 5 to $t_1.V > t_2.V$ and report all such results.

Recall, the main goal of this section is to recognize and prune beaten tuples. The next theorem says how this can be done.

Theorem 1 Given an RJAC problem instance $I = \{\mathbf{R}, S, jc, ac, k\}$, let T be the set of seen tuples from relation R_i. Tuple $t \in T$ is beaten iff t is subsumed by at least k other tuples in T.

3.4.3 Efficient Algorithm for Top-k RJAC

Given an instance of RJAC, $I = (\mathbf{R}, S, jc, ac, k)$, our algorithm kRJAC (see Algorithm 2) follows the standard rank join template [62, 110] as described in Section 3.3.1. However, it utilizes the pruning techniques developed in Sec. 3.4.1 and 3.4.2 to leverage the power of aggregation constraints.

Algorithm 2: kRJAC($\mathbf{R}$, S, jc, ac, k)

```
1 τ ← ∞;
2 O ← Join result buffer;
3 while |O| < k ∨ v(O.kthResult) < τ do
4     i ← ChooseInput();
5     t_i ← R_i.next();
6     if Promising(t_i, ac)          /* (c-)Anti-monotone pruning */
7         if ¬(Prune(t_i,HR_i,ac,k))      /* Subsumption pruning */
8             ConstrainedJoin(t_i, HR, ac, O);
9         τ ← UpdateBound(t_i, HR, ac);
```

Below, we first explore the pruning opportunities in the kRJAC algorithm using aggregation constraints (lines 6–8), and then discuss how the presence of aggregation constraints can affect the accessing strategy (line 4) and the stopping criterion (line 9).

Optimizing In-Memory Join Processing

First of all, to leverage the (c-)anti-monotonicity property of the aggregation constraints, in line 6 of Algorithm 2, whenever a new tuple t_i is retrieved from relation R_i, we invoke the procedure *Promising* (see Algorithm 3) which prunes tuples that do not satisfy the corresponding aggregation constraint.

Let $HR = \{HR_1, \ldots, HR_n\}$ be the in-memory buffers for all seen tuples from each relation. Similar to previous work [62], in line 8 of Algorithm 2, when a new tuple t_i is seen from R_i, we perform an in-memory hash join of

t_i with seen tuples from all HR_j, $j \neq i$. The idea of this hash join process is that we break each HR_i into hash buckets based on the join attribute value. Note that for an RJAC problem instance in which no join condition is present or $jc = true$, all seen tuples from the same relation will be put into the same hash bucket.

Algorithm 4 shows the pseudo-code for the aggregate-constrained hash join process. We first locate all relevant hash buckets from each relation (lines 1–2), then join these buckets together and finally check, for each join result found, whether it satisfies the aggregation constraints or not (lines 3–5).

Algorithm 3: Promising(t_i, ac)

```
1 foreach pc in ac do
2     if pc ::= MIN(A,p) ≥ (=) λ  return Eval_pc({t_i}) ≥ λ ;
3     else if pc ::= MAX(A,p) ≤ (=) λ  return Eval_pc({t_i}) ≤ λ ;
4     else if (pc ::= SUM(A,p) ≤ λ) ∨ (pc ::= SUM(A,p) = λ)
5         return Eval_pc({t_i}) > λ;
6 return true
```

Algorithm 4: ConstrainedJoin(t_i, HR, ac, O)

```
1 for j = 1,...,i−1,i+1,...,n do
2     B_j = LocateHashBuckets(t_i.J, HR);
3 foreach s ∈ B_1 ⋈ ··· ⋈ B_{i−1} ⋈ {t_i} ⋈ B_{i+1} ⋈ ··· B_n do
4     if s ⊨ ac and v(s) > v(O.kthResult)
5         Replace O.kthResult with s.
```

One important observation about this hash join process is that the worst case complexity for each iteration is $O(|HR_1| \times \cdots \times |HR_{i-1}| \times |HR_{i+1}| \times \cdots \times |HR_n|)$, which can result in a huge performance penalty if we leave all seen tuples in the corresponding buffers. As a result, it is crucial to *minimize the number of tuples retained in the HR's.* Next we will show how our subsumption-based pruning, as discussed in Section 3.4.2, can be used to remove tuples safely from HR.

Consider a hash bucket B in HR_i and a newly seen tuple t_i. We assume every tuple in a bucket B has the same join attribute value, so according to Theorem 1, if we find that there are at least k tuples in B which subsume

t_i, we *no longer need to place* t_i *in* HR_i. This is because we already have at least k tuples in B that are at least as good as t_i. We call this pruning *subsumption-based pruning* (SP). Furthermore, t_i does not need to be joined with HR_j, $j \neq i$, as shown in line 7 of Algorithm 2. We will show in Sec.3.6 that this subsumption-based pruning can significantly improve the performance of the kRJAC algorithm.

Algorithms 5 and 6 give the pseudo-code for the subsumption-based pruning process. We maintain for each tuple t a count $t.scount$ of the number of seen tuples that subsume t. Note that, although in the pseudo-code we invoke the Subsume procedure twice for each tuple t in the current hash bucket B, the two invocations can in fact be merged into one in the implementation.

Algorithm 5: Prune(t_i, HR_i, ac, k)

```
1 B ← LocateBucket(t_i, HR_i);
2 foreach t ∈ B do
3     if Subsume(t, t_i, ac) t_i.scount ← t_i.scount + 1;
4     if Subsume(t_i, t, ac)
5         t.scount ← t.scount + 1;
6         if t.scount ≥ k Remove t from B;
7 return t_i.scount ≥ k;
```

So given the basic subsumption-based pruning algorithm as presented in Algorithm 5, a natural question to ask is whether can we prune more tuples from the buffer? The answer is "yes". Assume we are looking for the top-k join results. As we consume more tuples from the underlying relations, the value of the stopping threshold τ may continue to decrease, which means some join results in the output buffer O may have a value larger than τ. These join results are guaranteed to be among the top-k and can be output.

Now suppose that the top k' results, for some $k' < k$, have been found so far. Then it is clear that we need only look for the next top $k - k'$ results among the remaining tuples. So when applying our subsumption-based pruning, we could revise k to $k - k'$, i.e., in a hash bucket B of a buffer HR_i, if a new tuple t_i is subsumed by $k - k'$ other tuples in HR_i, we can safely prune t_i from that buffer. We call this optimization *adaptive subsumption-based pruning* (ASP).

Consider the example of Figure 3.3. After retrieving four tuples in the `Museum` relation and three tuples in the `Restaurant` relation, we find one

Algorithm 6: Subsume(t_1, t_2, ac)

```
1  if t1.V < t2.V return false;
2  Dominate ← true;
3  foreach pc in ac do
4      switch pc do
5          case MIN(A,p) ≤ (=)λ and MAX(A,p) ≥ (=)λ
6              Dominate = Dominate ∧ (t1 ⊨ pc or t2 ⊭ pc);
7          case SUM(A,p) ≥ λ
8              Dominate = Dominate ∧ (t1 ⊨ pc or t1.A ≥ t2.A);
9          case SUM(A,p) ≤ λ
10             Dominate = Dominate ∧ (t1.A ≥ t2.A);
11         case SUM(A,p) = λ          /* {t1},{t2} ⊨ SUM(A,p) ≤ λ */
12             Dominate = Dominate ∧ (t1.A = t2.A);
13 return Dominate;
```

joinable set $\{t_3, t_8\}$ which is guaranteed to be the top-1 result, and we have pruned t_4. Using adaptive subsumption-based pruning, we can now also prune t_2 as it is subsumed by t_1.

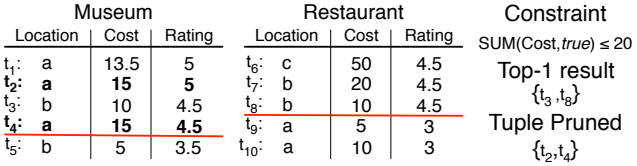


Figure 3.3: Adaptive subsumption-based pruning.

If adaptive subsumption-based pruning is utilized, from the correctness and completeness proof of Theorem 1, we can derive the following corollary.

Corollary 1 At the end of each iteration of the kRJAC algorithm, the number of accessed tuples retained in memory for each relation is minimal.

In the worst case, the overhead of the rank join algorithm using subsumption-based pruning compared to one which does not perform any pruning (both algorithms will stop at the same depth d) will be $O(d^2 \cdot c_{dom})$, where c_{dom} is the time for one subsumption test. This worst case situation will happen when no tuples seen from a relation subsume any other tuples. However, as

we show in Section 3.6, this seldom happens, and often d is very small after our pruning process.

Bounding Scheme and Accessing Strategy

When rank join involves more than two relations, the corner-bounding strategy should be replaced by a bounding strategy based on cover sets [54, 110]. As described in Section 3.3, for the optimal bounding scheme, to derive the stopping threshold τ, we need to consider each subset $\mathbf{R}'$ of $\mathbf{R}$, and join the HRs of relations in $\mathbf{R}'$ with the CRs of relations in $\mathbf{R} - \mathbf{R}'$. Because the cover set CR_i of each relation R_i considers only the value of an unseen item, data points in CR_i can be joined with any other tuple from a tuple buffer HR_j, where $i \neq j$. So the presence of aggregation constraints does not affect the operations in the bounding scheme that are related to the cover set, which means when joining CRs of $\mathbf{R} - \mathbf{R}'$, and when joining the join results of CRs of $\mathbf{R} - \mathbf{R}'$ and join results of HRs of $\mathbf{R}'$, we don't need to consider aggregation constraints. However, when joining HRs of $\mathbf{R}'$, in order for the derived bound to be tight, we need to make sure that each partial join result satisfies the aggregation constraints.

Similarly, for the accessing strategy that decides which relation to access a tuple from next, because the potential value of each relation is determined by the bounding scheme as discussed in [54, 62, 110], the existing accessing strategy can be directly used by taking the modified bounding scheme as described above into account.

3.5 Probabilistic Algorithm

Our kRJAC algorithm in Section 3.4 returns the exact top-k results. However, similar to the standard NRA [53] algorithm, this deterministic approach may be conservative in terms of its stopping criterion, which means that it still needs to access many tuples even though many of them will be eventually pruned. Theobald et al. [113] first investigated this problem and proposed a probabilistic NRA algorithm; however, their algorithm and analysis cannot be directly used to handle rank join (with aggregation constraints). In the rest of this section, we will describe a probabilistic algorithm, based on the framework of [113], which accesses far fewer tuples while guaranteeing the quality of the results returned.

Let $I = (\mathbf{R}, S, jc, ac, k)$ be a RJAC problem instance, where $\mathbf{R} = \{R_1, \ldots, R_n\}$. The main problem we need to solve is, at any stage of the algorithm, to estimate the probability that an unseen tuple can achieve a value better than

the value of the k^{th} tuple in the top-k buffer. This probability will clearly depend on the selectivity of the join condition jc and on the aggregation constraint ac. We assume the join selectivity of jc over $\mathbf{R}$ can be estimated using some existing techniques such as adaptive sampling [90]. We denote the resulting join selectivity by $\delta_{jc}(\mathbf{R})$, which is defined as the estimated number of join results divided by the size of the cartesian product of all relations in $\mathbf{R}$. Given a set $s = \{t_1, \ldots, t_n\}$ of n tuples, where $t_i \in R_i$, by making the uniform distribution assumption, we set the probability P_{jc} of s satisfying jc as $\delta_{jc}(\mathbf{R})$. Similarly, considering each primitive aggregation constraint pc in ac, we can also estimate the probability P_{pc} of s satisfying pc as the selectivity of pc over $\mathbf{R}$, denoted as $\delta_{pc}(\mathbf{R})$. We discuss in Sec. 3.5.1 how $\delta_{pc}(\mathbf{R})$ can be estimated under common data distribution assumptions. The probability P_{ac} of s satisfying ac can then be estimated as $P_{ac} = \prod_{pc \in ac} P_{pc}$.

Given a set of tuples $s = \{t_1, \ldots, t_n\}$, $t_i \in R_i$, assuming the join condition and the aggregation constraints are independent, we can estimate the probability of s satisfying the join condition jc and the aggregation constraints ac as $P_{jc \wedge ac} = P_{jc} \times P_{ac}$.

After some fixed number of iterations of the kRJAC algorithm, let the value of the k^{th} best join result in the output buffer O be min_k. We can estimate the probability $P_{>min_k}(R_i)$ that an unseen tuple t_i from R_i can achieve a better result than min_k. Suppose the current maximum value for an unseen item in R_i is $\overline{v_i}$. To estimate $P_{>min_k}(R_i)$, similarly to [113], we assume a histogram H_j^V for the value attribute V of each relation $R_j \in \mathbf{R}$ is available. Then using the histograms we can estimate the number N_i of tuple sets $\{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\}$, $t_j \in R_j$ s.t. $\overline{v_i} + \sum_{j \in \{1 \ldots n\} - \{i\}} v(t_j) > min_k$. We omit the obvious detail. Then the probability that t_i can join with any of these N_i tuple sets to become one of the top-k results can be estimated as $P_{>min_k}(R_i) = 1 - (1 - P_{jc \wedge ac})^{N_i}$.

Given a user specified threshold ϵ, we can stop our kRJAC algorithm when $\forall i \in \{1, \ldots, n\}$, $P_{>min_k}(R_i) \leq \epsilon$.

3.5.1 Estimating Constraint Selectivity

Given a PAC $pc ::= \mathrm{AGG}(A, p)\, \theta\, \lambda$, and n relations $R_1, \ldots, R_n$, to simplify the analysis, we assume $p = true$ and that attribute values of different relations are independent.

Consider an example of the binary RJAC problem: given a set $s = \{t_1, t_2\}$, with $t_1 \in R_1$, $t_2 \in R_2$. For the aggregation constraint $pc ::= SUM(A, true) \leq \lambda$, it is clear from Figure 3.4(a) that s can satisfy pc only

when $t_1.A$ and $t_2.A$ fall into the gray region. We call this gray region the *valid region* for pc, denoted VR_{pc}. Similarly Figure 3.4(b) illustrates the valid region for the constraint $pc ::= MIN(A, true) \leq \lambda$.

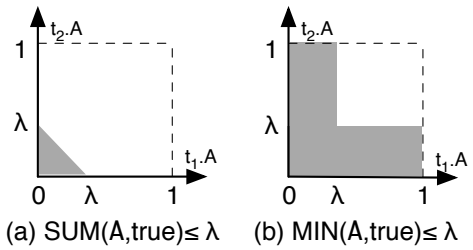


Figure 3.4: Selectivity of aggregation constraints.

Based on the valid region VR_{pc} for pc, we can estimate the selectivity of pc by calculating the probability of a tuple set s falling inside VR_{pc}.

Given a set $s = \{t_1, \ldots, t_n\}$ of n tuples, $t_i \in R_i$, and given a PAC $pc ::= \text{AGG}(A, true)\,\theta\,\lambda$, if we assume $t_1.A, \ldots, t_n.A$ are n independent random variables following a uniform distribution, we can calculate the closed formula for the probability $P(VR_{pc})$ of $t_1.A, \ldots, t_n.A$ falling inside VR_{pc} as follows:

- If $pc ::= SUM(A, true) \leq \lambda$: $P(VR_{pc}) = \frac{\lambda^n}{n!}$.
- If $pc ::= MIN(A, true) \leq \lambda$: $P(VR_{pc}) = 1 - (1 - \lambda)^n$.

These facts are easily verified. Because of symmetry, for $pc ::= SUM(A, true) \geq \lambda$ and $pc ::= MAX(A, true) \geq \lambda$, the corresponding probabilities are very similar to $pc ::= SUM(A, true) \leq \lambda$ and $pc ::= MIN(A, true) \leq \lambda$ respectively: we only need to replace λ by $1 - \lambda$ in the corresponding formulas. And for a PAC pc where θ is $=$, note that the probability is 0 under continuous distributions, so in practice, we will set these probabilities to a small constant which is estimated by sampling the database.

Similarly, if we assume that each $t_i.A$ follows other distributions such as exponential distribution, similar formulas can be derived.

3.6 Experiments

In this section, we study the performance of our proposed algorithms based on two synthetic datasets. The goals of our experiments are to study: (i)

the performance of various pruning techniques, (ii) the performance of the probabilistic method, and (iii) the result quality of probabilistic method. All experiments were done on a Intel Core 2 Duo machine with 4GB RAM and 250GB SCSI hard disk. All code is in C++ and compiled using GCC 4.2.

We call the synthetic datasets we generated the *uniform* dataset and the *exponential* dataset. For both datasets, the join selectivity between two relations is fixed at 0.01 by randomly selecting the join attribute value from a set of 100 predefined values. The value and other attributes are set as follows. For the uniform dataset, the value of each attribute follows a uniform distribution within the range [0,1]; for the exponential dataset, the value of each attribute follows an exponential distribution with mean 0.5. Note that in order to ensure values from the exponential distribution fall inside the range [0,1], we first uniformly pick 1000000 values from [0,1], and then resample these values following the exponential distribution. Values of each attribute are independently selected.

We implemented four algorithms: (a) the post-filtering based rank join algorithm (Post Filtering); (b) the deterministic algorithm with subsumption based pruning (SubS-Pruning); (c) the deterministic algorithm with adaptive subsumption based pruning (Adaptive SubS-Pruning); (d) the probabilistic algorithm with subsumption based pruning.

3.6.1 Efficiency Study

We first compare the algorithms in a binary RJAC setting. As can be seen from Figure 3.5, subsumption based pruning works very well for monotonic constraints. One interesting observation from Figure 3.5(d) is that, adaptive subsumption based pruning does not prune significantly more tuples than non-adaptive subsumption based pruning. By inspecting the dataset, we found out this is because there are k tuples which subsume every other tuple, so the adaptive pruning strategy has no effect in this case.

Figure 3.6 shows another example of one aggregation constraint, $SUM(A, true) \leq \lambda$, under the selectivity of 10^{-5}. As discussed in previous sections, such a constraint can result in both anti-monotonicity based pruning and subsumption based pruning. However, as can be seen from Figure 3.6, the anti-monotonicity based pruning can be very powerful which, in turn, renders the subsumption based pruning less effective.

We also tested our algorithms in settings where we have binary RJAC and multiple aggregation constraints (see Figure 3.7). For the case of $SUM(A, true) \geq \lambda$ and $SUM(B, true) \leq \lambda$ and overall selectivity is 10^{-5} ((a) and (b)), be-

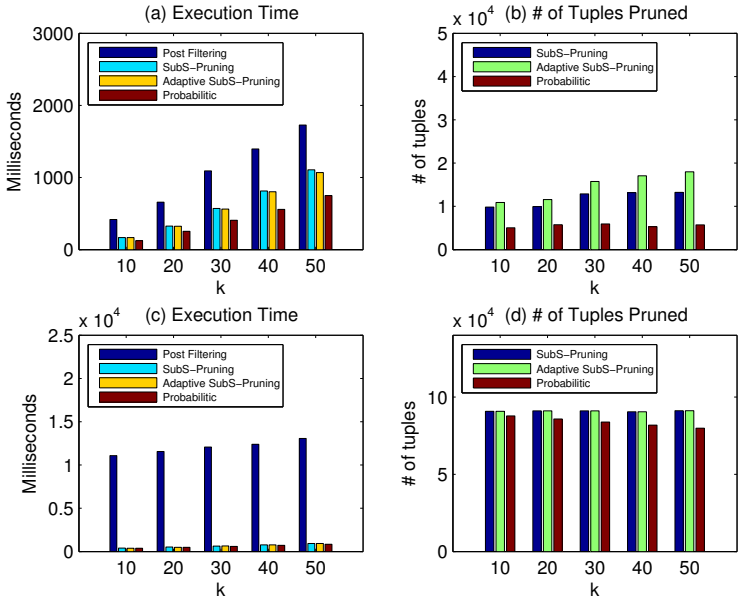


Figure 3.5: Uniform dataset: (a), (b) $SUM(A, true) \geq \lambda$, selectivity 10^{-5} ; (c), (d) $MIN(A, true) \leq \lambda$, selectivity 10^{-5}.

cause of the presence of an anti-monotone constraint, many tuples can be pruned so the subsumption based algorithm outperforms the post-filtering algorithm. However, as can be seen from Figure 3.7(c) and (d), when the selectivity of aggregation constraints is very high and no anti-monotonic or direct-pruning aggregation constraint is present, the overhead of subsumption testing causes the execution time of the subsumption based algorithms almost to match that of the post-filtering based algorithm. As future work, we would like to study cost-based optimization techniques which can be used to help decide which strategy should be used.

3.6.2 Probabilistic Algorithm

Similar to previous work on a probabilistic NRA algorithm, Figures 3.5, 3.6 and 3.7 show that our probabilistic algorithm will stop earlier than the deterministic and post-filtering based algorithms. In most experiments, the probabilistic algorithm accesses far fewer tuples from the underlying database than the other algorithms. We note that this property can be very important for scenarios where tuples are retrieved using web services [53],

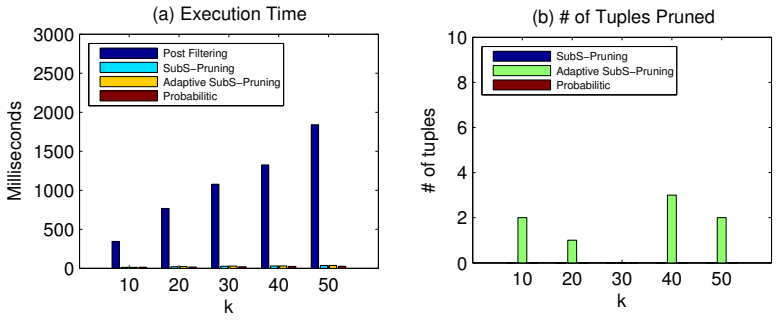


Figure 3.6: Uniform dataset, $SUM(A, true) \leq \lambda$, selectivity 10^{-5}.

for example, as a monetary cost might be associated with each access and the latency of retrieving the next tuple might be very high.

In terms of the quality of results returned, as Figure 3.8 shows for binary RJAC with several different aggregation constraints, the value of the join results returned by the probabilistic algorithm at each position k is very close to the exact solution. The percentage of value difference at each position k is calculated as $\frac{v(s_k)-v(s'_k)}{v(s_k)}$, where s_k is the exact k^{th} result and s'_k is the k^{th} result returned by the probabilistic algorithm.

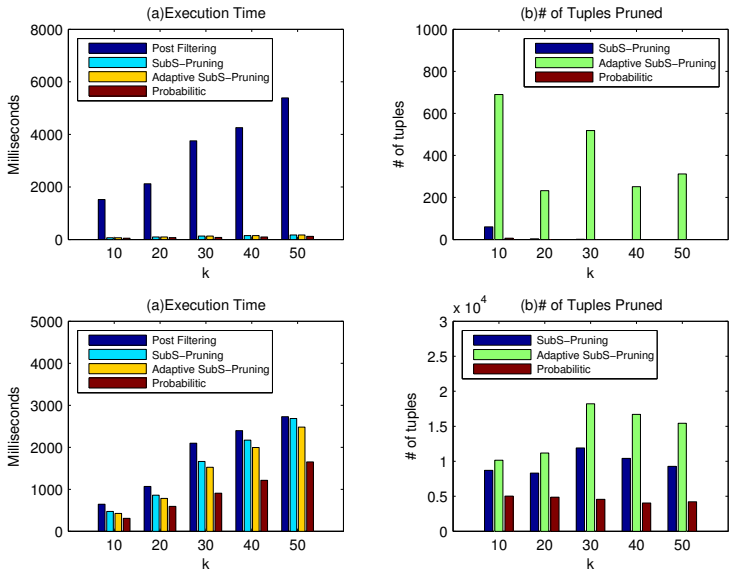


Figure 3.7: Uniform dataset: (a), (b) $SUM(A, true) \geq \lambda$, $SUM(B, true) \leq \lambda$, overall selectivity 10^{-5} ; (c), (d) $SUM(A, true) \geq \lambda$, $SUM(B, true) \geq \lambda$, overall selectivity 10^{-5}.

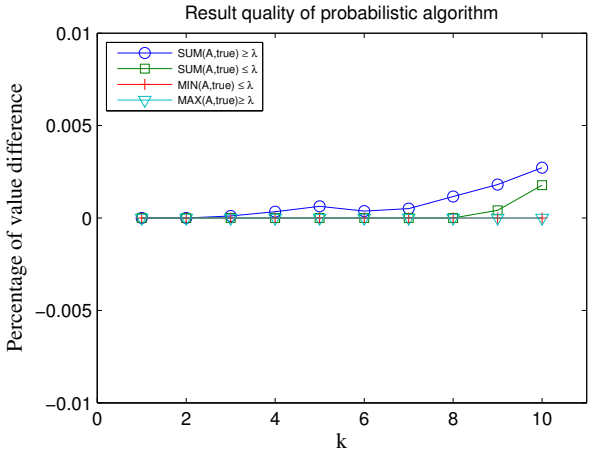


Figure 3.8: Quality of the probabilistic algorithm.

Chapter 4

Composite Recommendation with Hard Constraints: Flexible Package Schema

4.1 Introduction

The limitation of the solution proposed in the previous chapter is that it can only handle fixed package schema. In this chapter we consider the more general problem of composite recommendations with flexible package schema, where each recommendation comprises a set of items which can have different size/schema. And though we are dealing with budget constraints in the main part of this work, we note that other constraints can be also handled efficiently using the prunings proposed in [122], and post-filtering techniques as discussed in Section 4.5.

Consider each item is associated with both a value (rating or score) and a cost, and the user specifies a maximum total cost (budget) for any recommended set of items. Our composite recommender system consists of one or more recommender systems focusing on different domains. These component RecSys serve (i.e., recommend) top items in non-increasing order of their value (explicit or predicted ratings). In addition, our composite system has access to information sources (which could be databases or web services) which provide the cost associated with each item.

In our setting, the problem of deciding whether there is a recommendation (package) whose value exceeds a given threshold is NP-complete as it models the Knapsack problem [71]. Because of this, and the fact that we expect the component recommender systems to provide ratings for large numbers of items and access to these ratings can be relatively expensive[10], we devise approximation algorithms for generating the top-k packages as recommendations.

Other researchers have considered complex or composite recommenda-

[10]Especially when the ratings need to be predicted.

tions. CARD [32] and FlexRecs [74] are comprehensive frameworks in which users can specify their recommendation preferences using relational query languages extended with additional features or operators. In contrast, we are concerned with developing efficient algorithms for combining recommendations from RecSys that provide only ratings for items. Closer to our work is [19] which is concerned with finding packages of entities, such as holiday packages, where the entities are associated in some way. However, their packages are of fixed size, whereas we allow packages of variable size/schema. CourseRank [99, 101, 102] is a system for providing course recommendations to students, based on the ratings given to courses by past students and subject to the constraints of degree requirements. While we do not capture all CourseRank constraints, in our framework we have item costs and user budgets—essential features of the application areas we consider for deployment of our system—which are not captured by CourseRank. Similarly, item costs and user budgets are not considered for the problem of team formation in [78].

In this chapter, we restrict attention to the problem of recommending packages when there is just one component RecSys and no compatibility constraint is imposed. The problem remains intractable and still warrants approximation algorithms. We discuss in Section 4.5 how to extend our algorithms when multiple component RecSys and compatibility constraints are present.

Following are the contributions of this chapter:

- We propose a novel architecture for doing composite recommendation (Section 4.2).
- We propose a 2-approximation algorithm that is instance optimal [53] with an optimality ratio of one. This means that any other 2-approximation algorithm, that can only access items in non-increasing order of their value, must access at least as many items as our algorithm (Section 4.3.4.3.1).
- We further develop a greedy algorithm for returning top-k composite recommendations, which is much more efficient, guaranteed to return a 2-approximation, but is no longer guaranteed to be instance optimal (Section 4.3.4.3.2).
- We study how histograms can be leveraged to improve the performance of the algorithms (Section 4.3.4.3.3).

- We subject our algorithms to thorough empirical analysis using two real datasets. Our findings confirm that our algorithms always produce recommendations that are 2-approximations, with many of them being close to optimal. And the greedy algorithm is always significantly faster than the other two algorithms, while the greedy and instance optimal algorithms usually access substantially fewer items than the optimal algorithm. Finally, we show how histograms can further improve the empirical performance of the proposed algorithms (Section 4.4).

4.2 Architecture and Problem

4.2.1 System Architecture

In a traditional RecSys, users rate items based on their personal experience, and these ratings are used by the system to predict ratings for items not rated by an active user. The predicted ratings can be used to give the user a ranked recommendation (item) list.

As shown in Figure 4.1, our composite recommendation system is composed of one or more component RecSys and has access to external sources that provide the cost of a given item. An external source can be a local database or a web service. For example, Amazon.com can be consulted for book prices. In terms of computation, we abstract each RecSys as a system which serves items in non-increasing order of their value (rating or score) upon request. In addition, the system includes a compatibility checker module, which checks whether a package satisfies compatibility constraints, if any. We assume the compatibility checker consults necessary information sources in order to verify compatibility.

The user interacts with the system by specifying a cost budget, an integer k, and optionally compatibility constraints on packages. The system finds the top-k packages of items with the highest total value such that each package has a total cost under budget and is compatible.

4.2.2 Problem Statement

Given a set N of items and U of users, an active user $u \in U$, and item $t \in N$, we denote by $v_u(t)$ the *value* of item t for user u. We denote the value as $v(t)$ when the active user is understood. A RecSys predicts $v(t)$ when it is not available, by using the active user's past behavior and possibly that of other similar users. For $t \in N$, we denote by $c(t)$ the *cost* of item t. Given

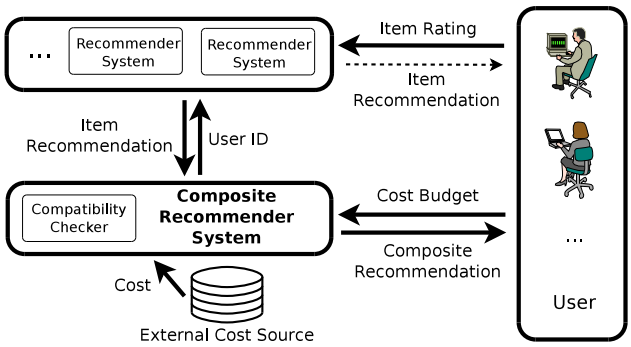


Figure 4.1: System Architecture

a set of items $R \subset N$, we define $c(R) = \Sigma_{t \in R}\, c(t)$ and $v(R) = \Sigma_{t \in R}\, v(t)$. Given a cost budget B, a set of items $P \subset N$ is called *feasible* if $c(P) \leq B$.

Definition 6 (Top-k Composite Recommendations) *Given an instance I of a composite recommendation system consisting of one component RecSys and an external information source, a cost budget B and an integer k, find the top-k packages $P_1, ..., P_k$ such that each P_i is feasible and among all feasible packages $P_1, ..., P_k$ have the k highest total values, i.e., $v(P) \leq v(P_i)$ for all feasible packages $P \notin \{P_1, ..., P_k\}$.*

When $k = 1$, the top-k composite recommendation problem (CompRec) can be viewed as a variation of the classical 0/1 knapsack problem [71]. Thus, even for the special case of top-1 composite recommendation, the decision problem "Is there a feasible package R which has value larger than a threshold β?" is NP-complete; and the complexity of the function problem of *finding* the maximum feasible package R is $\mathrm{FP}^{\mathrm{NP}}$-complete [98]. However, unlike the classical knapsack setting, our top-k composite recommendation problem has the restriction that items can be accessed only in non-increasing order of their value. Without loss of generality, we assume all items have cost smaller than the cost budget B.

Note that ratings of items from the component RecSys are retrieved using sorted access, while the cost of a given item is obtained via random access. Let c_s and c_r be the costs associated with these accesses. Then the total *access cost* of processing n items is $n \times (c_s + c_r)$. Notice that c_s and c_r can be large compared to the cost of in-memory operations: for both accesses information needs to be transmitted over the Internet, and for the sorted access, $v(t)$ may need to be computed. So, well-known algorithms for knapsack which need to access *all* items [71] may not be realistic. Thus,

an efficient algorithm for top-k CompRec should minimize the total access cost, i.e., it should minimize the number of items accessed and yet ensure the top-k packages are obtained.

It can be shown that if we have no background knowledge about the cost distribution of items, in the worst case, we must access all items to find top-k packages. In order to facilitate the pruning of item accesses, we thus assume that some background information about item costs is precomputed and maintained at the composite RecSys. The background cost information, which we denote generically by $\mathcal{BG}$, can be something as simple as a minimum item cost c_{min} (Section 4.3.4.3.1, 4.3.4.3.2) or a histogram collected from the external cost source (Section 4.3.4.3.3). This information can be materialized in our system and be refreshed regularly by re-querying the cost source.

Our composite recommendation problem can be considered as a special case of a resource-limited knapsack problem where, in addition to quality guarantee, the number of items to be accessed should also be minimized. So standard algorithms for knapsack, e.g., exact algorithms [71] and approximation algorithms [61, 118] may not be efficient as they always need to access the entire dataset. The only known variation of knapsack which deals with resource limitation is the Online Knapsack Problem [93]. However, for this problem, no access constraints are considered, only competitiveness in terms of quality is studied. And furthermore, no information about items can be inferred, which makes the problem significantly harder and difficult to approximate.

4.3 Composite Recommendations

In this section, we develop several approximation algorithms for top-1 CompRec, after which we extend them to handle top-k CompRec.

4.3.1 Instance Optimal Algorithms

As identified in Section 5.2, top-1 CompRec is a variation of the 0/1 knapsack problem where the underlying items can be accessed only in non-increasing order of their value (rating). Because of the huge potential size of the sets of items and the high cost of retrieving item information from the source, it is crucial for an algorithm to find high-quality solutions *while minimizing the number of items accessed*. Furthermore, as the 0/1 knapsack problem is NP-complete [71], we need to develop efficient approximation algorithms.

Top-1 Composite Recommendation

Given an instance I of top-1 CompRec, let $\mathcal{BG}$ denote the known background cost information and $S = \{t_1, ..., t_n\}$ be the set of items which have been accessed or seen so far.

Let $\bar{v}$ be the value of the first accessed item. Because items are accessed in non-increasing order of their value, $n \cdot \bar{v}$ is a trivial upperbound on the value that can be achieved by any knapsack solution for S.

For each $i \in \{1, ..., n\}$ and $v \in \{1, ..., n \cdot \bar{v}\}$, let $SS_{i,v}$ denote a subset of $\{t_1, ..., t_i\}$ whose total value is exactly v and whose total cost is minimized. Let $C(i, v)$ be the cost of $SS_{i,v}$ (where $C(i, v) = \infty$ if the corresponding $SS_{i,v}$ does not exist). Then it is well known from previous work [71, 118] that a pseudo-polynomial algorithm can be utilized to find the optimal knapsack solution for S by first calculating all $C(i, v)$ using the following recursive function, and then choosing the maximum value achievable by any subset $SS_{n,v}$ of which the total cost is bounded by budget B, i.e., $\max\{v \mid C(n, v) \leq B\}$.

$$C(i+1, v) = \begin{cases} \min\{C(i, v), c(t_{i+1}) + C(i, v - v(t_{i+1}))\} & \text{if } v(t_{i+1}) \leq v \\ C(i, v) & \text{otherwise} \end{cases} \tag{4.1}$$

Let the background cost information $\mathcal{BG}$ be given by c_{min}, the minimum cost of all items, let $v_{min} = \min_{t \in S} v(t)$ be the minimum value of all accessed items, and let OPT be the true optimal solution to the underlying top-1 CompRec instance I. We can find an upperbound V^* on the value $v(OPT)$ of the optimal solution using Algorithm 7: MaxValBound.

Algorithm 7: MaxValBound(S, C, B, $\mathcal{BG}$)

```
1 V* = ⌊B / c_min⌋ × v_min
2 for v ∈ {1, ..., n · v̄} do
3     if C(n, v) < B
4         V*=max{V*, v + ⌊(B − C(n,v)) / c_min⌋ * v_min}
5 return V*
```

Lemma 5 *Given S, C, B, $\mathcal{BG}$, we have: (1) the value V^* returned by MaxValBound is an upperbound on $v(OPT)$; (2) Value V^* is tight, in that*

there exists a possible unseen item configuration for which V^* is achievable by using a subset of accessed items and feasible unseen items[11].

Proof *Consider all valid possible top-1 CompRec instances, and let $OPT = S_{opt}$ denote the optimal solution with maximum value. We will prove both claims in the lemma at once. Consider an unseen item t^* with cost c_{min} and value v_{min}. Clearly, t^* is feasible. We can assume an unlimited supply of unseen items whose cost and value match those of t^*. We claim S_{opt} has no unseen item with cost or value not matching that of t^*. If it does, that item can be replaced with t^* while incurring no more cost and having no less total value. Let $S' = S_{opt} \cap S$, i.e., the subset of seen items in S_{opt}. Let $\mathbf{v} = v(S')$. Clearly, $1 \leq \mathbf{v} \leq n \times \bar{v}$ and $c(S') \geq C(n, \mathbf{v})$. $S_{opt} - S$ can contain at most $\lfloor \frac{B-c(S')}{c_{min}} \rfloor$ unseen items with cost and value matching those of t^*. Algorithm 1 does examine such a solution among the various candidates it considers. Hence the bound V^* computed by Algorithm 1 has the property that $V^* \geq v(S') + \lfloor \frac{B-c(S')}{c_{min}} \rfloor \times v_{min}$, proving (1). To see (2), since $v(OPT) = v(S_{opt})$ corresponds to the highest possible value of the optimal package over all possible configurations, we have $V^* \leq OPT$, so $V^* = OPT$, and V^* is achievable by an instance which is composed of S' and $\lfloor \frac{B-c(S')}{c_{min}} \rfloor$ unseen items with cost and value matching those of t^*.*

Given the upper bound V^* on the optimal solution, we next propose a 2-approximation algorithm for top-1 CompRec which is guaranteed to be instance optimal (see below). The algorithm, InsOpt-CR, is shown as Algorithm 8. One item is retrieved from the source at each iteration of the algorithm (lines 3–4). After accessing this new item, we can use the pseudo-polynomial algorithm to find an optimal solution R^o *over the accessed itemset* S (line 5). We calculate the upper bound value V^* of the optimal solution using MaxValBound. If $v(R^o) \geq \frac{1}{2} \times V^*$, the algorithm terminates; if not, it continues to access the next item (lines 7–8). The following example shows how InsOpt-CR works.

Example 2 *Let $I = \{t_1, t_2, \ldots, t_n\}$, $n \geq 102$, be a top-1 CompRec instance, where $v(t_1) = v(t_2) = 101$, $c(t_1) = c(t_2) = 100$, for $i = 3, \ldots, 101$, $v(t_i) = c(t_i) = 1$, and for $i = 102, \ldots, n$, $v(t_i) = 1$ and $c(t_i) = 0.5$. Let $B = 199$. Clearly, $\mathcal{BG} = c_{min} = 0.5$. After accessing the first 101 items, $S = \{t_1, \ldots, t_{101}\}$, $R^o = \{t_1\} \cup \{t_3, \ldots, t_{101}\}$, $v(R^o) = 200$. Because $c_{min} = 0.5$ and $v_{min} = 1$, we can calculate $V^* = 398$ and InsOpt-CR will stop since $v(R^o) \geq \frac{1}{2} \times V^*$.*

[11] An unseen item t is feasible iff $v(t) \leq v_{min}$ and $c(t) \geq c_{min}$

Algorithm 8: InsOpt-CR(N, B, $\mathcal{BG}$)

```
1 S ← An empty buffer
2 while TRUE do
3     t ← N.getNext()
4     S.Insert(t)
5     (R^o,C) ← OptimalKP(S, B)
6     V* = MaxValBound(S,C,B,BG)
7     if v(R^o) ≥ 1/2 × V*
8         return R^o
```

Given a top-1 CompRec instance I with optimal solution OPT, because $V^* \geq v(OPT)$, if $v(R^o) \geq \frac{1}{2} \times V^*$, then $v(R^o) \geq \frac{1}{2} \times OPT$, so InsOpt-CR returns a correct 2-approximation of OPT.

To analyze the optimality of our proposed algorithm, we utilize the notion of *instance optimality* proposed in [53].

Definition 7 Instance Optimality: *Let $\mathcal{A}$ be a class of algorithms, and let $\mathcal{I}$ be a class of problem instances. Given a non-negative cost measure $cost(\mathcal{A}, \mathcal{I})$ of running algorithm $\mathcal{A}$ over $\mathcal{I}$, an algorithm $A \in \mathcal{A}$ is* instance optimal *over $\mathcal{A}$ and $\mathcal{I}$ if for every $A' \in \mathcal{A}$ and every $I \in \mathcal{I}$ we have $cost(A, I) \leq c \cdot cost(A', I) + c'$, for constants c and c'. Constant c is called the* optimality ratio.

To prove the instance optimality of InsOpt-CR, we first show the following.

Lemma 6 *Given any top-1 CompRec instance I and any 2-approximation algorithm A with background cost information $\mathcal{BG}$ and the same access constraints as InsOpt-CR, A must read at least as many items as InsOpt-CR.*

Proof *Assume to the contrary that there exists an instance I and a 2-approximation algorithm A that stops earlier than InsOpt-CR on I. Let $S = \{t_1, \ldots, t_n\}$ be the set of items accessed by A at the time it stops. Assume that, after accessing the items in S, Algorithm 8 has $(R^o, C) = OptimalKP(S, B)$ and upper bound V^* as calculated in line 6.*

Because A stops earlier than InsOpt-CR, the condition that $v(R^o) \geq \frac{1}{2} \times V^*$ must not hold as otherwise InsOpt-CR will also stop, contradicting the assumption that A stops earlier than InsOpt-CR.

Then it is easy to show that, at the time A stops, $v(R^o)$ must be less than $\frac{1}{2} \times V^*$. For this case, given only S and $\mathcal{BG}$, it is possible to form a top-1 CompRec instance I' which shares the same prefix S as I, but the optimal solution for I' is V^*. (For example, if S_{opt} corresponds to the optimal possible solution calculated by MaxValBound, $V^* = v(S_{opt})$, and $S' = S_{opt} \cap S$, we can set the next $\lfloor \frac{B-c(S')}{c_{min}} \rfloor$ unseen items to have value v_{min} and cost c_{min}.) Then even the optimal solution R^o for S is not a 2-approximation for instance I', which contradicts the fact that A is a 2-approximation algorithm.

Theorem 2 *Let $\mathcal{I}$ be the class of all top-1 CompRec instances, and $\mathcal{A}$ be the class of all possible 2-approximation algorithms that are constrained to access items in non-increasing order of their value. Given the same background cost information $\mathcal{BG}$, InsOpt-CR is instance optimal over $\mathcal{A}$ and $\mathcal{I}$ with an optimality ratio of one.*

Proof *From Lemma 6, for any top-1 CompRec instance I and any 2-approximation algorithm A with background cost information $\mathcal{BG}$ and the same access constraints as InsOpt-CR, A must read at least as many items as InsOpt-CR. According to the definition of instance optimality and the proposed cost model, InsOpt-CR is instance optimal over $\mathcal{A}$ and $\mathcal{I}$ with an optimality ratio of one.*

Top-k Composite Recommendations

In addition to the best composite recommendation, it is often useful to provide the user with the top-k composite recommendations, where k is a small constant. In this section, we extend the algorithm proposed in Section 4.3.4.3.1.4.3.1 to one that returns the top-k composite recommendations. Similar to the top-1 case, due to the hardness of the underlying problem, we seek an efficient approximation algorithm which can give us high quality recommendations.

Given an instance I of top-k CompRec, assume $\mathcal{R}^I$ is the set of all *feasible composite recommendations*, i.e., $\mathcal{R}^I = \{R \mid R \subseteq N \wedge c(R) \leq B\}$). Following Fagin et al. [53] and Kimelfeld et al. [73], we define an α-approximation of the top-k composite recommendations to be any set $\mathcal{R}_k$ of $\min(k, |\mathcal{R}^I|)$ composite recommendations, such that, for all $R \in \mathcal{R}_k$ and $R' \in \mathcal{R}^I \backslash \mathcal{R}_k$, $v(R) \geq \frac{1}{\alpha} \times v(R')$.

To produce top-k composite recommendations, we will apply Lawler's procedure [80] to InsOpt-CR. Lawler's procedure is a general technique for enumerating the optimal top-k answers to an optimization problem, which

relies on an efficient algorithm to find the optimal (top-1) solution to the problem.

Algorithm 9: InsOpt-CR-Topk(N, B, $\mathcal{BG}$, k)

```
1  S ← An empty buffer
2  R_k ← An empty result buffer
3  while TRUE do
4      t ← N.getNext()
5      S.Insert(t)
6      (R^o,C) ← OptimalKP(S, B)
7      V* = MaxValBound(S,C,B,BG)
8      if v(R^o) ≥ 1/2 × V*
9          EnumerateTopk(S, R^o, V*, B, R_k, k)
10         if |R_k| == k
11             return |R_k|
```

InsOpt-CR-Topk in Algorithm 9 is the InsOpt-CR algorithm modified using Lawler's procedure. As can been seen from the procedure, all we need to change is that instead of returning the 2-approximation solution found in Algorithm 8 (line 8), we enumerate at this point all possible 2-approximation solutions using Lawler's procedure (line 9). If the number of 2-approximation solutions is at least k, then we can report the top-k packages found; otherwise, we continue accessing the next item (line 10–11). Algorithm 10 is an overview of the enumeration algorithm EnumerateTopk. Whenever a 2-approximation is found, we call this routine to enumerate possible 2-approximation results which can be obtained from the accessed items. In line 3 of Algorithm 10, EnumerateTopk will directly call the enumeration procedure as proposed in [80].

Algorithm 10: EnumerateTopk(S,R^o,V^*,B,$\mathcal{R}_k$,k)

```
1  R_k.Add(R^o)
2  while |R_k| ≤ k do
3      R^n = LawlerOptimalNext(S, B)
4      if v(R^n) < 1/2 × V*
5          return R_k
6      R_k.Add(R^n)
```

In InsOpt-CR-Topk, the enumeration of all possible 2-approximation solutions is straightforward. Since we know the upper bound V^*, we can simply utilize Lawler's procedure to enumerate candidate packages which are under cost budget and have aggregated value of at least half of V^*.

Lemma 7 *Given any instance I of top-k CompRec and any 2-approximation algorithm A with the same background cost information $\mathcal{BG}$ and access constraints as InsOpt-CR-Topk, A must read as many items as InsOpt-CR-Topk.*

Proof *If algorithm A stops earlier than InsOpt-CR-Topk, then at the time it stops, EnumerateTopk must return a result set whose size is less than k, because otherwise InsOpt-CR-Topk would also stop. Then, similar to Lemma 6, it is not possible to find a result set in S which is guaranteed to be a 2-approximation to the optimal top-k composite recommendation, which means by properly selecting unseen items, the result set returned by A can be made not to be a 2-approximation, contradicting the assumption.*

Theorem 3 *Let $\mathcal{I}$ be the class of all top-k CompRec instances, and $\mathcal{A}$ be the class of all possible 2-approximation algorithms that are constrained to access items in the non-increasing order of their value. Given the same background cost information $\mathcal{BG}$, InsOpt-CR-Topk is instance optimal over $\mathcal{A}$ and $\mathcal{I}$ with an optimality ratio of one.*

Proof *From Lemma 7, for any instance I of top-k CompRec and any 2-approximation algorithm A with the same background cost information $\mathcal{BG}$ and access constraints as InsOpt-CR-Topk, A must read as many items as InsOpt-CR-Topk. So according to the definition of instance optimality, InsOpt-CR-Topk is instance optimal over $\mathcal{A}$ and $\mathcal{I}$ with an optimality ratio of one.*

4.3.2 Greedy Algorithms

Although the instance optimal algorithms presented above guarantee to return top-k packages that are 2-approximations of the optimal packages, they rely on an exact algorithm for finding an optimal package using the items seen so far. Because this may lead to high computational cost, we propose more efficient algorithms next. Instead of using an exact algorithm to get the best package for the currently accessed set of items S, we use a simple greedy heuristic to form a high quality package R^G from S and then test whether R^G is globally a high quality package.

Compared with InsOpt-CR, our greedy solution Greedy-CR for top-1 CompRec needs to replace OptimalKP in InsOpt-CR with GreedyKP, which uses greedy heuristics [71] to find a high quality itemset in polynomial time [12], and to change R^o to the greedy solution R^G. Furthermore, instead of using the tight upperbound calculated by MaxValBound, we need to use a heuristic upperbound which is calculated by Algorithm 11: MaxHeuristicValBound.

Algorithm 11: MaxHeuristicValBound(S, B, $\mathcal{BG}$)

```
1 τ ← v_min / c_min
2 Sort S = {t_1, ..., t_n} by value/cost ratio
3 m = max{m | v(t_m)/c(t_m) ≥ τ ∧ c(R_m) ≤ B}
4 R_m = {t_1, ..., t_m}
5 if m == n
6 |   V* = v(R_m) + τ * (B − c(R_m))
7 else
8 |   V* = v(R_m)+max{τ, v_{m+1}/c_{m+1}} * (B − c(R_m))
9 return V*
```

It follows from known results about knapsack that, similar to InsOpt-CR, Greedy-CR will always generate a correct 2-approximation to the optimal solution.

However, unlike InsOpt-CR, Greedy-CR is not instance optimal among all 2-approximation algorithms with the same constraints, as the following example shows.

Example 3 *Let $I = \{t_1, t_2, \ldots, t_n\}$, $n \geq 102$, be a top-1 CompRec instance (as considered in Example 2), where $v(t_1) = v(t_2) = 101$, $c(t_1) = c(t_2) = 100$, for $i = 3, \ldots, 101$, $v(t_i) = c(t_i) = 1$, and for $i = 102, \ldots, n$, $v(t_i) = 1$ and $c(t_i) = 0.5$. Let $B = 199$, $\mathcal{BG} = c_{min} = 0.5$ and approximation ratio $\alpha = 2$. From Example 2, we know that after accessing the first 101 items, $S = \{t_1, \ldots, t_{101}\}$, $v(R^o) = 200$, $V^* = 398$ and InsOpt-CR will stop. However, at this moment $R^G = \{t_1\}$, and $v(R^G) = 101 < \frac{1}{2} \times V^*$. So Greedy-CR will continue accessing new items and it can be easily verified that Greedy-CR needs to access another 98 items before it stops.*

[12] Note that any approximation algorithm for knapsack [71] can be plugged in here without affecting the correctness and instance optimality of the resulting algorithm.

We note that, in practice, cases such as the above may occur rarely. In fact, in our experimental results (Section 4.4) we observed that, on a range of datasets, Greedy-CR exhibited a very low running time while achieving similar access costs and overall result quality when compared to InsOpt-CR.

Similar to Section 4.3.4.3.1.4.3.1, we can easily extend Greedy-CR to Greedy-CR-Topk by using Lawler's procedure [80] to enumerate all possible high quality packages after one such package is identified. However, unlike InsOpt-CR-Topk which guarantees instance optimality, here we simply use Lawler's procedure to enumerate all candidate packages using the greedy algorithm instead of the exact algorithm. Similar to [73], we show in the following theorem that for top-k CompRec, if an α-approximation algorithm is utilized in Lawler's procedure instead of the exact algorithm which finds the optimal solution, we get an α-approximation to the top-k composite recommendations.

Theorem 4 *Given an instance I of top-k CompRec, any α-approximation algorithm A for top-1 CompRec can be utilized with Lawler's procedure to generate a set $\mathcal{R}_k$ of composite recommendations which is an α-approximation to the optimal set of top-k composite recommendations.*

Proof *We prove the above theorem using induction on k. Given α-approximation algorithm A and instance I, it is clear that the top-1 composite recommendation set $\mathcal{R}_1$ generated by A is an α-approximation to the optimal top-1 composite recommendation. Assume that the top-k composite recommendation set $\mathcal{R}_k$ generated by Lawler's procedure and A is an α-approximation to the optimal top-k composite recommendation, which means $\forall R \in \mathcal{R}_k$, $\forall R' \in \mathcal{R}^I \backslash \mathcal{R}_k$, $\alpha \cdot Value(R) \geq Value(R')$. Then according to the property of Lawler's procedure, the $k+1$st composite recommendation R_{k+1} generated is an α-approximation to the optimal composite recommendation from $\mathcal{R}^I \backslash \mathcal{R}_k$, so $\forall R' \in \mathcal{R}^I \backslash \mathcal{R}_k$, $\alpha \cdot Value(R_{k+1}) \geq Value(R')$. It is easy to verify that $\forall R \in \mathcal{R}_k \cup \{R_{k+1}\}$, $\forall R' \in \mathcal{R}^I \backslash (\mathcal{R}_k \cup \{R_{k+1}\})$, $\alpha \cdot Value(R) \geq Value(R')$, so the top-$(k+1)$ composite recommendation set $\mathcal{R}_{k+1}$ generated by A is an α-approximation to the optimal top-$(k+1)$ composite recommendation.*

So the quality of the packages generated by the resulting enumeration process can be guaranteed. In this enumeration process, given a candidate package, we use the greedy algorithm to get the next candidate package for each sub-search space in Lawler's procedure, and if all of them are not guaranteed to be 2-approximations, the enumeration will stop.

However, similar to Greedy-CR, it is obvious that Greedy-CR-Topk is not instance optimal. We note that, in practice, the difference between the

results generated by InsOpt-CR-Topk and Greedy-CR-Topk (in terms of the aggregate values of packages generated) may be very small.

4.3.3 Histogram-Based Optimization

For the algorithms proposed in previous sections, we have a very pessimistic assumption regarding background cost information, namely $\mathcal{B}G = c_{min}$, the global minimum cost. However in practice, we often have access to much better statistics about the costs of items, e.g., histograms. In this section, we will demonstrate how histograms can be incorporated to improve the proposed algorithms.

As can be observed from Algorithm 8, background cost information is used to determine a tight upperbound V^* on the value $v(OPT)$ of the optimal solution. A pessimistic background cost information assumption will result in a loose estimation of V^* and thus require more item accesses.

Let $\mathcal{H}$ be a histogram of the costs of items. $\mathcal{H}$ will divide the range of costs into $|\mathcal{H}|$ non-overlapping buckets $b_1, \ldots, b_{|\mathcal{H}|}$. For each bucket b_i, $\mathcal{H}$ stores the number of items whose cost falls inside the corresponding cost range. In classical histograms used in relational databases, it is often assumed that items which fall in the same bucket are uniformly distributed within the cost range. However, it can easily be shown that this assumption may result in an under-estimation of the upperbound value V^*, thus causing the proposed approximation algorithms to be incorrect. In order to solve this problem and guarantee that the estimated V^* is a correct upperbound of $v(OPT)$, we can simply assume that the cost of each item within a bucket b_i is the minimum item cost within b_i, denoted $b_i.c_{min}$.

In Algorithm 12, we list the modified procedure for determining the upperbound value V^* given the histogram $\mathcal{H}$ on the cost of unseen items. The algorithm will utilize a sub-procedure $MaxItems(\mathcal{H}, c)$ which, given the histogram $\mathcal{H}$ and a cost threshold c, returns the maximum number of items whose total cost is less than to equal to c.

Algorithm 12: MaxValBound-H(S, C, B, $\mathcal{H}$)

```
1 V* = MaxItems(H, B) * v_min
2 for v ∈ {1, ..., n · v̄} do
3     if C(n, v) < B
4         V*=max{V*, v + MaxItems(H, B − C(n, v)) * v_min}
5 return V*
```

Lemma 8 *Algorithm MaxValBound-H returns a correct upperbound of $v(OPT)$.*

Proof *Assume to the contrary that the V^* returned by MaxValBound-H is not a correct upperbound. Hence there must exist a configuration of unseen items such that there exists a package R for which $v(R) > V^*$. Let the set of seen items in R be R_{seen}, and the set of unseen items in R be R_{unseen}. Then during the execution of MaxValBound-H, according to the algorithm which calculates C, we must have considered a $C(n, v)$, such that $C(n, v) \leq c(R_{seen})$ and $v \geq v(R_{seen})$. And it is clear that for each unseen item t, its value is less than or equal to v_{min} and its cost is larger than or equal to the cost as indicated by the corresponding bucket in $\mathcal{H}$. So for $C(n, v)$, the set of unseen items R'_{unseen} considered by MaxValBound-H must have the property that $c(R'_{unseen}) \leq c(R_{unseen})$ and $v(R'_{unseen}) \geq v(R_{unseen})$. So $v(R) \leq V^* = v + v(R'_{unseen})$, which leads to a contradiction.*

As we will show in Section 4.4, histogram-based optimization can significantly improve the efficiency of the proposed algorithms.

4.4 Experiments

In this section, we study the performance of our proposed algorithms based on both real and synthetic datasets.

4.4.1 Experimental Setup and Datasets

		1st Package		2nd Package		3rd Package		4th Package		5th Package	
		Sum	Avg	Sum	Avg	Sum	Avg	Sum	Avg	Sum	Avg
MovieLens	Optimal	427	46.7	426	46.6	425	46.7	424	46.7	423	46.6
	InsOpt-CR-Topk	386	47.5	385	47.4	385	47.3	384	47.2	383	47.2
	Greedy-CR-Topk	384	47	381	47	380	46.8	379	46.7	379	46.7
TripAdvisor	Optimal	300	50	300	50	300	50	300	50	300	50
	InsOpt-CR-Topk	185	50	175	50	165	50	160	50	155	50
	Greedy-CR-Topk	220	50	210	50	210	50	205	50	205	50
Uncorrelated Data	Optimal	1092	36.4	1091	36.4	1090	36.3	1090	36.3	1089	36.5
	InsOpt-CR-Topk	929	43.6	926	43.6	925	43.6	925	43.6	924	43.5
	Greedy-CR-Topk	945	42.9	939	42.8	938	42.8	936	42.7	931	42.8
Correlated Data	Optimal	122	5.3	122	5.2	122	5.2	122	5.1	122	5.2
	InsOpt-CR-Topk	110	6.7	110	6.7	110	6.7	110	6.6	110	6.5
	Greedy-CR-Topk	110	6.6	110	6.6	109	7.6	109	6.5	109	7.15

Table 4.1: Quality Comparison for Different Composite Recommendation Algorithms

The goals of our experiments were: (i) evaluate the relative quality of Inst-Opt-CR and Greedy-CR compared to the optimal algorithm, in terms of both the total and average values of the top-k packages returned, and

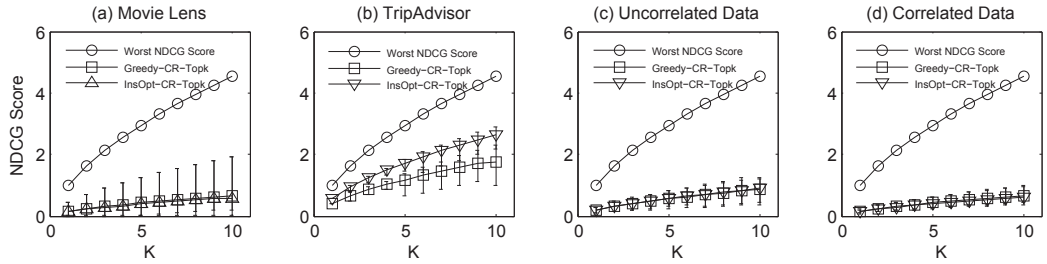


Figure 4.2: NDCG Score for Top-k Packages

(ii) evaluate the relative efficiency of the algorithms with respect to the number of items accessed and the actual run time. All experiments were done on a Xeon 2.5GHz Quad Core Windows Server 2003 machine with 16GB RAM and a 128GB SCSI hard disk. All code is in Java using JDK/JRE 1.6.

We use four datasets in our experiments. The first dataset is from MovieLens[13]. We use the 10 million rating MovieLens dataset which contains 1 million ratings for 3900 movies by 6040 users. In our experiments, we used the running time of movies, obtained from IMDB[14], as cost and we assume users are interested in packages of movies where the total running time is under a given budget.

TripAdvisor[15] is a well-known website where users can share and explore travel information. For our experiments, we crawled user rating information from places of interest (POIs) in the 10 most popular cities in the US, excluding POIs which had one or no reviews. The dataset contains 23658 ratings for 1393 POIs by 14562 users, so it is very sparse.[16] We associate with each POI in the dataset a cost which is based on log(number of reviews) and scaled to the range of 1 to 50. The intuition behind choosing such a cost function is that the more popular a POI is (in terms of number of reviews), the more likely it is to be crowded or for the tickets to be expensive. In practice, we may also use some existing work like [47] to mine from online user-generated itineraries other cost measures, e.g., average time users spent at each POI, average cost of visiting each POI, etc.

For the MovieLens and TripAdvisor datasets, we use a simple memory-based collaborative filtering algorithm [16][17] to generate predicted ratings

[13] `http://www.movielens.org (visited on 03/16/2015)`

[14] `http://www.imdb.com (visited on 03/16/2015)`

[15] `http://www.tripadvisor.com (visited on 03/16/2015)`

[16] Pruning more aggressively rendered it too small.

[17] Our algorithms do not depend on a specific recommendation algorithm; in practice,

for each user. The ratings are scaled and rounded to integers ranging from 1 to 50.

For the MovieLens dataset, we randomly selected 20 users from the 23594 user pool, and the budget for each user was fixed at 500 minutes[18]. For the TripAdvisor dataset, because of the sparsity of the underlying user rating matrix, we selected the 10 most active users as our sample for testing the algorithms, and set the user cost budget to 50.

We also tested our algorithms on synthetic correlated and uncorrelated datasets. For both datasets, item ratings are randomly chosen from 1 to 50. For the uncorrelated dataset, item costs are also randomly chosen from 1 to 50, but for the correlated dataset, the cost of item t is randomly chosen from $\min\{1, v(t)-5\}$ to $v(t)+5$. In both datasets, the total number of items is 1000, and the cost budget is set to 50. For all datasets, we assume the background cost information $\mathcal{BG}$ is simply the global minimum item cost.

4.4.2 Quality of Recommended Packages

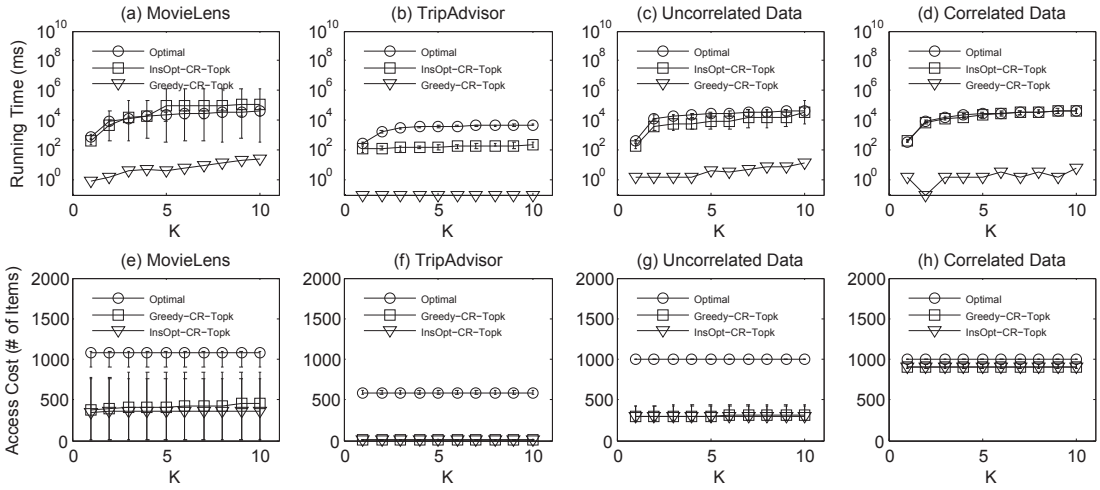


Figure 4.3: (a)–(d) Running Time for Different Datasets; (e)–(h) Access Cost for Different Datasets

For each dataset, Table 4.1 shows the quality of the top-5 composite recommendations returned by the optimal and approximation algorithms.

our framework assumes ratings come from existing recommender systems.

[18]For all datasets, we tested our algorithms under various cost budgets with very similar results, so other budgets are omitted.

We use as measures of quality the aggregated value of each package (SUM column) and the average item value of each package (AVG column).

It can be verified from Table 4.1 that our approximation algorithms do indeed return top-k composite packages whose value is guaranteed to be a 2-approximation of the optimal. Furthermore, from the average item value column, it is clear that our proposed approximation algorithms often recommend packages with high average value, whereas the optimal algorithm often tries to fill the package with small cost and small value items. So by sacrificing some of these lower quality items, the proposed approximation algorithms may manage to find high quality packages much more efficiently.

To better study the overall quality of returned packages, we also adopt a modified Normalized Discounted Cumulative Gain (NDCG) [66] to measure the quality of the top-k composite packages returned by the approximation algorithms against the optimal algorithm. Let $R^o = \{P_1^o, \ldots, P_k^o\}$ be the top-k packages returned by the optimal algorithm, and $R^a = \{P_1^a, \ldots, P_k^a\}$ be the top-k packages returned by the approximation algorithm. The modified NDCG score is a weighted sum of aggregated package value difference at each position of the returned top-k list, and is defined as:

$$NDCG(R^o, R^a) = \sum_{i=1}^{k} \frac{\log(1 + \frac{v(P_i^o) - v(P_i^a)}{v(P_i^o)})}{\log(1+i)}$$

The ideal value for the modified NDCG score is 0, where the top-k packages returned have exactly the same value as the optimal top-k packages. The worst possible value for the modified NDCG score is $\sum_{i=1}^{k} \frac{\log 2}{\log(1+i)}$, where each package returned has an aggregated value of 0. In Figure 4.2, we show for the 4 datasets the NDCG score of the top-k packages (k ranging over 1 to 10) returned by the instance optimal algorithm and the greedy algorithm. It is clear that, while having a substantial run time advantage, the greedy algorithm can achieve a very similar overall top-k package quality compared to the instance optimal algorithm. We also note that both approximation algorithms have a very small NDCG score.

4.4.3 Efficiency Study

The running times of our algorithms on the 4 datasets are shown in Figure 4.3 (a)–(d), while access costs are shown in Figure 4.3 (e)–(h). For MovieLens, TripAdvisor and the uncorrelated dataset, it can be seen that on average the greedy algorithm Greedy-CR-Topk has excellent performance in terms of *both* running time and access cost. The instance optimal algorithm

InsOpt-CR-Topk also has low access cost, but its running time grows very quickly with k since it needs to solve exactly many instances of knapsack, restricted to the accessed items.

As can be seen in Figure 4.3 (h), the only dataset where both the greedy and instance optimal algorithms have a high access cost is the correlated dataset (but notice that the greedy algorithm still has good running time). The reason for this is that, for the correlated dataset, the global minimum cost corresponds only to items which also have the least value. Thus the information it provides on the unseen items is very coarse. In practice, one solution to this might be to use more precise background cost information, such as provided by histograms, for example, as described in Section 4.3.3 and evaluated below.

In Figure 4.4, we compare the performance of the instance-optimal algorithm where the background cost information is simply the minimum cost with that where it is histogram-based. For the histogram-based approach, each histogram has 50 buckets and each bucket is of equal width. As can be seen from Figure 4.4 (e)–(h), the histogram-based approach consistently accesses fewer or an equal number of tuples compared to the minimum cost-based approach. This further results in savings in the overall running time of the algorithm, as can be seen from Figure 4.4 (a)–(d). The only exception is on the TripAdvisor dataset, where the histogram-based approach may sometimes be slightly slower than the minimum cost-based approach. We note that this is because, for this dataset, both approaches access only a few tuples. Thus, as indicated by Algorithm 12, our histogram-based approach may incur a small overhead in counting tuples which may not be necessary for this case. Similar savings to those reported in Figure 4.4 can be observed for the histogram-based greedy algorithm.

4.5 Discussion

As mentioned in Section 4.2, our framework includes the notion of a package satisfying compatibility constraints. For example, in trip planning, a user may require the returned package to contain no more than 3 museums.

To capture these constraints in our algorithms, we can define a Boolean *compatibility function* $\mathcal{C}$ over the packages under consideration. Given a package P, $\mathcal{C}(P)$ is *true* if and only if all constraints on items in P are satisfied. We can add a call to $\mathcal{C}$ in InsOpt-CR-Topk and Greedy-CR-Topk after each candidate package has been found. If the package fails the compatibility check, we just discard it and search for the next candidate package. In terms

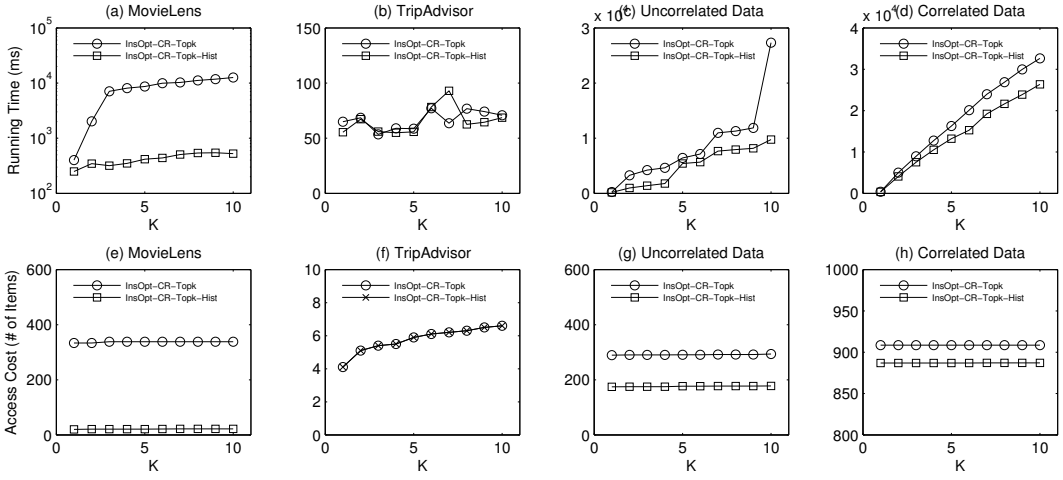


Figure 4.4: (a)–(d) Running Time Comparison for Different Datasets; (e)–(h) Access Cost Comparison for Different Datasets

of access cost, it can easily be verified that the modified InsOpt-CR-Topk algorithm is still instance optimal.

It is worth noting that the Boolean compatibility function defined here allows for greater generality than the constraints studied in previous work such as [19, 102]. However, for application scenarios where only one specific type of constraint is considered, e.g., having one item from each of 3 predefined categories, more efficient algorithms like Rank Join [54, 123] can be leveraged.

Furthermore, although in the previous algorithms we assume there is only one component recommender system, it is straightforward to combine recommendation lists from several component recommender systems by creating on-the-fly a "virtual recommendation list", e.g., select at each iteration the item which has the maximum value/rating across all recommender systems.

4.6 Application in Travel Planning

In travel planning, users may also have a budget on time that can be spend on a trip, and if this is the case, the underlying problem become a Orienteering problem [38].

Given a set N of POIs, a set U of users, an active user $u \in U$, and a POI $t \in N$, we denote by $v_u(t)$ the *value* of POI t for user u. We denote the value as $v(t)$ when the active user is understood. A RS predicts $v(t)$ when it

is not available, by using the active user's past behaviour and possibly that of other similar users. For $t \in N$, we denote by $tc(t)$ the *time cost* and by $mc(t)$ the *monetary cost*, of POI t. Given a set of POIs $R \subseteq N$, we define $v(R) = \Sigma_{t \in R}\, v(t)$, $tc(R) = \Sigma_{t \in R}\, tc(t)$ and $mc(R) = \Sigma_{t \in R}\, mc(t)$.

In addition to the cost associated with each POI, we may also need to consider the cost spent on traveling to corresponding POIs. For each POI pair (t_1, t_2), $t_1, t_2 \in N$, we denote by $d(t_1, t_2)$ the shortest distance between t_1 and t_2. And given a set of POIs $R \subseteq N$, we define $w(R)$ as the minimum distance walk which covers all POIs in R, and let $tcw(R)$ and $mcw(R)$ be the corresponding time and monetary cost for taking $w(R)$ by assuming an average speed/cost per unit of distance.

Definition 8 Top-k Composite Sequence Recommendations: *Given an instance I of a composite recommendation system consisting of one component RecSys and an external information source, a cost budget B_t on time, a cost budget B_m on money and an integer k, find the top-k packages $P_1, ..., P_k$ such that each P_i has $mc(P_i) + mcw(P_i) \leq B_m$, $tc(P_i) + tcw(P_i) \leq B_t$, and among all feasible packages, $P_1, ..., P_k$ have the k highest total values, i.e., $v(P) \leq v(P_i), 1 \leq i \leq k$ for all feasible packages $P \notin \{P_1, ..., P_k\}$.*

Note that ratings of POIs from the component RecSys are retrieved using sorted access, while the cost of a given POI is obtained via random access. Let c_s and c_r be the costs associated with these accesses. Then the total *access cost* of processing n POIs is $n \times (c_s + c_r)$. Notice that the number of POIs in the system is often huge, and c_s and c_r can be large compared to the cost of in-memory operations, as often for both accesses, information may need to be transmitted through the Internet.

In our system, we also assume some background cost information about each POI is available. The background cost information can be a histogram $\mathcal{H}$ collected from the cost database or just the minimum POI monetary (time) cost mc_{min} (tc_{min}). This information can be materialized in our system and be refreshed regularly by rescanning the cost database. We denote the background cost information as $\mathcal{BG}$.

When $k = 1$, the top-k composite sequence recommendation problem (CompRec-Seq) can be viewed as a variation of the orienteering problem [38] with the restriction that POIs can be accessed only in non-increasing order of their value. Similar to the composite recommendation problem for sets as discussed in the previous sections, we assume we have some simple background cost information such as the minimum shortest distance d_{min} between POIs.

4.6.1 Algorithm

Composite sequence recommendation is closely related to the orienteering problem [38], which seeks a maximum value walk on a graph subject to a budget constraint on the cost. To simplify the presentation, we will ignore discussing cost on each POI (i.e., the so-called node cost) as this can be handled by a reduction to the original orienteering problem [47].

Similar to the composite recommendation problem for sets of items, to minimize the number of POIs retrieved while having the result quality guaranteed, we need to adapt the algorithm template as described in previous sections and iteratively calculate the optimal solution for the accessed POI-set S along with a *tight upper bound* on the possible true optimal solution to the underlying composite sequence recommendation problem instance.

Given an (exponential-time) exact orienteering solver, we can calculate the optimal solution for the subgraph G of the original POI graph, induced by the accessed POI-set S. However, it is more challenging to get a *tight* upper bound on the value of the true optimal solution for the composite sequence recommendation problem.

Let d_{min} be the background cost information about the minimum distance between POIs, $v_{min} = \min_{t \in S} v(t)$ and t^* be an "imaginary" unseen POI which has value v_{min}. A tight upper bound on the possible true optimal solution can be computed by the procedure MaxOriValBound shown in Algorithm 13.

It can be proven that, using procedure MaxOriValBound, we can give a correct instance-optimal α-approximation algorithm CR-Seq-Top1 for the composite sequence recommendation problem. Similar to composite set recommendation, to get better practical performance, we can utilize approximation algorithms for the orienteering problem such as [38] instead of exact algorithms. However, the resulting algorithm will not be instance optimal.

Algorithm 13: MaxOriValBound(G,B_t,B_m,$\mathcal{BG}$)

```
1 foreach Edge (v1, v2) ∈ G do
2     n* = ⌊ d(v1,v2) / d_min ⌋
3     Add to G a path P between v1 and v2 which is composed of n* "imaginary"
      POIs, each a copy of t*, and d(P) = d(v1, v2)
4 S = OptimalOrienteer(G,B_t,B_m)
5 Augment S with "imaginary" POIs t* if possible
6 return v(S)
```

In addition to the best composite recommendation, it is often useful

to provide the user with the top-k composite recommendations, where k is a small constant. Similar to the top-1 case, due to the hardness of the underlying problem, we seek an efficient approximation algorithm which can give us high quality recommendations.

Given an instance I of the top-k composite recommendation problem, assume $\mathcal{R}^I$ is the set of all *feasible composite recommendations*, i.e., $\mathcal{R}^I = \{R \mid R \subseteq N \wedge mc(R) \leq B_m, tc(R) \leq B_t\}$ for composite set recommendation and $\mathcal{R}^I = \{R \mid R \subseteq N \wedge mc(R) + mcw(R) \leq B_m, tc(R) + tcw(R) \leq B_t\}$ for composite sequence recommendation. Following Fagin et al. [53], we define an α-approximation of the top-k composite recommendations to be any set $\mathcal{R}_k$ of $\min(k, |\mathcal{R}^I|)$ composite recommendations, such that, for all $R \in \mathcal{R}_k$ and $R' \in \mathcal{R}^I \backslash \mathcal{R}_k$, $v(R) \geq \frac{1}{\alpha} \times v(R')$.

Lawler's procedure [80] is a general technique for enumerating optimal top-k answers to an optimization problem, which relies on an efficient algorithm to find the optimal solution to the problem. We have proven in Section 4.3.2 that for the top-k composite recommendation problem, if the algorithm for the optimal solution in Lawler's procedure is replaced by an α-approximation algorithm, instead of optimal top-k answers, we get an α-approximation to the optimal set of top-k composite recommendations. So in our system we leverage Lawler's procedure and CR-Seq-Top1 to produce top-k approximate composite recommendations.

Chapter 5

Composite Recommendation with Soft Constraints

5.1 Introduction

Standard item recommender systems (IRS) recommend to users personalized lists of single items based on previous transactions of the users in the system. This model has become extremely popular because of its wide application and success on websites such as Amazon, Last.fm, and Netflix. For instance, even in 2006, 35% of purchases on Amazon originated from recommended items [77]. The promise of IRS has also been recognized by the database community, resulting in a series of recent research projects such as RecStore [82], FlexRecs [74], aimed at pushing IRS inside the database engine.

However, as pointed out in an influential survey [16], IRS suffers from limited flexibility. For instance, users may want recommendations of more than one item, e.g., a shopping cart of cell phones, accessories, and data plans on Amazon, lists of songs on Last.fm, or lists of movies on Netflix. For realizing this kind of *package recommendation system* (PRS) using current item recommendation interfaces, a user has to either manually assemble the packages by exploring recommended items (e.g., Amazon), or browse and search packages created by other people (e.g., Last.fm, Netflix). With an exponential number of possible combinations of items, as well as a huge number of user-created packages, both approaches for finding packages of interest quickly become tedious. For a business, the drawback of not being able to find packages for users means lost opportunities to sell packages, or lowering user satisfaction by drowning users in oceans of user-created content. Thus it is desirable to have a PRS which can *learn* users' interests and recommend to each user packages that are of potential interest.

Given a set of n items, if there is no bound on the size of a package, there exist $2^n - 1$ possible packages of items (see Figure 5.1). Consider the utility $U(p)$ of a package p to a user u. Obviously $U(p)$ depends on the items in the

(a)

item	f_1	f_2
t_1	0.6	0.2
t_2	0.4	0.4
t_3	0.2	0.4

(b)

package	items	package	items
p_1	$\{t_1\}$	p_5	$\{t_2,t_3\}$
p_2	$\{t_2\}$	p_6	$\{t_1,t_3\}$
p_3	$\{t_3\}$	p_7	$\{t_1,t_2,t_3\}$
p_4	$\{t_1,t_2\}$		

Figure 5.1: Examples of packages of items.

package p. As discussed in [16, 87, 129], the feature values of each package p are usually aggregations of item feature values. E.g., let f_1 in Figure 5.1(a) be the cost of an item. For package p, $\mathsf{sum}_1(p)$ defines the total cost of the items in p, which is the cost of the package. Let f_2 in Figure 5.1(a) be the rating of an item. Then $\mathsf{avg}_2(p)$ defines the average rating of the items in p, i.e., the average quality of the package. In Figure 5.1(b), $\mathsf{sum}_1(p_4) = 1$ and $\mathsf{avg}_2(p_4) = 0.3$. For instance, when purchasing a package of books or CDs from Amazon, users may want the average rating of items in the package to be as high as possible, and the total cost of items to be as small as possible. So $U(p) = g(\mathsf{sum}_1(p), \mathsf{avg}_2(p))$, where function g is increasing in $\mathsf{avg}_2(p)$, and decreasing in $\mathsf{sum}_1(p)$. Similar examples can also be found when reasoning about the utility of packages on Last.fm and Netflix, where the cost of an item can be the price of a song/movie, and the rating can take the form of any combinations of the average ratings, the number of listens, the number of likes, and the number of purchases of a song/movie.

Given this general framework of characterizing features and utilities of packages, one intuitive way of recommending packages is to present to the user all *skyline packages* [87, 129], i.e., packages which cannot be *dominated* by another package on every feature. In the above example, a package p_1 is a skyline package if there does not exist another package whose total cost is lower and average rating is higher. However, as shown empirically in [87, 129], the number of skyline packages can be in the hundreds or even thousands for a reasonably-sized dataset. So presenting all of these packages to a user is impractical.

Another way to recommend packages is to define *hard constraints* on some features, and optimize the remaining features in the form of a utility function (e.g., see [120, 121]). For the above example, we could require the total cost of a package to be at most \$500, and then find packages which satisfy this constraint and maximize the average ratings of items in the package. However, this approach also has the following practical limitations. Firstly, users often only have a rough idea of what they want in a desirable

package. E.g., when summing costs of all items in a package, they may only say "smaller is better". Thus hard constraints on a feature may result either in sub-optimal packages when the budget is set too low, or a huge number of candidate packages when the budget is set too high. Secondly, the importance of each feature specified by the user is usually unknown. E.g., for some users, the monetary budget may not be that important and they can afford to pursue a high quality package while sacrificing a "reasonable" amount of money; whereas other users may be very sensitive to the total cost of the package, and may have limited flexibility in terms of monetary budget. We note that a PRS may be informed about the relative importance of the different criteria by the user; however, it is not realistic to expect a user to know, e.g., that they are 0.8 interested in the overall cost, and 0.2 interested in the overall quality of a package!

Instead, following recent work on multi-dimensional ranking of items [27, 34, 63], we take a *quantitative* approach to ranking packages based on multiple criteria. Specifically, we consider that for each user u, there is an intuitive "implicit" linear utility function U, which captures u's preference or trade-off among different features for choosing a desirable package. By leveraging this utility function, we can easily rank all packages, and present the best ones to the user. E.g., in the above example, for a user u who has equal preference on the cost and average item rating of a package, we can use $U(p) = -0.5\mathsf{sum}_1(p) + 0.5\mathsf{avg}_2(p)$, where a negative weight is used to indicate that smaller cost is better.

This approach captures all the criteria of the package ranking problem in a single score function, thus it avoids the first issue regarding hard constraints. However, as mentioned in the second issue, it is unrealistic to assume that the user will define the utility function *exactly* for the system. Thus, similar to recent work such as [27, 34], *we will assume the weights of the utility function are hidden*, and follow a *preference elicitation* framework which *explores* and *exploits* a user's preferences based on feedback received, and learns the hidden weights of the utility function over time.

Preference elicitation (PE) has been studied extensively in the Artificial Intelligence community [27, 34]. The general idea of PE is to capture users' preferences using a utility function, and then learn the parameters of this utility function through feedback w.r.t. certain elicitation queries. Most of this work relies on a specific type of query, called a *gambling query*. Though using gambling queries is well founded in terms of decision theory, to date it has only been applied to applications with *extremely small domains*. Also the form of the gambling query requires that the user be explicitly asked this query by the system through protocols such as a user survey. This limitation

makes it undesirable for deployment in a recommender system where user feedback either needs to be very simple such as "like"s or ratings, or to be taken implicitly, e.g., from item click-throughs on web sites.

In this chapter, we propose to use simple package comparison as the elicitation query. Users are presented with a list of recommended packages whenever they login to the system. These packages are composed of (i) top packages, w.r.t. the current knowledge of the user's utility function, selected according to a chosen ranking semantics as discussed in Section 5.2.3, and (ii) a set of random packages, which are used to explore uncertainty in users' preferences. Users' interaction with the recommended packages are logged as implicit signals to the system, showing that they are more interested in the clicked package than the other packages (noise in user feedback is discussed in Section 5.6). A user can choose to adopt any of the presented packages, and once they do so, their feedback is extracted implicitly, without causing any disruption in their normal interaction with the RS. This means the proposed framework can be cleanly integrated into existing services to capture and update users' preferences, without any abrupt interruption for the user using the service. This is in contrast with standard PE methods which require interactive elicitation and a dramatic change to the service work flow.

Specifically, our proposed PRS assumes each user has an associated utility function U which is parameterized by a weight vector w, and the uncertainty of U is captured by a distribution $\mathbf{P}_w$ over the space of possible weight vectors w. We assume that the prior of $\mathbf{P}_w$ is a mixture of Gaussians following [34], as mixture of Gaussians can approximate any arbitrary probability density function. Given $\mathbf{P}_w$, our PRS can directly leverage it to present the user with a small number of recommended packages, and record the user's interaction with the packages. However, a major challenge is that the posterior of $\mathbf{P}_w$ given user feedback has no closed form solution, as we shall see. To circumvent this, we propose a sampling-based framework which obviates the need for a posterior. Instead, package preferences resulting from user feedback can be translated into constraints on the samples from $\mathbf{P}_w$. However, this raises the question of how we can obtain samples satisfying the constraints as efficiently as possible. A related question is whether previously obtained samples can be *maintained* against new user feedback. We discuss our solution to these challenges in Section 5.3.

Finally, we note that following the Bayesian uncertainty-based framework, the posterior distribution of $\mathbf{P}_w$ at any time captures the current optimal representation of a user's preferences over packages w.r.t. all observed feedback.

Contributions of this chapter is listed as follows:

- We propose a PRS which captures user preferences using a linear utility function, but avoids the limitations of skyline packages and hard constraint-based systems;
- To solve the problem of determining parameters for the utility function, we leverage a non-intrusive Bayesian-based PE framework by assuming a prior distribution $\mathbf{P}_w$ over parameter w;
- The posterior of $\mathbf{P}_w$ given user feedback has no closed form solution in general, so we propose different constrained sampling strategies to solve this problem. We show that an approach based on simple rejection sampling may waste many samples, resulting in poor overall performance, whereas more sophisticated sampling strategies such as importance sampling and MCMC-based sampling make better use of the feedback, and are more efficient.
- Given the utility function U and the uncertainty which is captured by $\mathbf{P}_w$, we discuss how top-k packages can be generated w.r.t. our current knowledge of the user's preferences, following different ranking semantics inspired by recent work from different communities.
- Finally, we address the problem of how to maintain the set of samples generated when new feedback is received.

The rest of the chapter is organized as follows. We define our package recommendation problem in Section 5.2, with the sampling-based solution discussed in Section 5.3. Our algorithm for finding the best packages based on a set of sample weight vectors is presented in Section 5.4. Experimental results demonstrating the efficiency and effectiveness of various sampling methods and ranking algorithms are reported in Section 6.5.

5.2 Problem Setting

5.2.1 Package Profiles

We assume that we are given a set T of n items, each item being described by a set of m features $\{f_1, \ldots, f_m\}$. Each item $t \in T$ can be represented as an m-dimensional vector $\vec{t} = (t_1, \ldots, t_m)$, where t_i denotes the value of item t on feature f_i. For simplicity, when no ambiguity arises, we use t to denote both an item and its corresponding feature vector $\vec{t}$. Also, without loss

of generality, we assume all feature values are non-negative real numbers. In practice, different items can be associated with different feature sets, so some feature values for an item t might be `null`.

As mentioned in the introduction, users' preferences over packages are usually based on *aggregations* over feature values of items in a package. E.g., the sum of the costs of items defines the overall cost of a package, while the average of the ratings of items defines the overall quality of a package. Thus we define an *aggregate feature profile* (or simply *profile*) of a package as follows.

Definition 9 *An* aggregate feature profile *(or* profile*) is defined as* $V = (\mathsf{A}_1, \ldots, \mathsf{A}_m)$*, where each* A_i *corresponds to feature* f_i*,* $1 \leq i \leq m$*, and is one of the aggregation functions* min*,* max*,* sum*,* avg *or* `null`*, where* `null` *means that the corresponding feature* f_i *should be ignored.*

Note that we simplify the presentation by assuming one aggregation per feature, but our proposed algorithms can be easily extended to handle more than one aggregation per feature. Given a package p and a profile V, we define the feature value vector $\vec{p}$ of p w.r.t. V as $\vec{p} = (\mathsf{A}_1(p), \ldots, \mathsf{A}_m(p))$, where each $\mathsf{A}_i(p)$ is the aggregate value of items in p w.r.t. feature f_i. Following the usual semantics for evaluating aggregate functions, for min, max and sum we have $\mathsf{A}_i(p) = \mathsf{A}_i(\{t_i \mid t \in p \wedge t_i \neq \texttt{null}\})$, and for avg we have $\mathsf{avg}_i(p) = \mathsf{sum}_i(\{t_i \mid t \in p \wedge t_i \neq \texttt{null}\})/|\mathsf{p}|$. Similar to the feature value vector of an item, when there is no ambiguity, we simply use p to denote both the package p and its corresponding feature value vector $\vec{p}$. Furthermore, we denote each feature value $\mathsf{A}_i(p)$ of package p by p_i when profile V is clear from the context.

Note that given a fixed item set T, it is trivial to calculate the maximum aggregate value for a feature that can be achieved by any package. E.g., for $\mathsf{avg}_1(p)$, the maximum average value on f_1 that can be achieved by any package is simply the maximum f_1 value of all the items. So we assume in the following that each individual aggregate feature value is normalized into $[0, 1]$ using the maximum possible aggregate value of the corresponding feature.

5.2.2 Package Utility and Preference Elicitation

Intuitively, the utility of a package p for a user depends on its feature vector and we wish to learn this utility. The space of all mappings between possible aggregate feature values and utility values is uncountable, making this task challenging. Fortunately, most preferences exhibit an internal structure that

can be used to express the utility concisely, e.g., an *additive utility function* is commonly assumed in practice [70]. In this work, for a package p and a given profile V, we assume the utility of p can be specified using an additive utility function U, which uses a weighted linear combination of the corresponding (aggregate) feature values in p.

$$U(p) = w_1 p_1 + \cdots + w_m p_m \tag{5.1}$$

For simplicity, we use w to denote the weight vector $(w_1, \ldots, w_m)$. Without loss of generality, we assume each parameter w_i falls in the range $[-1, 1]$, where a positive (negative) w_i means a larger (resp., smaller) value is preferred for the corresponding feature. Note that we can transform all negative w_i's into their absolute value by setting $p'_i = 1 - p_i$ on the corresponding feature for all packages.

A framework based on a utility function essentially defines a total order over all packages, where similar to previous works such as [112], we assume ties in utility score are resolved using a deterministic tie-breaker such as the ID of a package. This differentiates the approach from that of [87, 129] which aim to return all *skyline packages*, the number of which can be prohibitively large, as previously noted.

Despite its intuitive appeal, there are two major challenges in adopting the utility-based framework for PRS in practice. First, users are usually not able to specify (or even know beforehand) the exact weights w_i of the utility function U. Thus, we must model the uncertainty in U, and elicit user's preferences by means of interactions. Second, unlike [87] and [129] which consider packages of fixed size, we allow package size to be flexible in our framework. We believe this is natural. E.g., given a system-defined maximum package size ϕ of say 20, we consider all possible package sizes ranging from 1 to 20. Efficient determination of packages of flexible size that maximize a user's utility under partial knowledge about the utility function from elicited preferences is far more challenging than finding packages of a given fixed size.

One popular way of characterizing uncertainty in U is through Bayesian uncertainty [27, 34][19], in which for each user, we assume the exact value of the weight vector w is not known, but w can be described by a probability distribution $\mathbf{P}_w$. We assume w follows a mixture of Gaussians: indeed, it has been shown that a mixture of Gaussians can approximate any arbitrary probability density function [22].

[19]The other possibility is strict uncertainty, which requires a set of possible utility functions (down to the weights) to be known, which is more restrictive.

While $\mathbf{P}_w$ can be initialized with a system-defined default distribution, in the long run $\mathbf{P}_w$ can be learned by leveraging the feedback provided by the user. In this work, we assume user feedback is in the form of selecting one preferred package from a set of packages presented to them. This form of feedback is popularly known as *example critiquing* in conjoint analysis [115] and preference elicitation [89].

For a given user u, let the feedback from u preferring package p_1 to package p_2 be denoted by $p_1 \succ p_2$. This feedback can be leveraged to update the posterior of $\mathbf{P}_w$ through Bayes' rule in Equation 5.2, where $\mathbf{P}(p_1 \succ p_2 \mid w)$ defines the likelihood of $p_1 \succ p_2$ given w. Note that since each specific w defines a total order over all packages, the value of $\mathbf{P}(p_1 \succ p_2 \mid w)$ is either one or zero. We tentatively assume that every user feedback is consistent, in that the provided preferences correspond to a partial order, and discuss in Section 5.6 how this assumption can be relaxed.

$$\mathbf{P}_w(w \mid p_1 \succ p_2) = \frac{\mathbf{P}(p_1 \succ p_2 \mid w)\mathbf{P}_w(w)}{\int_w \mathbf{P}(p_1 \succ p_2 \mid w)\mathbf{P}_w(w)\mathrm{d}w} \tag{5.2}$$

However, Gaussian mixtures are not closed under this kind of update [27], meaning we cannot obtain a closed-form solution for the posterior as presented in the above equation. One popular way to deal with such a situation is to force the posterior to again be a mixture of Gaussians, and thus the posterior can be learned by refitting a Gaussian mixture through algorithms such as expectation maximization (EM) [22]. However, the cost of refitting through EM is extremely high, so we take a different approach of representing the posterior by maintaining both the prior distribution and the set of feedback preferences received. The details of our proposal are described in Section 5.3.1.

5.2.3 Presenting Packages

While the preference elicitation frameworks discussed in the previous section can be exploited to update the knowledge of $\mathbf{P}_w$ for any specific user, there is still a remaining question of how to select and present packages to a user in order to get feedback.

In general, given the uncertainty in the utility function, packages presented to the user serve as a way to *explore* and *exploit* users' preferences simultaneously. I.e., on the one hand, we want to exploit our current knowledge about a user's preferences and try to present to them the best packages possible according to the current $\mathbf{P}_w$. On the other hand, we want to explore the uncertainty in the user's preferences, and present packages to them

(a)

w	values	prob.
w_1	(0.5,0.1)	0.3
w_2	(0.1,0.5)	0.4
w_3	(0.1,0.1)	0.3

(b)

profile
$(\mathsf{sum}_1, \mathsf{avg}_2)$

(c)

utility	p_1	p_2	p_3	p_4	p_5	p_6
w_1	0.35	0.3	0.2	0.575	0.4	0.475
w_2	0.31	0.54	0.52	0.475	0.56	0.455
w_3	0.11	0.14	0.12	0.175	0.16	0.155

(d)

top-2 packages	
w_1	p_4,p_6
w_2	p_5,p_2
w_3	p_4,p_5

Figure 5.2: Examples of different ranking semantics.

which might not be considered by our current knowledge about the user's preferences. These packages serve the purpose of correcting bias introduced from the initial distribution of $\mathbf{P}_w$ and combating mistakes and noise from user feedback. In this work, we follow a simple but promising way of presenting packages to the user by mixing current best packages along with random packages, so that the current best packages can be used to exploit our current knowledge about the user, and the random packages can be used to explore the user's preferences.

While it is straightforward to pick a random package, it is challenging to pick the best packages, as there is no universally accepted semantics on how packages should be ranked given the uncertainty in the utility function. Instead of committing to a specific package ranking semantics, we consider various alternative semantics, and discuss how the different semantics can be neatly integrated into our PRS framework.

The first ranking semantics we consider is based on expectation (EXP), which has been adopted as the most popular semantics for ranking items in preference elicitation papers in the artificial intelligence community [27, 34]. In the following, by a package space P, we mean the set of all possible packages formed using items from T and having size no larger than ϕ (the maximum package size).

Definition 10 (EXP) *Given a package space P and probability distribution $\mathbf{P}_w$ over weight vectors w, find the set of top-k packages P_k w.r.t. expected utility value, i.e. $\forall p \in P_k$, $\forall p' \in P \backslash P_k$, $\mathbf{E}_w(w \cdot p) \geq \mathbf{E}_w(w \cdot p')$.*

Example 4 *Consider the example in Figure 5.1. Assuming the maximum package size is 2, the package space P is given by $\{p_1, \ldots, p_6\}$. If the profile under consideration is $(\mathsf{sum}_1, \mathsf{avg}_2)$, then the maximum value for a size-2 package on feature 1 is 1, and the maximum value of a size-2 package on feature 2 is 0.4. We can normalize packages' feature values using these two maximum values. E.g., for package p_1 in Figure 5.1(b),*

$\mathsf{sum}_1(p_1) = 0.6$, $\mathsf{avg}_2(p_1) = 0.2$, *so the normalized feature value vector for* p_1 *is* $(0.6/1, 0.2/0.4) = (0.6, 0.5)$. *To simplify the presentation, we assume in Figure 5.2(a) that there are only three weight vectors,* w_1, w_2 *and* w_3, *under consideration, the probability for which is given in the third column. Given the weight vector information, we can easily calculate the utility of each package under each weight vector, as shown in Figure 5.2(c). E.g., the utility of package* p_1 *under* w_1 *is* $0.6 \times 0.5 + 0.5 \times 0.1 = 0.35$. *The expected utility value for each package can be calculated accordingly, using the probability of each weight vector. E.g., the expected utility for* p_1 *is* $0.35 \times 0.3 + 0.31 \times 0.4 + 0.11 \times 0.3 = 0.262$. *For this example, it is not difficult to verify that* p_4 *has the largest expected utility, followed by* p_5. ■

The second ranking semantics we consider is based on the probability of a package being in the top-σ position under different parameter settings (TKP). This is inspired by recent work on learning to rank in the machine learning community [33]. Let $P_{\succ}(p \mid w) = \{p' \mid p' \in P, w \cdot p' > w \cdot p\}$ denote the set of packages in P which have utility larger than package p, given a fixed w. Let $W_{\succ}$ denote the set of weight vectors w under which a package p is dominated by σ or fewer other packages, i.e., $|P_{\succ}(p \mid w)| \leq \sigma$. Since the utility function is convex, we can readily show that $\forall$ w_1, w_2, $w_1 \neq w_2$, if $w_1 \cdot p' > w_1 \cdot p$, and $w_2 \cdot p' > w_2 \cdot p$, then for any $\alpha \in [0, 1]$, $(\alpha w_1 + (1-\alpha)w_2) \cdot p' > (\alpha w_1 + (1-\alpha)w_2) \cdot p$. Thus we can prove that $W_{\neg\succ} := \{w \mid \sigma < |P_{\succ}(p \mid w)|\}$ forms a continuous and convex region, and $W_{\succ}$ is also continuous. So we define the probability of $p \in P$ being ranked among the top-σ packages as $\mathbf{P}(p \mid \mathbf{P}_w, \sigma) = \int_{w \in W_{\succ}} \mathbf{P}_w(w) \mathrm{d}w$.

Definition 11 (TKP) *Given a package space* P *and a probability distribution* $\mathbf{P}_w$ *over weight vectors* w, *find the top-*k *packages* P_k *w.r.t. the probability of being ranked in the top-*σ *positions, i.e.,* $\forall p \in P_k$, $\forall p' \in P \backslash P_k$, $\mathbf{P}(p \mid \mathbf{P}_w, \sigma) \geq \mathbf{P}(p' \mid \mathbf{P}_w, \sigma)$.

Example 5 *In Figure 5.2(d), we show the top-2 package list for each weight vector. We can calculate that the probability of* p_5 *being in a top-2 package list is* $0.4 + 0.3 = 0.7$. *Package* p_5 *has the largest probability of all candidate packages, followed by* p_4 *for which the probability is 0.6.* ■

The third and fourth ranking semantics we consider are the most probable ordering (MPO) and the optimal rank aggregation (ORA), which have been discussed in recent work on sensitivity analysis of querying top-k items under uncertainty [112]. We note that unlike EXP and TKP which represent the desirability of each individual package independently, adapted to

our setting, MPO and ORA represent the desirability of the top-k package list P_k as a whole.

For MPO, given a fixed w, let $I(P_k \mid w)$ be an indicator function which denotes whether P_k is the top-k package under w, i.e., $I(P_k \mid w) = 1$ if $\nexists p' \in P \backslash P_k$, $w \cdot p' > w \cdot p$, for all $p \in P_k$; and $I(P_k \mid w) = 0$ otherwise. Let W_{P_k} denote the set of weight vectors w under which $I(P_k \mid w) = 1$. Similar to TKP, we can show W_{P_k} forms a continuous region, so the probability of P_k being the top-k package can be defined as $\mathbf{P}_o(P_k \mid \mathbf{P}_w) = \int_{w \in W_{P_k}} \mathbf{P}_w(w) \mathrm{d}w$.

Definition 12 (MPO) *Given a package space P and a probability distribution $\mathbf{P}_w$ over weight vectors w, find the top-k packages P_k w.r.t. the most probable ordering, i.e., $\forall P'_k \subseteq P$, $|P'_k| \leq k$, $P'_k \neq P_k$, $\mathbf{P}_o(P_k \mid \mathbf{P}_w) \geq \mathbf{P}_o(P'_k \mid \mathbf{P}_w)$.*

Example 6 *In Figure 5.2(d), we can directly see the probability of each top-2 package list by referring to the probability of the corresponding weight vector. Clearly, the best top-2 package list under MPO is p_5, p_2.* ■

Lastly, we consider ORA. The original definition of ORA for item ranking is based on possible rankings of all items in the database [112], which is obviously undesirable in our case since the package space is exponential. Thus we adapt ORA in line with recent work on comparing top-k lists [52].

Let $D(P_k, P'_k)$ be the *Kendall tau distance with penalty parameter θ* between two top-k lists P_k and P'_k (similar development can be done using *Spearman's footrule*) [52], which counts the number of pairwise disagreements in the relative order of packages in the two top-k package lists. For two distinct packages $p_1, p_2 \in P_k \cup P'_k$, $p_1 \neq p_2$, consider the following four cases: (1) $p_1, p_2 \in P_k$ and $p_1, p_2 \in P'_k$. If the utility value order between p_1 and p_2 is different in the two lists, we set $D_{p_1,p_2}(P_k, P'_k) = 1$, otherwise we set $D_{p_1,p_2}(P_k, P'_k) = 0$; (2) Both packages appear in one list, and only one package appears in the other list. W.l.o.g., assume $p_1, p_2 \in P_k$, $p_1 \in P'_k$. We set $D_{p_1,p_2}(P_k, P'_k) = 1$ if p_2 is ranked higher than p_1 in P_k, otherwise we set $D_{p_1,p_2}(P_k, P'_k) = 0$; (3) If one package appears in P_k, while the other package appears in P'_k, then we set $D_{p_1,p_2}(P_k, P'_k) = 1$; (4) If both packages appear in one list and neither appears in the other list, we set $D_{p_1,p_2}(P_k, P'_k) = \theta$. $D(P_k, P'_k)$ can be defined as follows.

$$D(P_k, P'_k) = \sum_{p_1 \neq p_2, p_1, p_2 \in P_k \cup P'_k} D_{p_1,p_2}(P_k, P'_k) \tag{5.3}$$

Given $D(P_k, P'_k)$, ORA tries to find the "centroid" top-k package list which minimizes its distance to all possible top-k package lists given $D(P_k, P'_k)$

and the probability $\mathbf{P}_o(P'_k \mid \mathbf{P}_w)$ of P'_k being ranked as the actual top-k package, which is the same as in MPO.

Definition 13 (ORA) *Given a package space P and a probability distribution $\mathbf{P}_w$ over weight vectors w, find the top-k packages P_k w.r.t. the Kendall tau distance with penalty parameter θ, i.e., $P_k = \arg\min_{P_k} \sum_{P'_k} D(P_k, P'_k) \cdot \mathbf{P}_o(P'_k \mid \mathbf{P}_w)$.*

Example 7 *In the example in Figure 5.2, we enumerate all possible 2-package lists, finding the distance between each of these and each top-2 package list. E.g., for (p_4, p_6) and (p_4, p_5), the distance is 1, since the two lists do not agree on the order between p_5 and p_6. Then we calculate the overall expected distance between each 2-package list and each top-2 package list according to the definition of ORA. In our example, the best top-2 package list under ORA is (p_4, p_5) Its distance to the three top-2 lists (p_4, p_6), (p_5, p_2), and (p_4, p_5) is 1, 2, and 0 respectively, and its expected distance to these three top-2 lists is $1 \times 0.3 + 2 \times 0.4 + 0 \times 0.3 = 1.1$.* ■

In summary, while different ranking semantics might lead to the same top-k packages (e.g., in our example, p_4, p_5 are the top-2 packages for both EXP and ORA), or they might lead to very different top-2 packages (e.g., in our example, the top-2 packages for MPO are p_5, p_2, while the top-2 packages for TKP are p_5, p_4). We note that these different ranking semantics have been successfully adopted in different communities, and as we shall see in Section 5.3, our proposed PE framework can be easily adapted to leverage any of these different ranking semantics.

5.3 A Sampling-based Framework

To accommodate the preference elicitation framework and various ranking semantics for selecting packages for recommendation, we propose to use a sampling-based framework for PRS. Unlike the geometric approach proposed in previous papers such as [112], a sampling-based solution can be easily adapted to handle cases with higher dimensionality, as we will show empirically in Section 6.5. We first discuss simple rejection sampling in Section 5.3.1. We then consider more sophisticated sampling techniques in Section 5.3.2. In Section 6.4.2, we discuss how to optimize the constraint violation checking process for sample generation. Finally in Section 5.3.4, we discuss how previously generated samples can be reused given newly received user feedback, i.e., we discuss sample maintenance.

5.3.1 Rejection Sampling

Given the distribution $\mathbf{P}_w$ over w, an intuitive solution for finding the best packages under $\mathbf{P}_w$ is first to sample the weight vectors w according to $\mathbf{P}_w$, and then for every w sampled, try to find the best package under w. The best package results obtained from each sampled w can be aggregated for estimating the final list of best packages. The required aggregation logic depends on the ranking semantics and the details will be discussed in Section 5.4. This approach is intuitive: w's are sampled from $\mathbf{P}_w$, and packages which are ranked higher under these w's that have a higher probability are likely to be given a greater weight.

As discussed in Section 5.2.2, given current recommendations to the user and the feedback received, we need to constantly refit the distribution $\mathbf{P}_w$ so that it reflects the updated user preferences. However refitting the Gaussian mixture $\mathbf{P}_w$, say using the EM algorithm [27], after every received feedback using Equation (5.2) can be extremely time consuming. So a naïve way of performing *refit-and-sample* may be inefficient. Thus we consider an alternative approach of maintaining both the prior distribution $\mathbf{P}_w$ and all feedback *without* refitting the Gaussian mixture.

The key idea is that every feedback $p_1 \succ p_2$ rules out weight vectors w under which $p_1 \succ p_2$ is not true. For those w's which do satisfy $p_1 \succ p_2$, the feedback alone does not change their relative order with respect to $\mathbf{P}_w$, i.e., for w_1, w_2 which both satisfy $p_1 \succ p_2$, if $\mathbf{P}_w(w_1) > \mathbf{P}_w(w_2)$, without any further information, we have $\mathbf{P}_w(w_1 \mid p_1 \succ p_2) > \mathbf{P}_w(w_2 \mid p_1 \succ p_2)$.

So this means that we can use rejection sampling to sample directly from the posterior. I.e., we can sample a random w from the current $\mathbf{P}_w$, and if w violates any user feedback, we reject this sample. Otherwise, the sample is accepted. Clearly the rejection sampling method will only keep samples which conform to the feedback received from the user, and as shown above, the relative order of probabilities of two weight vectors being sampled still conforms to their original relative order following the distribution $\mathbf{P}_w$.

5.3.2 Feedback-aware Sampling

The simple rejection sampling scheme proposed above may work well when the amount of feedback is small. However, as the amount of feedback grows, the cost of this approach increases, as samples become more likely to be rejected. Thus a better sampling scheme should be "aware" of the feedback received, and try to avoid generating invalid samples, i.e., those that violate any provided feedback constraint.

Recall that user feedback produces a set of pairwise preferences of the form $p_1 \succ p_2$, where p_1 and p_2 are packages. Given a set S_ρ of these preferences, it can be shown that the set of valid weight vectors w, i.e., those that satisfy all feedback preferences, has the following useful property.

Lemma 9 *The set of valid weight vectors which satisfy a set of preferences S_ρ forms a convex set.*

Proof *By definition, for any w_1, w_2 which satisfy S_ρ, $\forall \rho := p_1 \succ p_2 \in S_\rho$, $w_1 \cdot p_1 \geq w_1 \cdot p_2$ and $w_2 \cdot p_1 \geq w_2 \cdot p_2$. Then $\forall \alpha \in [0, 1]$, $\alpha w_1 \cdot p_1 \geq \alpha w_1 \cdot p_2$ and $(1 - \alpha) w_2 \cdot p_1 \geq (1 - \alpha) w_2 \cdot p_2$. Combining these two inequalities shows that any convex combination of w_1 and w_2 also forms a valid w.*

So valid weight vectors form a continuous and convex region. By exploiting this property, we can leverage different sampling methods which can bias samples more towards those which are inside the valid region.

Importance Sampling

The general idea of *importance sampling* is that, instead of sampling from a complex probability distribution (*original distribution*), which in our case is $\mathbf{P}_w(w \mid S_\rho)$, we sample from a different *proposal distribution* $\mathbf{Q}_w$, which is more likely to satisfy the constraints given by the feedback set S_ρ. However, this process will introduce a set of samples which do not follow the original distribution, so we need to correct this *bias* by associating each sample w from $\mathbf{Q}_w$ with an importance weight (or simply weight, when there is no ambiguity) $q(w)$. Next we will discuss how this framework can be employed in solving our problem.

Since valid weight vectors w.r.t. S_ρ form a continuous and convex region, samples which lie close to the "center" of this region are more likely to satisfy S_ρ. However, finding the center of an arbitrary convex polytope is extremely complex and time consuming [26], which negates the motivation for using importance sampling, namely efficiency. Instead, we use a simple geometric decomposition-based approach, which partitions the space into a multi-dimensional grid, and approximates the center of the convex polytope using the centers of the grid cells which overlap with it.

In Figure 5.3, we show a simple two dimensional example of the above approach. Initially, the entire valid region is divided into a 3×3 grid as depicted in Figure 5.3(a). Given feedback $\rho := p_1 \succ p_2$, we know any invalid w satisfies the property that $w \cdot p_1 < w \cdot p_2$, or $w \cdot (p_2 - p_1) > 0$, i.e., w is invalid if w is above the line $p_2 - p_1 = 0$. As shown in Figure 5.3(b),

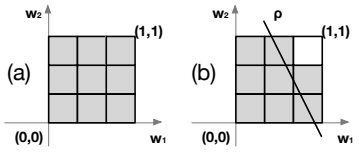


Figure 5.3: Approximate center of a convex polytope.

all w's which are in the top-right grid cell are above the line corresponding to ρ, so we can eliminate this cell from consideration, and the center of the region of valid w's can be approximated by the center of the remaining eight cells. The latter can be calculated by simply taking the average of the eight cell centers.

We note that whether there exists a w in a grid cell which satisfies a constraint ρ can be checked in time linear in the dimensionality of the feature space. Also, finding those cells which violate new feedback can be facilitated by organizing the grid cells into a hierarchical structure such as a *Quad-tree* [55].

Given the center w^* of the valid region, an intuitive choice for the proposal distribution would be a Gaussian $\mathbf{Q}_w \sim \mathcal{N}(w^*, \Sigma)$ with mean w^*, covariance Σ. To correct the bias introduced in samples from $\mathbf{Q}_w$, in importance sampling, we could associate each valid sample w with an *importance weight* $q(w) = \mathbf{P}_w(w)/\mathbf{Q}_w(w)$. Intuitively, this importance weight compensates for the difference between $\mathbf{P}_w$ and $\mathbf{Q}_w$.

The importance weight of each sample $q(w)$ can be easily adapted to different ranking semantics. For EXP, we multiply $q(w)$ by the utility value calculated for each package under consideration for this w. For TKP, MPO and ORA, instead of adding one to the corresponding counter of each package w.r.t. the given w, we add $q(w)$.

MCMC-based Sampling

Another popular approach for sampling from complex distributions is the *Markov Chain Monte Carlo* or MCMC method. This approach generates samples from the distribution by simulating a Markov chain. We construct the Markov chain in a way such that it gives more importance to the regions which are valid, i.e., the stationary distribution of the Markov chain is the same as the posterior $\mathbf{P}_w(w \mid S_\rho)$.

Since valid weight vectors form a single continuous and convex region, we could simply find a first valid weight vector w, and then perform a random

walk from w to a new weight vector w'. Note that in order to explore the valid region w.r.t. the current set of preferences S_ρ, it is clearly desirable that each step of the random walk explores only a close region around the current w, as otherwise, w' generated from w may be more likely to be outside the valid region. Thus we define a length threshold l_{max}, and set the transition probability $Q(w' \mid w)$ from w to w' as follows:

$$Q(w' \mid w) = \begin{cases} 1/l_{max} & \text{if } \|w' - w\| \leq l_{max} \\ 0 & \text{otherwise.} \end{cases} \tag{5.4}$$

One of the most popular MCMC-based sampling algorithms is Metropolis-Hastings (MH). Using MH we can generate samples following the Markov chain defined by $Q(w' \mid w)$. Given a current weight parameter w, we randomly pick a weight parameter w' such that the distance $\|w' - w\|$ is less than or equal to l_{max}. If w' satisfies the preferences in the feedback set S_ρ, w' is accepted as the next sample with a probability α, as defined in Equation 5.5. If w' is rejected (i.e., with probability $(1 - \alpha)$), we use a copy of the w as the next sample.

$$\alpha = \min\{1, \frac{\mathbf{P}_w(w')Q(w \mid w')}{\mathbf{P}_w(w)Q(w' \mid w)}\} \tag{5.5}$$

Note that $Q(w' \mid w)$ is obviously symmetric in our case, so α can be simply calculated as $\alpha = \min\{1, \frac{\mathbf{P}_w(w')}{\mathbf{P}_w(w)}\}$.

Following recommendations in the MH sampling literature [22], we pick one sample from every δ samples generated, where δ is called the *step length*, rather than including all generated samples in the final sample pool. This avoids generating highly correlated samples.

5.3.3 Optimizing Constraint Checking Process

Suppose σ packages $p_1, \ldots, p_\sigma$ are presented to the user. Note that according to the preference elicitation protocol, in every interaction only one package is picked by the user as the most preferred package. Without loss of generality, let p_1 be that package. This results in $\sigma - 1$ pairwise package preferences: $\rho_1 := p_1 \succ p_2, \ldots, \rho_{\sigma-1} := p_1 \succ p_{\sigma-1}$. Thus, as more feedback is received from the user, the number of package preferences we need to deal with increases quickly.

This raises the following two issues: (1) cycles in preferences, and (2) cost of checking whether a sample w should be rejected. We note that cycles in

preferences can be resolved by presenting packages in a cycle to the user, and asking the user to choose the best out of them (which reverses the direction of one edge in the cycle and breaks the cycle). Next we will discuss how a set S_ρ of pairwise package preferences can be organized in order to facilitate efficient checking of whether a sampled weight vector w is valid.

One intuitive solution is to reduce redundant package preferences by exploiting the *transitivity* of the preference relation. It is easy to see that the preference relation $\succ$ over packages is transitive for additive utility functions, i.e., for any packages p_1, p_2, p_3 and any weight vector w, $w \cdot p_1 > w \cdot p_2$ and $w \cdot p_2 > w \cdot p_3$ imply $w \cdot p_1 > w \cdot p_3$. Thus, there is no need to verify satisfaction of $p_1 \succ p_3$ for a sample w whenever w satisfies $p_1 \succ p_2$ and $p_2 \succ p_3$. It follows that the number of preferences we need to check is at most linear in the number of distinct packages (implicitly) appearing in the feedback.

To eliminate redundant preferences received from the user, we can maintain preferences in a *directed acyclic graph* (DAG) G_ρ: an edge (p_i, p_j) represents the preference $p_i \succ p_j$. Then any *transitive reduction* algorithm [17] can be applied to eliminate redundant preferences.

5.3.4 Sample Maintenance

It is clear from the previous section that depending on the number of feedback preferences received from a given user, the sampling process may actually be quite time consuming. Thus, it is desirable to avoid generating samples from scratch whenever new feedback is received. In other words, it is desirable to *maintain* previously generated samples against new incoming feedback.

Given a probability distribution $\mathbf{P}_w$ of w, a set S_ρ of preferences, and a sample pool S, instead of regenerating all samples, we can simply replace samples which violate the new feedback, and retain samples which do not violate any new feedback. This approach works since the probability of each valid w always follows $\mathbf{P}_w$, regardless of the newly received feedback.

A simple idea for replacing invalid samples in the pool is to scan through all samples in the pool one by one, and check whether each satisfies the new feedback. This simple approach will be effective if many samples might violate the new feedback received. However, the performance will be very poor if hardly any samples from S actually violate the newly received feedback.

Note that, as discussed in the previous section, for feedback $\rho := p_1 \succ p_2$, w violates ρ if $w \cdot (p_2 - p_1) > 0$. Thus finding those $w \in S$ which violate ρ is the same as finding all weight vectors which have a projected value on

$p_2 - p_1$ larger than 0. This problem can be solved by leveraging the classic threshold algorithm (TA) framework [63], by iteratively enumerating the largest w from S w.r.t. the query vector $p_2 - p_1$ until the maximum possible score of any unseen w is less than or equal to 0. Obviously this TA-based algorithm is very efficient when not many samples violate the new preference. However, for cases where most samples violate the new feedback, the cost of the TA-based algorithm may be much higher than the naïve algorithm of checking every sample in the pool for possible preference violation.

Algorithm 14: RejectedSampleCheck(S, $\rho = p_1 \succ p_2$)

```
1  Q ← An empty set for rejected sample w's;
2  L^w ← Lists of samples sorted based on feature values;
3  while true do
4      l^w ← Access lists in L^w in round-robin fashion;
5      w ← getNext(l^w);
6      τ ← boundary value vector from L^w;
7      if w · (p2 − p1) > 0 then Add w to Q;
8      if τ · (p2 − p1) ≤ 0 then Break;
9      if C_processed + C_remain ≥ (1 + γ)|S| then
10         Scan and check each remaining w in l^w, Break;
11 return Q;
```

Motivated by this, we propose a hybrid approach shown in Algorithm 14. We organize the samples into m lists $\mathcal{L}^w = l_1^w, \ldots, l_m^w$, where each list l_i^w is a total ordering of items based on the values of the corresponding feature f_i. Given new feedback ρ, we start with the TA-based algorithm, and if the current number $\mathcal{C}_{processed}$ of items processed plus the number $\mathcal{C}_{remain}$ of remaining items in the current list is larger than or equal to $(1 + \gamma)$ of the total number of items, we stop the TA process, and instead scan through the remainder of the current list, checking the validity of each sample within this list. Here, γ is a parameter which can be tuned based on the actual performance, with smaller γ leading to performance closer to the simple scanning algorithm, and larger γ leading to performance closer to the pure TA-based algorithm.

5.4 Search for Best Packages

If we have a top-k package solver which can produce the top-k packages for a given weight vector w, then given a set S of sample weight vectors, we can

easily find the overall top-k packages under different ranking semantics as follows:

For EXP, we need to estimate the expected utility of packages and return the top-k packages w.r.t. the estimates. Given all the top-k package results obtained from the samples $w \in S$, we maintain the sum of the utility values for each package appearing in the results. Then the sample utility mean of each package is simply the utility sum divided by the number of times the package appears in a result. Note that we only need to consider those packages which appear in the top-k package list w.r.t. at least one sample w.

For TKP, we just need to maintain a counter for each package which appears in the result set, and the k packages which appear most frequently in this set will be the result under TKP.

For MPO, instead of maintaining statistics for each package that appears in the result set, we maintain a counter for each top-k package list. The final top-k package list under MPO is the one with the largest counter value.

For ORA, the final top-k package list should be the one which minimizes the sum of its (Kendall tau) distance to any top-k package list in the result set. So we can simply store all the top-k package lists and then find the final top-k packages under ORA using any available algorithm [112].

Thus, given a sampling-based framework, a key step in finding the top-k packages given a set S of sample weight vectors is to find the top-k packages for any specific w, which we address next.

Given a set T of items and a fixed w for the utility function, the problem of getting the k best *items* w.r.t. w can be done using any standard top-k query processing technique [63]. However, because we are dealing with subsets of items, the problem becomes challenging since a naïve solution which first enumerates all possible packages, and then uses a top-k query processing algorithm for finding the best k packages would be prohibitively expensive. Below, we discuss how classical top-k query processing algorithms can be adapted to finding the top-k *packages*, given a fixed w.

Following recent research on top-k item query processing [63], one intuition is that for a top-k package p, the likelihood of having a high utility item in p is often higher than the likelihood of having a low utility item in p. Thus by accessing items in their descending utility order, we could potentially locate the top-k packages by accessing only a small number of items. To facilitate efficient processing over different weight vectors, we order items based on their utility w.r.t. each individual feature. We denote the resulting set of sorted lists by $\mathcal{L}$.

Given a fixed w and utility function U, Algorithm 15 gives the pseudo-

code of the overall algorithm framework TopPkg. As shown in the algorithm, TopPkg first sorts the underlying items into different lists, where each list is an ordering of the items in T w.r.t. the desirable order on one specific feature according to the utility function (line 2). E.g., consider $U(p) = 0.5\mathsf{avg}_1(p) + 0.5(1 - \mathsf{sum}_2(p))$, where $\mathsf{avg}_1(p)$ and $\mathsf{sum}_2(p)$ are the normalized aggregation values of the package w.r.t. the corresponding feature. The algorithm sorts items into two lists, l_1 and l_2, where in l_1, items are sorted in non-increasing order of feature 1, and in l_2, items are sorted non-decreasing order of feature 2 [20]. As discussed before, the intuition is that by accessing items with better utility values w.r.t. each individual feature in the utility function, we can potentially quickly find the top-k packages of the items.

After constructing the set of lists $\mathcal{L}$, TopPkg accesses items from lists in $\mathcal{L}$ in a round-robin fashion (line 4–5). We assume items reside in memory, so their feature values can be retrieved quickly. After accessing each new item t, we can obtain the new boundary value vector τ in which each feature value equals the corresponding feature value of the last accessed item in each list (line 6). So essentially, the feature vector τ corresponds to the maximum possible utility value for an unaccessed item.

Next, we can expand the existing packages in the queue Q by incorporating the new item t (line 7), a process described in Section 5.4.2. During package expansion, a current lower bound utility threshold η^{lo} can be obtained by looking at the kth best package so far in the queue Q, and an upper bound utility threshold η^{up} of any possible package can be obtained by referring to the maximum utility value an unaccessed item can have, a calculation described in Section 5.4.1. Obviously, if $\eta^{up} \leq \eta^{lo}$, we can safely return the current top-k packages, as no future packages can have higher utility than the current top-k packages (line 8–9).

5.4.1 Upper Bound Estimation for Best Package

Given the accessed item information, one important problem in TopPkg is the estimation of the upper bound value a package can have. In this section, we will discuss algorithms for estimating this upper bound value.

Given a fixed weight vector w, the utility value $U(p)$ of a specific package p can be calculated as $p \cdot w$, so it depends only on items within the package p. Given the fact that items in each list of $\mathcal{L}$ are ordered in non-increasing utility of the corresponding feature, the maximal marginal utility value of

[20] Since each sorted list can be accessed both forwards and backwards, there is no need to maintain two separate lists when different desirable orders are required on the same feature.

Algorithm 15: TopPkg(U, T, w, k)

```
1 Q ← A priority queue of packages having one item ∅;
2 L ← Lists of items in T sorted according to util. func. U;
3 while true do
4     l ← Access lists in L in round-robin fashion;
5     t ← getNext(l);
6     τ ← boundary value vector from L;
7     (η^lo, η^up) ← expandPackages(U, Q, t, τ));
8     if η^up ≤ η^lo then break ;
9 return top-k packages in Q;
```

an unseen item is obviously bounded by the imaginary item with feature vector τ.

Given a utility function U, we say that U is *set-monotone* if for any packages p, p' of items, we have $U(p \cup p') \geq U(p)$. E.g., $U(p) = 0.5\mathsf{sum}_1(s) + 0.5(1-\mathsf{min}_2(s))$ is set-monotone. Clearly, if U is set-monotone, the maximum utility of a package p can be achieved by packing as many items with feature vector τ (were they to exist) as possible into p. On the other hand, if U is not set-monotone, e.g., when some aggregate feature values in U are based on avg, we can show that the upper bound value of p in this case is given by packing as many items with feature vector τ into p as possible, as long as the marginal utility gain of this addition is positive.

Lemma 10 *Given a package p, a utility function U with fixed w, and a sequence of items $t_1, \ldots, t_m$ such that every feature value of t_i is no worse than that of t_{i+1}, then $U(p \cup \{t_1, \ldots, t_i\}) - U(p \cup \{t_1, \ldots, t_{i-1}\}) \geq U(p \cup \{t_1, \ldots, t_{i+1}\}) - U(p \cup \{t_1, \ldots, t_i\})$, $1 < i < m$.*

Proof *The result follows from that fact that every feature value of t_i is no worse than that of t_{i+1} w.r.t. U.*

An algorithm for estimating the upper bound value, upper-exp, is shown as Algorithm 16, where ϕ is the maximum allowed package size.

5.4.2 Package Expansion

Consider the problem of expanding the set of packages in queue Q on accessing a new item t. A naïve way of expanding packages would be to try to add t to every possible package in Q as long as the resulting package satisfies the package size budget, inserting the new packages into Q. Obviously

Algorithm 16: upper-exp(p, U, τ, ϕ)

```
1 p' ← p;
2 if U is set-monotone then
3     for i ∈ [1, φ − |p|] do p' ← p' ∪ {τ} ;
4     return U(p')
5 else
6     for i ∈ [1, φ − |p|] do
7         if U(p' ∪ {τ}) − U(p') > 0 then p' ← p' ∪ {τ} ;
8         else return U(p');
9     return U(p')
```

utilizing this strategy is equivalent to enumerating all possible combinations of the accessed items. Thus, while it is guaranteed to be correct, it is highly inefficient.

Given a package p, one intuitive optimization is that if incorporating any unaccessed item cannot improve the value of p, we do not need to consider p in the expansion phase. E.g., let $U(p) = 0.5\mathsf{avg}_1(p) + 0.5\mathsf{min}_2(p)$, with $p = (0.5, 0.5)$ and $\tau = (0.4, 0.4)$. Clearly, any unaccessed item in $\mathcal{L}$ will have a utility worse than or equal to that of τ, so there is no need to consider p for expansion in the future.

To incorporate this optimization, we split the priority queue Q in TopPkg into two sub-queues Q_+ and Q_-. Queue Q_+ stores packages which can be further expanded (while improving utility), while Q_- stores packages which cannot be further expanded (while improving utility) and so can be pruned from the expansion phase. In Algorithm 17 for the expansion phase, we only need to iterate through packages in Q_+ (lines 2–12), and for each package $p \in Q_+$, we test whether p can be improved by incorporating the new item t (line 3). If true, we generate a new package p', and insert it into the appropriate sub-queue based on whether it can be further improved by an unaccessed item or not (lines 5–8). If false, we can check whether the current p can be further improved by referring to the updated τ, and p will be moved to Q_- if it cannot be improved (lines 9–11).

5.5 Experimental Evaluation

In this section, we study the performance of various algorithms proposed in this work based on one real dataset of NBA statistics and four synthetic datasets. The goals of our experiments are to study: (i) the performance

Algorithm 17: expandPackages(U, Q, t, τ)

```
1  (η^lo, η^up) ← lower/upper bound value;
2  foreach p ∈ Q_+ do
3      if U(p ∪ {t}) > U(p) then
4          p' ← p ∪ {t};
5          if U(p' ∪ {τ}) > U(p')) then
6              Q_+ ← Q_+ ∪ {p'};
7              η^up ← max(η^up, upper-exp(p', U, τ, φ));
8          else Q_− ← Q_− ∪ {p'} ;
9      if U(p ∪ {τ}) > U(p) then
10         η^up ← max(η^up, upper-exp(p, U, τ, φ));
11     else Q_+ ← Q_+ − {p}, Q_− ← Q_− ∪ {p} ;
12 η^lo ← U(Q_−[k]) or 0 if fewer than k packages in Q_−;
13 return (η^lo, η^up)
```

of various sampling techniques w.r.t. our package recommendation problem; (ii) the effectiveness of the proposed pruning process; (iii) the performance of various maintenance algorithms as the system receives new feedback. We implemented all the algorithms in Python, and all experiments were run on a Linux machine with a 4 Core Intel Xeon CPU, OpenSuSE 12.1, and Python 2.7.2.

The NBA dataset is collected from the Basketball Statistics website [6], which contains the career statistics of NBA players until 2009. The composite recommendation would be to recommend a set of NBA players which have good aggregated statistics over different measures, e.g., high average 3 points per game, high rebounds per game. The dataset has 3705 NBA players and we randomly selected 10 (out of 17) features (each feature corresponds to statistics of NBA players such as points per game) to be used in our experiments. The synthetic datasets are generated by adapting the benchmark generator proposed in [25]. The *uniform* (UNI) dataset and the *powerlaw* (PWR) dataset are generated by considering each feature independently. For UNI, feature values are sampled from a uniform distribution, and for PWR, feature values are sampled from a power law distribution with $\alpha = 2.5$ and normalized into the range $[0, 1]$. In the *correlated* (COR) synthetic dataset, values from different features are correlated with each other, while in the *anti-correlated* (ANT) synthetic dataset, values from different features are anti-correlated with each other. Each synthetic dataset has 10 features and has 100,000 tuples.

5.5.1 Comparing Sampling Methods

In Figure 5.4, we show an example of how different sampling methods generate 100 valid 2-dimensional sample w parameters given 5000 packages and 2 randomly generated preferences. As discussed previously, each preference $\rho := p_1 \succ p_2$ defines a linear hyperplane. A sample w satisfies ρ iff $w \cdot (p_1 - p_2) \geq 0$, or w is above the corresponding hyperplane. In Figure 5.4 (a), given the set of valid sample w's (black dots) and the set of invalid sample w's (red crosses), we can infer the two linear hyperplanes which correspond to the two given preferences and bound valid sample w's to the bottom. It is clear from the figure that unless these two preferences are way "above" the center of $\mathbf{P}_w$, many sample w's from $\mathbf{P}_w$ will be invalid. Thus using rejection sampling, many samples will be wasted and we need to spend considerable time checking whether each sample w satisfies all preference constraints received.

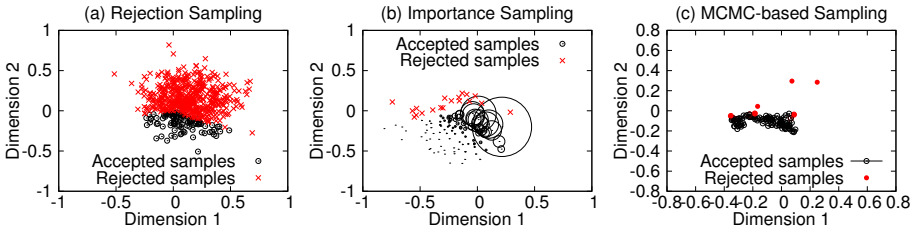


Figure 5.4: Example of various sampling algorithms.

On the other hand, the two feedback-aware sampling strategies will generate far fewer invalid samples. E.g., in Figure 5.4 (b), the importance sampling technique samples from a proposal distribution which is more to the center of the valid region, so samples generated are more likely to satisfy all constraints. Notice, each sample w is also associated with a weight, which is captured by the size of the dot/circle in the figure. The higher the probability w has under the original distribution, and the lower the probability w has under the proposal distribution, the larger the weight of w.

As can be seen from Figure 5.4 (c), MCMC-based sampling first needs to find one random valid sample w. Note that during this process we leverage the simple rejection sampling, thus these rejected samples (denoted as isolated red crosses in the figure) will not be part of the random walk process in MCMC. Then from this valid sample w, we initiate a random walk from the neighborhood of w, which follows the original distribution of $\mathbf{P}_w$ using a Metropolis-Hastings sampler as discussed Section 5.3.2.

5.5.2 Constraint Checking

As discussed in Section 5.3.1, no matter which sampling method we use, an important task is to efficiently check whether a sample satisfies all the feedback constraints received from a user. In Figure 5.5, we show how pruning strategies discussed in Section 6.4.2 benefit the overall checking performance by varying the number of features, the number of samples, and the number of Gaussians in the mixture distribution while keeping the other variables fixed at a default value. We set the default value for the number of randomly generated preferences to 10000, the number of packages to 5000, the number of Gaussians to 1, the number of features to 5, and the number of samples to 1000. As can be seen from this figure, when we vary one parameter while fixing other parameters at their default values, the pruning strategy can robustly generate at least a 10% improvement. Results under other different default values are similar.

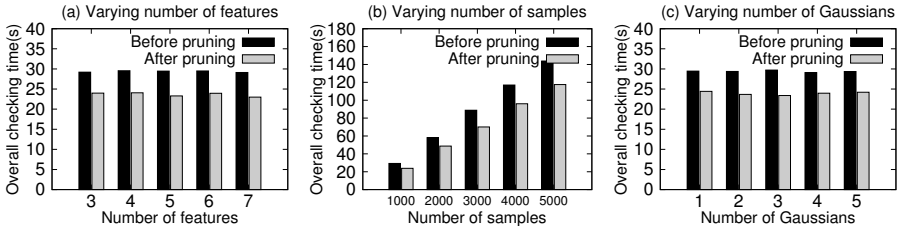


Figure 5.5: Efficiency of the pruning strategy.

5.5.3 Overall Time Performance

In this section, we report the overall time performance for package recommendation over different datasets. All time results reported are based on an average of 5 runs.

In Figure 5.6, we compare overall time performance for generating top-k package recommendations under Rejection Sampling (RS), Importance Sampling (IS), and MCMC-based Sampling (MS). In these experiments, we randomly select one ranking semantic and vary one of the following two parameters while fixing the remaining parameters at their default values: (1) Number of valid samples required; (2) Number of features. We also tested varying the number of feedback preferences received, and the number of Gaussians in the mixture distribution; the results are very similar to varying the number of valid samples, and thus are omitted.

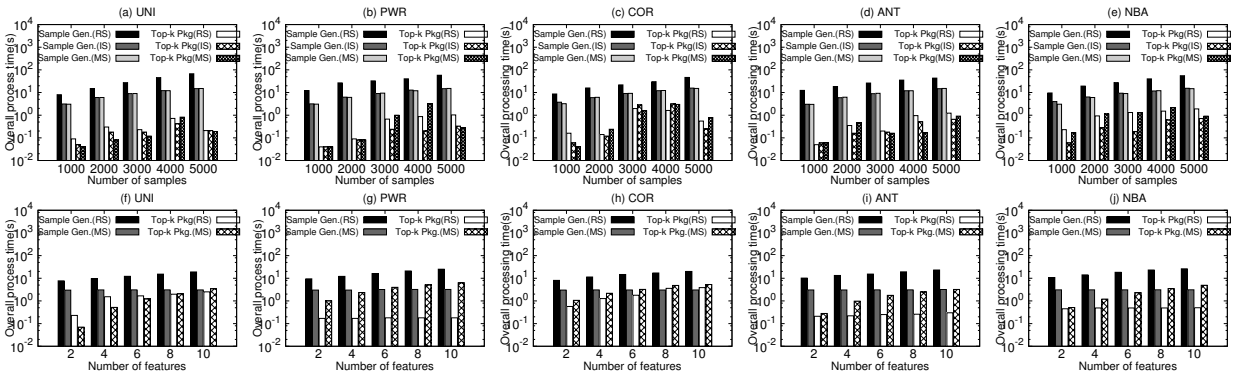


Figure 5.6: Overall time performance under various sampling algorithms.

From Figure 5.6 (a)–(e), with a log-scale for processing time, we observe that the sampling cost for generating valid sample w's mostly outweighs or at least is comparable to the cost for generating top-k packages, as usually the top-k packages can be found by just checking the first few high utility items. Also the rejection sampling cost is usually much higher than that of the other two feedback-aware sampling approaches.

As can be seen from Figure 5.6 (f)–(j), importance sampling is excluded from high dimensional experiments because finding the center of a high-dimensional polytope is computationally intractable [46]. Even using the simple grid-based approximation algorithm as discussed in Section 5.3.2, the cost is exponential w.r.t. the dimensionality. Specifically, when the dimensionality is over 5, the time to find the center will quickly exceed the time for rejection sampling. Indeed, when dimensionality is 6, the algorithm cannot finish within 30 minutes whereas simple rejection sampling only needs a couple of seconds. As can be seen from Figure 5.6 (f)–(j), MCMC-based sampling scales well w.r.t. dimensionality.

5.5.4 Sample Quality

To measure the quality of different sampling methods, we compare the top-5 package list generated w.r.t. different ranking semantics and different sampling methods. In our experiments, we set the number of samples to 5000 (we verified that increasing the number of samples beyond 5000 does not change the top package rankings for different datasets), the number of feedback preferences received to 1000, the number of features to 4, and the number of Gaussians in the mixture distribution to 2. Results under dif-

ferent value settings are similar thus excluded. Each package is associated with a unique and random package id.

In Table 5.1, we show the top-5 packages under different sampling methods and different ranking semantics on UNI; experimental results on other datasets and other settings are similar and are thus omitted. As can be seen, given enough samples, the top package results from different sampling methods typically tend to become very similar. The reason is that although different ranking semantics may potentially result in different top package lists, they can be correlated with each other. E.g., as in our example, if one list of top packages dominates the results given a set of samples, TKP, MPO and ORA may tend to give very similar results. This is because picking the same list of top packages guarantees that packages in this list may also appear most frequently among all top packages. ORA may also pick this same list as it tends to minimize the distance between it and all other top package lists. EXP may not be affected by this as it is determined by the expected utility of the package, so a package appearing frequently may not necessarily have high expected utility value.

	Rej. Sampling	Imp. Sampling	MCMC Sampling
EXP	1,2,3,6,7	1,2,3,6,7	1,2,3,6,7
TKP	6,7,8,9,10	6,7,8,9,10	6,7,8,9,10
MPO	6,7,8,9,10	6,7,8,9,10	6,7,8,9,10
ORA	6,7,8,9,10	6,7,8,9,10	6,7,8,9,10

Table 5.1: Top-5 package id's for different sampling methods and different ranking methods on UNI.

5.5.5 Sample Maintenance

As discussed in Section 5.3.4, upon receiving new feedback, a naïve method of scanning through previous samples to determine which samples need to be replaced might be costly if the number of rejected samples is low, whereas a top-k algorithm might help by quickly scanning through the pre-processed sample lists, and determining whether all samples satisfy the constraints. However, this algorithm suffers from a substantial overhead when the number of rejected samples is large. Thus we propose a hybrid method which starts following the top-k based approach, then falls back to the default naïve method if the top-k process cannot stop early.

To assess the actual performance of these three algorithms, we consider in the following experiment a setting where the number of previously gener-

ated samples is set to 10000 (results using other values are similar and thus omitted), every other parameter is fixed at a default value similar to previous experiments. We randomly generate sets of 1000 feedback preferences, and then according to the number of samples rejected w.r.t. the feedback, we group the maintenance costs into 7 buckets, where each bucket is associated with a label indicating the maximum number of samples rejected (see Figure 5.7 (a)). Results are placed in the bucket with the smallest qualifying label. Maintenance cost results are averaged for all cases within the same bucket.

According to Figure 5.7 (a), the top-k based algorithm is a clear winner when the number of rejected samples is small. As the number of rejected samples grows, the performance of top-k based algorithms will deteriorate, especially the non-hybrid method. But the hybrid method introduces only a small overhead over the naïve algorithm because of the fall-back mechanism, and this overhead can be tuned through the parameter γ. In Figure 5.7 (b), we show how the ratio of each of top-k cost and hybrid cost over the naïve approach varies with γ. It shows that when γ is very small, the average performance of the hybrid method is very similar to the naïve algorithm as the algorithm is forced to check fewer samples. By slightly increasing γ (e.g., to 0.025 as in Figure 5.7 (b)), the hybrid method can show over 15% improvement compared to the naïve method. When γ increases further, the performance deteriorates as it becomes similar to the non-hybrid method. We note that this property means we could adaptively decrease γ in practice until a reasonable performance gain can be observed.

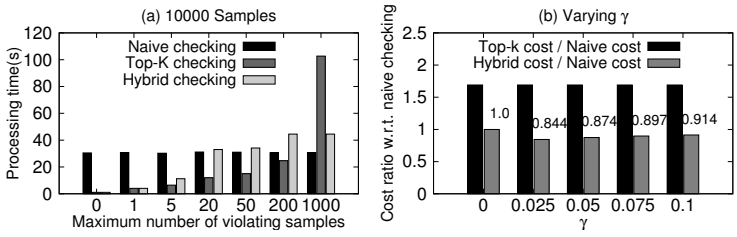


Figure 5.7: Experiments on sample maintenance.

5.6 Discussion

A user's online interaction can be noisy. E.g., a user may accidentally click on a package or may change their mind after clicking. A popular method for

modeling this kind of noisy user feedback is to assume that every feedback received has a probability ψ of being "correct" [27]. We can incorporate this noise model into our framework by assuming that every feedback is independent. Then instead of rejecting a sample whenever it violates some feedback preference, we condition its rejection using the probability that at least one violated feedback preference is correct, i.e., $1 - (1 - \psi)^x$, where x is the number of feedbacks w violates. This can be easily incorporated into importance sampling and MCMC-based sampling.

As discussed in [129], users may sometimes specify predicates on the schema of a desired package, e.g., when buying a set of books, at least two should be novels. We can handle such predicates in the top package generation process discussed in Section 5.4. The idea is that when generating a new candidate package, we evaluate the predicates and retain this package only if it satisfy the specified predicates.

As future work, we plan to investigate how TopPkg presented in Section 5.4 can be further optimized using *domination-based pruning* [87, 129]. The intuition is that by pruning away candidate packages which are not promising, we can further reduce Q, which will be iteratively searched in the expansion phase. However, we note that usually these pruning strategies also come with a non-trivial computational cost, so a systematic cost-based study of different pruning strategies under our proposed PRS model would be interesting.

Chapter 6

Further Optimization using Materialized Views

6.1 Introduction

As discussed in the previous chapters, no matter under hard constraint or soft constraint, the key algorithm in our composite recommendation system can be regarded as a special case of the general top-k query processing problem. While there exists various methods which can optimize the performance of top-k query processing, here in this chapter we discuss how a particular simple and easy to be implemented technique, *query answering using cached views*, can be used to facilitate our top-k query processing problems. while most of this chapter focuses on discussing how this technique can be leveraged to facilitate classical top-k query processing w.r.t. items. At the end of this chapter, we show how this technique can be applied in different top-k query processing algorithms in composite recommendation.

For the applications under consideration w.r.t. items, typically a simple linear score function is used to aggregate the attribute values of a tuple into a score, due to its intuitiveness [36, 44, 45, 60, 117, 128]. Figure 6.1 (a) shows an example relation R which contains 6 tuples over attributes A, B and C. Consider a query Q_1 which asks for the top-3 tuples with the highest values under the score function $f_1(t) = 0.1t[A]+0.9t[B]$. The result is shown as (a cached view) V_1 in Figure 6.1 (b).

While various efficient algorithms have been proposed for processing top-k queries [53, 63], one significant limitation is that they cannot take advantage of the cached results of previous queries. E.g., consider the previous example query Q_1 whose result V_1 is shown in Figure 6.1 (b). Suppose a (possibly different) user subsequently asks the top-1 query Q' with the score function $f'(t) = 0.2t[A] + 0.8t[B]$. Then, as we will see in later sections, we can use the previous cached result V_1 to determine, *without accessing the original database* R, that t_5 is the top-1 answer for Q'.

Leveraging cached query results to scale up query answering has recently

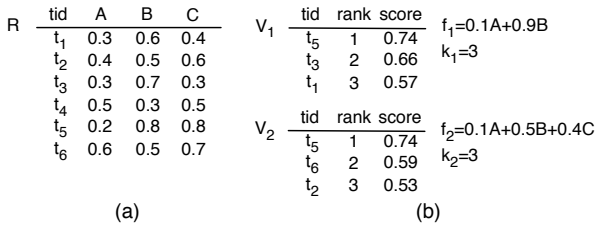


Figure 6.1: (a) A relation R with three attributes A, B, C; (b), two cached views V_1, V_2 which contain top-3 tuples according to the the two score functions $f_1(t) = 0.1t[A] + 0.9t[B]$, $f_2(t) = 0.1[A] + 0.5[B] + 0.4[C]$ respectively.

become increasingly popular for most large scale websites. For example, the popular Memcached [3] caching system has reportedly been adopted by many large scale websites such as Wikipedia [10], Flickr [1], Twitter [8] and Youtube [12]. The application of cached query results or materialized views for speeding up query answering in relational databases, the so-called query answering using views (QAV) problem, has been extensively studied (see [58] for an excellent survey). This problem has been shown to have applications in data integration [75], query optimization [84], and data warehouse design [114].

For top-k query processing, recently there have been some initial efforts at using materialized query results for speeding up query answering. In the PREFER system, Hristidis et al. [60] consider the problem of how to select one best materialized view for answering a query. Their setting is quite restrictive, as it cannot exploit multiple materialized views, and it also makes a strong assumption that all attributes of the underlying base table are always utilized for all top-k queries. Overcoming these limitations, Das et al. [45] propose a novel algorithm, called LPTA, which is able to utilize multiple materialized views for answering a top-k query. Ryeng et al. [109] extend these techniques to answer top-k queries in a distributed setting.

Though LPTA overcomes many of the limitations of PREFER, unfortunately it still suffers from several significant limitations. Firstly, the core techniques proposed in [45] rely on the assumption that either (1) each top-k view is a *complete ranking of all tuples in the database*, or (2) that the *base views*, which are *complete rankings of all tuples in the database according to the values of each attribute, are available*. These assumptions may often be unrealistic in practice.

Consider the example of finding top-k movies. There are several popular

websites which provide top-k lists of movies based on different criteria. For example, Metacritics [4] provides a ranked list of (up to 5639) movies based on *Metascore* [5], which is aggregated from critics and publications like the New York Times (NYT) and the San Francisco Chronicle; IMDB [2] provides a top-250 list of movies based on votes from their users; and RottenTomatoes (RT) [7] provides a top-100 list of movies based on the *Tomatometer* score, which is calculated based on critics. Here, the top-k lists on Metacritics, IMDB, and RT can be regarded as materialized views. Because of the huge number of movies available, it is impractical to obtain the complete ranking of all movies from each of the sources, and for the same reason, we cannot assume base views corresponding to the complete ranking of all movies on each of the individual scores, e.g, NYT score, are available. Consider the query that asks for top-k movies according to an aggregation of NYT score, IMDB score, and Tomatometer score. Since the only information we have access to is the top-k movies from the Metacritics, IMDB, and RT, *the technique proposed in [45] cannot be used to answer this query*. Similar examples can also be found in other domains: finding the top-k universities based on university ranking lists from U.S. News [9] and The Times [11]; or finding the top-k cars based on automobile ranking lists from U.S. News [14] and Auto123 [13].

The second issue with the LPTA algorithm proposed in [45] is that it uses linear programming (LP) as a sub-procedure to calculate the upper bound on the maximum value achievable by a candidate result tuple, and the LPTA algorithm needs to call this sub-procedure iteratively. It has been demonstrated in [45] that for low dimensionality scenarios (e.g., 2 or 3), the cost of this LP overhead is reasonable. However, we will show in our experiments that for scenarios with higher dimensionality, which we note is very common, this iterative invocation of the LP sub-procedure may incur a high computational overhead.

Finally, for both PREFER [60] and LPTA [45], a potentially costly view selection operation is necessary. For example, the view selection algorithm in [45] requires the simulation of the top-k query process over the histograms of each attribute, and the processing cost is linear with respect to the number of views. This cost can be prohibitive given a large pool of cached query (view) results. Furthermore, (histograms over) base views are often not available in practice, restricting the applicability of this view selection procedure.

In this chapter, we propose two novel algorithms for the problem of top-k query answering using cached views. Our first algorithm LPTA$^+$ is an extension of LPTA as proposed in [45]. In LPTA$^+$, we make a novel observation on the characteristics of LPTA, and by taking advantage of the

fact that our views are cached in memory, we can usually avoid the iterative calling of the LP sub-procedure, thus greatly improving the efficiency over the LPTA algorithm. LPTA$^+$ can be useful for scenarios with a small number of views and low dimensionality. For the case where the number of cached views is large and the dimensionality is high, we further propose an index structure called the *inverted view index* (*IV-Index*), which stores the contents of all cached views in a central data structure in memory, and can be leveraged to answer a new top-k query efficiently *without any need for view selection*.

Specifically, we make the following contributions:

- We consider the general problem of top-k query answering using views, where base views are not available, and the cached views include only the top-k tuples which need not cover the whole view (Section 6.2).
- For scenarios where we are not allowed to maintain additional data structures, we extend LPTA and propose a new algorithm, LPTA$^+$, which can significantly improve performance over LPTA (Section 6.3).
- We further propose a novel index-based algorithm, IV-Search, which leverages standard space-partitioning index structures, and can be much faster than LPTA/LPTA$^+$ in most situations. We consider two different strategies for the IV-Search algorithm, and discuss additional optimization techniques (Section 6.4).
- We present a detailed set of experiments showing that the performance of our proposed algorithms can be orders of magnitude better than the state-of-the-art algorithms (Section 6.5).

Related work is discussed in Section 6.6.

6.2 Problem Setting

Given a schema $\mathbf{R}$ with m numeric attributes $A_1, \ldots, A_m$, we denote a relation instance of $\mathbf{R}$ by R. In practice, $\mathbf{R}$ may have other non-numeric attributes as well, but we are concerned only with the numeric attributes. Every tuple $t \in R$ is an m-dimensional vector $t = (t[1], \ldots, t[m])$, where $t[i]$ denotes the value of t on attribute A_i, $i = 1, \ldots, m$. Similar to previous work on top-k query processing, we assume that attribute values are normalized in the range of $[0, 1]$, and that each tuple t also has a unique tuple id.

Similar to [45], we define a top-k query Q over $\mathbf{R}$ as a pair (f, k), where k is the number of tuples required, and $f : [0,1]^m \to [0,1]$ is a linear function which maps the m attribute values of a tuple t to a preference score, i.e., $f(t) = w_1 t[1] + \cdots + w_m t[m]$, where $w_i \in [0,1]$, and $\sum_i w_i = 1$. Note that since every w_i is non-negative, the function f is clearly *monotone*, i.e., for two tuples t_1 and t_2, if $t_1[i] \geq t_2[i]$, $i = 1, \ldots, m$, then $f(t_1) \geq f(t_2)$.

Given a relation R and a query $Q = (f, k)$, without loss of generality, assume that $k \leq |R|$ and that larger f values are preferred. Then the semantics of top-k query processing is to find k tuples in R which have the largest values according to the query score function f. We can formally define the answer to a top-k query as follows.

Definition 14 Top-k Query Answer: *Let $Q = (f, k)$ be a top-k query over relational schema $\mathbf{R}$. Given a relation R over $\mathbf{R}$, the answer of Q on R, $Q(R)$, is a list of tuples from R such that $|Q(R)| = k$, and $\forall t \in Q(R)$ and $\forall t' \in R \backslash Q(R)$, $f(t) \geq f(t')$. Finally, tuples of higher rank in $Q(R)$ have a higher score according to the score function f.*

A *top-k cached view*, or a top-k view for brevity, is defined similarly to a top-k query, except the results of a top-k view are cached in memory. For each tuple t in a cached view, we assume all attribute values $t[i]$, $i = 1, \ldots, m$, will also be cached in memory, and thus can be efficiently accessed at query time. We allow random access by id on the cached tuples. Given a view $V_i = (f_i, k_i)$, without any ambiguity, we reuse V_i also to denote the list of (k_i) tuples materialized along with their ranks and scores w.r.t. f_i.

We use $V_i[j]$ to denote the tuple $t \in V_i$, which has the jth highest score w.r.t. f_i, with ties broken using tuple id, i.e., a tuple with a smaller tuple id will be ranked higher. Similarly for a given relation R, we denote the jth tuple in R following the order defined by a score function f as $R_f[j]$.

Let $\mathcal{V} = \{V_1, \ldots, V_p\}$ be a set of views, where $V_i = (f_i, k_i)$ is a top-k_i view, and let $Q = (f, k)$ be a top-k query. Inspired by the notion of *certain answers* when answering a non-ranking query using views [15], we say a relation R is *score consistent* with the set $\mathcal{V}$ of views, if for any view $V_i = (f_i, k_i) \in \mathcal{V}$, the jth tuple $R_{f_i}[j]$ in R w.r.t. f_i has the same score as the jth tuple $V_i[j]$ in V_i, i.e., $f_i(R_{f_i}[j]) = f_i(V_i[j])$, for $j = 1, \ldots, k_i$. Note that we do *not* require $R_{f_i}[j]$ to have the same tuple id as $V_i[j]$, since the score of a tuple is determined solely by its attribute values and not by its tuple id (Definition 14).

Given a set of views $\mathcal{V} = \{V_1, \ldots, V_p\}$, a score consistent relation R is the counterpart of a possible world under the closed world assumption (see [15]).

Accordingly, we define a tuple $t \in V_i$, $i = 1, \ldots, p$, to be a *certain answer* to Q if, for any relation R which is score consistent with $\mathcal{V}$, $f(t) \geq f(R_f[k])$, i.e., the score of tuple t is no worse than the score of the kth tuple in R under the query score function f. Motivated by the previously mentioned applications where we need to efficiently answer a query using merely top-k views, we consider the following problem.

Definition 15 Top-k QAV (kQAV)*: Given a set $\mathcal{V} = \{V_1, \ldots, V_p\}$ of top-k_i views, $i = 1, \ldots, p$ and a top-k query $Q = (f, k)$, find all certain answers of Q, denoted $Q(\mathcal{V})$, up to a maximum of k answers.*

Notice that we have no access to the complete ranking of tuples in the views nor access to the base views. Similar to query answering using views in a non-ranking setting [15], given only the view set $\mathcal{V}$, we need to find the certain answers. The number of certain answers may be less than, equal to, or more than k. Since $Q = (f, k)$, we restrict the output to a maximum of k certain answers, where any ties at rank k are broken arbitrarily.

As an example, consider the set of views $\mathcal{V} = \{V_1, V_2\}$ as shown in Figure 6.1 (b) and assume the base relation R is no longer available. Assume we are given the query $Q = (f, 1)$, where $f = 0.1A + 0.8B + 0.1C$. Using the techniques proposed in Section 6.3, we can determine that $\{t_5\}$ is the set of certain answers to Q. Intuitively, this is because after accessing the second tuple in V_1 and V_2, we can derive that for all unseen tuples, the maximum possible value w.r.t. Q is 0.6425 which is smaller than the current best tuple t_5 for which $f(t_5) = 0.74$. And if $Q = (f, 4)$, we can only find 3 certain answers to Q, which are t_5, t_3 and t_1. This is because after accessing all three tuples in V_1 and V_2, the maximum possible value w.r.t. Q is 0.56 for all unseen tuples, and only t_5, t_3, t_1 have projected values larger than or equal to 0.56.

6.2.1 System Overview

Motivated by the applications discussed in the introduction, we consider the following top-k query evaluation framework as illustrated in Figure 6.2. For a top-k query $Q = (f, k)$ submitted to the query processing system, the query executor will consult the cached top-k views to find the maximum set of certain answers to Q. In this work, we will focus on the kQAV problem where the goal is to efficiently find certain answers using only the given top-k views. In the following sections, we will first adapt and improve the LPTA algorithm as originally proposed in [45] for addressing the kQAV problem as

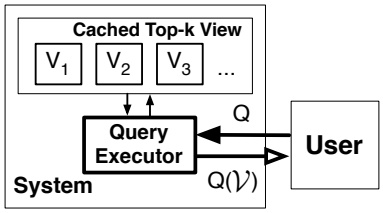


Figure 6.2: System overview.

defined above, where neither the complete ranking of tuples nor the base views are available. We will then discuss how a standard space partition-based index can be used to further optimize the performance of the algorithm.

6.3 LPTA-based kQAV Processing

In this section, we first discuss LPTA, the state-of-the-art algorithm proposed in [45] for answering a top-k query using a set of views.[21] We shall see in Section 6.3.1 that LPTA has several limitations. We first review LPTA and discuss how it can be adapted to produce certain answers when cached views are not complete rankings of tuples and no base views are available. In Section 6.3.2, we propose a new algorithm LPTA$^+$ which overcomes the limitations of LPTA.

6.3.1 Algorithm LPTA

In [45], Das et al. first studied the problem of answering a top-k query using multiple views. Similar to the TA algorithm [53], the authors of [45] assume that the underlying database can be randomly accessed to retrieve tuple attribute information using tuple ids, and that each view stores a list of tuple ids along with the scores. They focus on the scenario where either each view is a complete ranking of all tuples in R, or the base views, which are complete rankings of all tuples in R according to the values of each attribute, are available. Thus a top-k query can always be answered exactly and completely. We next briefly review the LPTA algorithm presented in [45].

Consider the score function f of each query/view also as a vector $\vec{f}$ from the origin O, representing the direction of increasing value. Given the assumption on the score function, the vector defined by any possible

[21] Recall that they assume that views are complete rankings of tuples or that base views are available.

score function considered will reside in the first quadrant. For now, we will assume that for every cached view $V_i = (f_i, k_i)$, it is the case that $k_i = |R|$ and the tuples in V_i are sorted based on f_i, or their projected values on $\vec{f_i}$. In Figure 6.3 (a), we show an example of the relation R from Example 6.1 when projected on the first two dimensions A and B. Given a query $Q = (f, k)$, we can rank the tuples by projecting them onto $\vec{f}$, as shown in the figure.

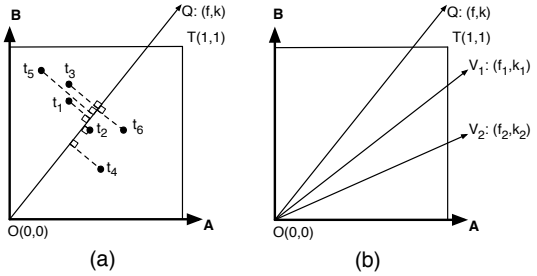


Figure 6.3: Example of LPTA.

Recall that we have a set $\mathcal{V}$ of p views, and assume that a set $\mathcal{U} \subseteq \mathcal{V}$ of r views has been selected in order to answer the query (we discuss the view selection problem below). In order to answer a top-k query $Q = (f, k)$, the LPTA algorithm accesses tuples sequentially from the r views. For each tuple t accessed, the algorithm performs a random access to the database in order to retrieve the attribute value information of t. The current candidate top-k results can be easily maintained from the accessed tuples. However, it is more challenging to find the maximum value τ that can be achieved by any unseen tuple, which is critical for the stopping criterion of the LPTA algorithm. Let the last tuple accessed in each view $V_i = (f_i, k_i) \in \mathcal{U}$ be denoted by $\bar{t}_i$, $i = 1, \ldots, r$. As observed in [45], τ can be calculated by solving the following linear programming (LP) problem:

$$\begin{aligned} \max_{t} \quad & \tau = f(t) \\ \text{subject to:} \quad & f_i(t) \leq f_i(\bar{t}_i), i = 1, \ldots, r \\ & 0 \leq t[i] \leq 1, i = 1, \ldots, m \end{aligned} \tag{6.1}$$

The "LP solver" is clearly more complex and time consuming than other components in the LPTA algorithm, so instead of invoking this solver every time a new tuple is accessed from a view V_i, LPTA accesses tuples from the r views in a lock-step fashion, i.e., the LP solver will be called once for every r tuples accessed.

The pseudocode for LPTA is given in Algorithm 18. We initialize a priority queue X based on the score function f of Q (line 1) and the threshold value τ (line 2). The algorithm then iteratively accesses tuples from the r views in a lock-step fashion (lines 4–5). For each set of r tuples accessed, the algorithm finds the value of τ by solving the LP problem (Formula 6.1) (line 8). If the kth tuple $X[k]$ in the priority queue has value no less than τ, the algorithm can stop.

Algorithm 18: LPTA($\mathcal{U} = \{V_1, \ldots, V_r\}$, $Q = (f, k)$)

```
1 X ← an empty priority queue;
2 τ ← ∞;
3 repeat
4 |   {t̄_1, ..., t̄_r} ← getNextTuple(U);
5 |   retrieveTupleInfo({t̄_1, ..., t̄_r});
6 |   X.insert({t̄_1, ..., t̄_r});
7 |   X.keepTop(k);
8 |   Find τ by solving the LP problem in Formula (6.1);
9 until noNewTuple(U) or (|X| = k and f(X[k]) ≥ τ);
```

As we will demonstrate in Section 6.3.2, while the cost of iteratively calling the LP solver is reasonable when the dimensionality for the given input relation R is low, the cost increases significantly as the dimensionality grows. *We will discuss in Section 6.3.2 how this increased cost can be avoided by leveraging innate characteristics of the kQAV problem.*

Another problem that remains to be addressed in using LPTA is how to choose the r views from a potentially large pool of cached views, so that query processing cost can be minimized. As shown in [45], we need no more than m views for processing a query on an m-dimensional relation R (so $r \leq m$), and this view selection process is critical for the performance of the LPTA algorithm. In [45], the authors first observe that for the 2-dimensional case, we can prune views by considering the angle between view score function vectors and the query score function vector. Given a query score function f and two view score functions f_1, f_2, if $\vec{f_1}$ and $\vec{f_2}$ are to the same side of $\vec{f}$, then we only need to select the view which has the smaller angle to $\vec{f}$ for answering the query, while the other view can be pruned. For example, consider the two cached views V_1 and V_2 along with query Q in Figure 6.3 (b). Because $\vec{f_1}$ has the smaller angle to $\vec{f}$, Q can be answered using V_1, while V_2 can be pruned.

However, this pruning technique may not be very useful for high dimen-

sional scenarios. As has been shown in recent work [109], the pruning of views in the general case may involve solving an LP problem whose number of constraints is proportional to the total number of views. This is clearly not practical when the number of views is large, but this is precisely the situation that arises when we want to answer a query using previously cached results. Thus, in [45], the authors adopt a greedy strategy for selecting views.

The view selection algorithm ViewSelect in [45] can be described as follows. Let $\mathcal{U}$ be the current set of views selected. ViewSelect will select the next view to be added to $\mathcal{U}$ by using function EstimateCost to simulate the actual top-k query Q on the *histograms* [65] of the views in $\mathcal{U}$ and those of the remaining views. If there is no view which can improve the cost of the current set of views, the algorithm stops and returns the current set of views selected.

Since each call of the EstimateCost sub-procedure again involves solving LP problems against the histograms of the corresponding cached views, the computational cost for view selection turns out to be very high. In Section 6.3.2 we will first improve LPTA by removing many of the calls to the LP solver. Then in Section 6.3.3, we will show how we could use an LPTA-based algorithm for handling the general kQAV problem with top-k views.

6.3.2 Algorithm LPTA$^+$

The original LPTA algorithm relies heavily on repeatedly invoking the LP solver for both view selection and query processing, since the number of times the LP solver will be invoked is proportional to the number of calls to the LPTA algorithm (on both views and histograms) multiplied by the number of tuples accessed from the views/histograms. This is especially problematic when the dimensionality is high, since the cost of LP solver increases significantly as dimensionality grows.

To test this intuition, we conducted a preliminary experiment to measure the relative contribution of the LP solver and other operations to the overall cost. For a randomly generated dataset, where each attribute value of a tuple is chosen randomly from a uniform distribution, Figure 6.4 shows how query processing cost increases as dimensionality increases. The results were obtained by selecting from a pool of 100 randomly generated views, and by averaging the time of processing 100 randomly generated top-10 queries, with all views cached in memory. As can be seen from the figure, the processing cost of the LP solver dominates the cost of other operations

in the LPTA algorithm. As the dimensionality grows, the cost of the LP solver increases quickly while the cost of other operations remains essentially constant.

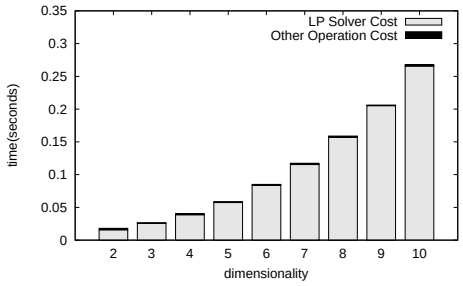


Figure 6.4: Query processing cost of LPTA as the dimensionality increases.

An important question is whether all these invocations of the LP solver are actually necessary. We will soon see that, by taking advantage of the fact that the views are cached in memory and so can be accessed sequentially with very small overhead, it will be sufficient to solve the LP problem just a few times for most executions of the LPTA algorithm.

To see this, we need to first to discuss how an LP solver works. We assume in this chapter that the LP solver is based on the SIMPLEX algorithm [43], which is the most widely used LP algorithm. The general SIMPLEX algorithm usually works in two phases. The goal of the first phase is to find one *feasible solution*[22] to the original problem, while the goal of the second phase is to find the optimal solution to the original problem. Because the formulation of our problem as represented by Formula (6.1) is in *standard maximization form* [43] (i.e., there are no constraints of the form $w_1t[1] + \cdots + w_mt[m] \geq \theta$ except the non-negative variable constraints), the first phase of finding a feasible solution is essentially trivial. Thus we need to concentrate on the second phase of the SIMPLEX algorithm.

We call each non-zero variable in a feasible solution a *basic solution variable* or BSV. In order to obtain the optimal solution in the second phase, we use the *pivoting* technique, which essentially replaces one BSV by a variable which is not currently a BSV, in the hope that the target value τ can be increased.

Now recall from the LPTA algorithm in Section 6.3.1 that for every r tuples read, we need to solve a new LP problem. An interesting characteristic of this process is that, for every LP problem formulated, the only change is

[22] A feasible solution to an LP problem is a solution which satisfies all the constraints.

in the *Right Hand Side* (RHS) of Formula 6.1, specifically $f_i(\overline{t_i})$; other parts of the constraints remain the same.

This characteristic motivates us to consider the following improvement to the LPTA algorithm. As before, we start by solving the LP problem once for the first set of r tuples accessed, deriving the BSVs for the optimal solution in the process. Then, when new tuples are accessed, we can reuse the previously derived BSVs, and check whether they lead to the optimal solution. If they do, then we have obtained the optimal solution for the new LP problem without exploring different possible BSVs using pivoting, which can be very costly [43]. The check above can be done more efficiently than pivoting. We note that this technique is different from previous work on *Incremental Linear Programming* [20], where the focus is on the more general problem of adding/removing/updating constraints.

The intuition behind the above optimization can also be illustrated using geometric properties. Consider the 2-dimensional example in Figure 6.5. Let t_1 and t_2 be the last two tuples accessed from V_1 and V_2 respectively. The optimal solution for the LP problem in Formula (6.1) can be obtained at vertex c of the convex polytope $Oacb$ in Figure 6.5 (a). Since the values of c on dimensions A and B are both positive, we know that A and B are the BSVs of the optimal solution. After we have accessed two new tuples t_3, t_4 from V_1, V_2, we need to shift the two edges ac and bc of the convex polytope down and left to $a'c'$ and $b'c'$, as shown in Figure 6.5 (b). Given the fact that the score functions of the cached views are all monotone, it is very likely that, for the new convex polytope $Oa'c'b'$, the optimal solution will be at the vertex c', which again has positive A and B values, and thus corresponds to the same BSVs. This shows that the optimal solution corresponding to the new tuples can be obtained by choosing the same set of BSVs in the LP problem, i.e., we do not need to repeat the pivoting steps to find the optimal BSVs.

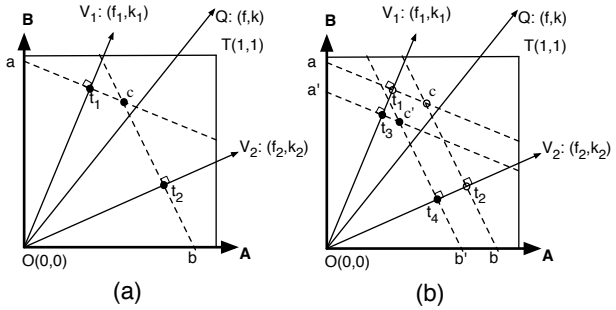


Figure 6.5: Example of LPTA$^+$.

The pseudocode of the new LPTA$^+$ algorithm is shown in Algorithm 19. Compared with LPTA, the difference lies in how the τ value is calculated (lines 8–15). For the first set of tuples accessed, we run the LP solver and derive the corresponding optimal BSVs B and τ (lines 8–9). After that, in each iteration we check whether re-pivoting is needed by using the function *isValidOptimal* to verify whether the existing BSVs lead to a new optimal solution (lines 11–12); if they do, we derive τ directly, otherwise we solve the LP problem again and derive the new B and τ. Function *isValidOptimal* basically pushes variables in B directly into the BSVs of the SIMPLEX algorithm, and checks whether it forms a valid solution considering the new RHS vector. The overhead of this operation is small and can clearly avoid many unnecessary pivoting steps in the SIMPLEX algorithm.

Algorithm 19: LPTA$^+$($\mathcal{U} = \{V_1, \ldots, V_r\}$, $Q = (f, k)$)

```
1  X ← an empty priority queue;
2  τ ← ∞, B ← nil;
3  repeat
4      {t̄_1, ..., t̄_r} ← getNextTuple(U);
5      retrieveTupleInfo({t̄_1, ..., t̄_r});
6      X.insert({t̄_1, ..., t̄_r});
7      X.keepTop(k);
8      if B is nil then
9          Compute the optimal BSVs B and τ using an LP solver;
10     else
11         derive new RHS vector b using {t̄_1, ..., t̄_r};
12         if isValidOptimal(U, B, b) then
13             derive the new τ directly;
14         else
15             Compute the optimal BSVs B and τ using an LP solver;
16 until noNewTuple(U) or (|X| = k and f(X[k]) ≥ τ);
```

Since LPTA$^+$ improves only the efficiency of calculating τ, we know that both LPTA and LPTA$^+$ will examine the same number of tuples from $\mathcal{U}$. As we will demonstrate in the experiments, the reuse of BSVs in LPTA$^+$ usually has a very small cost, and thus by avoiding many unnecessary pivoting steps, LPTA$^+$ can be much more efficient than LPTA in practice.

6.3.3 Handling the General kQAV Problem

Although LPTA$^+$ can improve the efficiency of LPTA, we still need to extend it to handle the general kQAV problem, where we have only top-k views rather than complete rankings of tuples, and no base views.

Our first observation is that, given a fixed set of views $\mathcal{U} = \{V_1, \ldots, V_r\}$, we can find all the certain tuples from $\mathcal{U}$ by using the LPTA$^+$ algorithm with the following simple modifications: (1) if the algorithm stops before all tuples in $\mathcal{U}$ are exhausted, we have already found a set of top-k certain answers for the query, since every possible unseen tuple will have a value no better than the current top-k results; (2) if we have exhausted all tuples in $\mathcal{U}$, let τ be the threshold value derived from the last tuple of each view; if we remove from the candidate top-k queue all tuples which have value smaller than τ, then the remaining tuples in the queue are guaranteed to be certain answers. Similar to the first case, the pruning of the tuples in the candidate top-k queue here is sound because τ indicates the maximum value that can be achieved by an unseen tuple, say t. Every tuple t' which is pruned has a value less than τ, so there exists a possible relation instance R which is score consistent with $\mathcal{U}$, and at the same time contains an unlimited supply of tuples that have the same attribute values as t. Thus t' cannot become a top-k result for this R since it will be dominated by t.

Now one question is whether, given a set of cached views, we can find a minimal subset of views which can give us the *maximum* set of certain answers to the query $Q = (f, k)$ (up to a total of k). Unfortunately as discussed in [45], an obvious algorithm to determine the best subset of views has a high complexity since we need to enumerate all possible combinations of r views. Instead, following the heuristics proposed in [45], we propose the modification to the LPTA/LPTA$^+$ algorithm described below. This modification guarantees that we will find the maximum set of certain answers to Q and that its complexity is linear in the number of views, but it does not guarantee that the number of views used is minimal.

Consider the second case above (we do not need any changes to the first case since that already finds a set of top-k certain answers). Instead of pruning tuples which have a value less than τ, we keep these candidate tuples and iteratively consider each of the remaining views. For each view $V' \in \mathcal{V} - \mathcal{U}$, we investigate all tuples in V' one by one, replacing existing candidate tuples with them whenever they have higher value with respect to Q; meanwhile, we try to refine the threshold value τ by considering the last tuple accessed in V'. During this process, if we have k candidate answers which have value larger than or equal to τ, we know we have found the top-k

certain answers; otherwise, if all views have been exhausted, we can get the maximum set of certain answers by pruning from the candidate queue those which have value less than τ.

It is straightforward to see that the above heuristic, when used in conjunction with LPTA instead, gives us a procedure for finding all certain answers to Q (up to a maximum of k). *Thus, LPTA can be used to find certain answers even when base views or complete tuple rankings are not available.*

6.4 IV-Index based Top-k QAV

Though the LPTA$^+$ algorithm proposed in Section 6.3 improves the efficiency of the original LPTA algorithm by avoiding unnecessary pivoting operations, the algorithm still needs to invoke the LP solver multiple times, during both view selection and query processing. When the underlying relation has high dimensionality, the cost of LP solver calls can be considerable. This motivates the quest for an even more efficient algorithm for finding the certain answers.

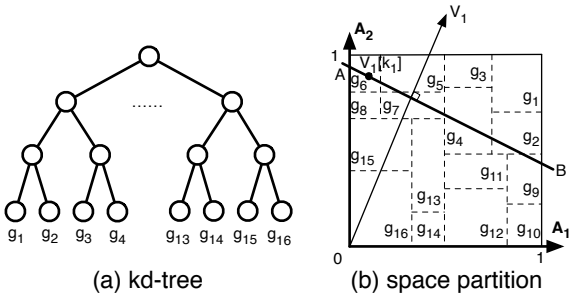
To this end, we propose a simple index structure, called the *Inverted View Index* (IV-Index). Using this index greatly reduces the number of invocations of the LP solver, allowing all certain answers in $Q(\mathcal{V})$ to be returned quickly.

6.4.1 Inverted View Index

Given the set $\mathcal{V}$ of cached views, we first collect all tuples in these views into an *Inverted View Index* (IV-Index) $\mathcal{I} = (\mathcal{T}, \mathcal{H}_V, \mathcal{H}_t)$. The components of the index are as follows: $\mathcal{H}_t$ is a lookup table which returns the attribute value information for a tuple given its id; $\mathcal{H}_V$ is a lookup table which returns the definition of a view, and $\mathcal{T}$ is a high-dimensional data structure. In this work, we utilize a kd-tree as the underlying high-dimensional data structure as it has been shown to have the most balanced performance compared with other high-dimensional indexing structures [24]. However, we note that the techniques we propose can be easily adapted to utilize *quad-trees* or other indexing structures.

Each node g in the kd-tree $\mathcal{T}$ represents an m-dimensional region, with the root node g_{root} of $\mathcal{T}$ representing the entire region from $(0, \ldots, 0)$ to $(1, \ldots, 1)$. The kd-tree is built as follows. Starting from the root node, we recursively partition the region associated with the current node g into two parts based on a selected dimension and a splitting hyperplane. These two

sub-regions are represented by two nodes which will become the children of g in $\mathcal{T}$. Once this recursive process has completed, the disjoint regions represented by the leaf nodes of $\mathcal{T}$ form a partitioning of the whole m-dimensional space. An example of a kd-tree along with the partitioning is shown in Figure 6.6.



(a) kd-tree (b) space partition

Figure 6.6: Example of (a) a kd-tree, and (b) the corresponding partition of 2-dimensional space.

For a node g, without ambiguity, we also use g to denote the region associated with the node. To facilitate query processing, we associate each leaf node g of $\mathcal{T}$ with a set T_g of tuple ids (tids), corresponding to tuples in the cached views that belong to g. Given a node (region) g, let the value range of g on each of the m dimensions be $[g_l^1, g_u^1]$, ..., $[g_l^m, g_u^m]$, and let $t_g^{\vdash} = (g_l^1, \ldots, g_l^m)$ and $t_g^{\dashv} = (g_u^1, \ldots, g_u^m)$. Then for any monotone function f, it is clear that the maximum (minimum) value that can be achieved by any tuple in g is $f(t_g^{\dashv})$, (resp., $f(t_g^{\vdash})$).

Since the set of top-k views cached in the memory may not cover the complete set of tuples in the database, it is clear that we may only have "partial" knowledge about regions associated with some leaf nodes in $\mathcal{T}$. Let R be *any* relation that is score consistent with $\mathcal{V}$. Given a region g, let R_g denote the set of tuples in R whose values fall inside g. Then we say that a region g is *complete*, or $\kappa(g) = true$, if $T_g = R_g$ for every score-consistent relation R; otherwise we say that g is *partial*, or $\kappa(g) = false$. This is a semantic property and it is expensive to check it directly. A sufficient condition for checking the completeness of a region g is given in the following lemma.

Lemma 11 *A region g is complete if there exists a top-k cached view $V_i = (f_i, k_i)$ in $\mathcal{V}$ for which $f_i(V_i[k_i]) < f_i(t_g^{\vdash})$.*

Proof *If $f_i(V_i[k_i]) < f_i(t_g^{\vdash})$, then clearly for any score-consistent relation*

R, $\forall t \in R_g$, $f_i(V_i[k_i]) < f_i(t)$. So according to the definition of top-k cached view, all tuples in R_g must belong to V; hence $R_g = T_g$.

A 2-dimensional example of Lemma 11 is shown in Figure 6.6 (b). This example shows a vector $\vec{f_1}$ corresponding to a view $V_1 = (f_1, k_1)$ along with the k_1'th tuple $V_1[k_1]$ from the view. If we draw a line AB through $V_1[k_1]$ which is perpendicular to $\vec{f_1}$, we can observe that $t^{\vdash}_{g_1}$ and $t^{\vdash}_{g_3}$ are above AB; thus $f_1(V_1[k_1]) < f_1(t^{\vdash}_{g_1})$, $f_1(V_1[k_1]) < f_1(t^{\vdash}_{g_3})$, and g_1, g_3 are complete. On the other hand, $f_1(V_1[k_1]) > f_1(t^{\vdash}_{g_2})$, so if the only top-$k$ cached view we have is V_1, we are not able to determine whether g_2 is complete or not. This is because we do not have enough information about the part of g_2 which is below AB. If R contains no tuple which falls inside this region, g_2 is complete; however, if R does contain tuples which fall inside this region, g_2 is partial.

We note that it is not possible to derive a necessary and sufficient condition for checking the completeness of a region given only the top-k cached views. This is because we will have to consult the original database R to check whether the regions which cannot be decided using Lemma 11, e.g., g_2 in above example, are complete or not. Obviously this process can be expensive, and more importantly, it is against the purpose of our kQAV framework which is to answer queries using only top-k cached views. So we will simply label regions whose completeness cannot be decided by Lemma 11 as partial. Alternative weaker sufficient conditions for completeness checking are left as future work.

Consider a partial leaf node g in $\mathcal{T}$ for a top-k cached view $V_1 = (f_1, k_1)$. If $f_1(t^{\vdash}_g) \leq f_1(V_1[k_1]) \leq f_1(t^{\dashv}_g)$ (i.e., the hyperplane which crosses $V_1[k_1]$ and is perpendicular to $\vec{f_1}$ intersects with g), we will store a pair $p = (V_1, V_1[k_1].id)$ in a *cross view* set P_g associated with g. In p, the first entry is a pointer to the definition of V_1, while the second entry is the tuple id of $V_1[k_1]$. If no such views exist, i.e., the view is complete, $P_g = \emptyset$. Consider the example of Figure 6.6 (b), and suppose that V_1 is the only top-k cached view. Then $(V_1, V_1[k].id)$ is in P_{g_2} as well as in P_{g_4},P_{g_5},P_{g_6},P_{g_7} and P_{g_9}.

6.4.2 IV-Search Algorithm

Given an IV-Index $\mathcal{I}$, a top-k query $Q = (f, k)$ can be answered by traversing the corresponding kd-tree of $\mathcal{I}$ using a strategy such as *best-first search* [108].

The pseudocode of our first algorithm, called IVS-Eager, is given in Algorithm 20. The algorithm traverses the kd-tree $\mathcal{T}$ by visiting first those nodes which have larger maximum value with respect to Q (lines 3–17), as

indicated by $f(t_g^{\dashv})$, since these nodes may have good potential to contain tuples which have high value with respect to Q. If the current node g is a leaf node, then we extract all tuples within g and check whether they can become new candidate top-k results (lines 9–11). In addition, if a leaf node g is partial, we need to collect information from P_g, which defines the region of the unseen tuples which cannot be covered by the top-k cached views, and solve a linear programming problem to find the maximum value that can be achieved by any unseen tuples in g (lines 12–13). Finally, if the current node g has its maximum value $f(t_g^{\dashv})$ less than or equal to $f(X_r[k])$, which is the value of the kth candidate tuple in X_r, the algorithm can stop, since according to the best-first search strategy, any unseen nodes cannot contain a tuple which is better than $X_r[k]$ (line 14).

Algorithm 20: IVS-Eager($\mathcal{I}$=($\mathcal{T}, \mathcal{H}_V, \mathcal{H}_t$),$Q = (f, k)$)

```
1  X_n ← an empty priority queue for kd-tree nodes;
2  X_r ← an empty priority queue for candidate results;
3  X_n.enqueue(g_root, f(t⊣_{g_root}));
4  τ ← ∞;
5  while ¬X_n.isEmpty() do
6      g ← X_n.dequeue();
7      τ ← min(τ, f(t⊣_g));
8      if isLeaf(g) then
9          foreach t ∈ g do
10             X_r.enqueue(t, f(t));
11         X_r.keepTop(k);
12         if ¬κ(g) then
13             τ ← min(τ, LPSolve(P_g, Q));
14         if |X_r| = k ∧ f(X_r[k]) ≥ f(t⊣_g) then break ;
15     else
16         foreach g_c ∈ children(g) do
17             X_n.enqueue(g_c, f(t⊣_{g_c}));
18 return {t | t ∈ X_r ∧ f(t) ≥ τ};
```

The correctness of IVS-Eager follows from the best-first search strategy, since every unseen tuple will have value smaller than $X_r[k]$ with respect to Q. In addition, the updating of the threshold value τ ensures that every tuple returned is a certain answer.

One inefficiency in IVS-Eager is that, for every partial leaf node encountered, we need to invoke an LP solver to update the threshold value τ. This can be expensive for the following two reasons: first, as shown in the example of Figure 6.6 (b), each top-k cached view might be stored in the cross view set of many nodes, so there might be duplicated computation if we solve the LP problem for every node individually; second, when the dimensionality is high, the number of such partial nodes will be large. In the following, we propose another algorithm, called IVS-Lazy, which needs to solve only one (potentially larger) LP problem.

Algorithm 21 lists the pseudocode of IVS-Lazy. The difference with IVS-Eager is that whenever a partial leaf node g is encountered in IVS-Lazy, we store the cross view set of g in a cache C_n (line 13) rather than immediately solve the LP problem and update the threshold τ as is done in IVS-Eager. After we have exhausted all nodes in the kd-tree, we collect all the view information in C_n and solve a single LP problem (lines 18–19).

Further Optimization via View Pruning

As can be observed from Algorithms IVS-Eager and IVS-Lazy, a critical operation in both algorithms is to collect constraints from the cross view set(s), and solve the LP problem given the query and constraints. Since the complexity of an LP problem may increase considerably with respect to the number of constraints, pruning constraints which are not useful can be very important for the overall query performance.

Let $Q = (f, k)$ be the query to be processed, and assume that we have accessed more than k tuples, i.e., $|X_r| \geq k$, using each of the two IV-Index based search algorithms. Now consider the point at which we solve the LP problem, i.e., line 13 in IVS-Eager and line 19 in IVS-Lazy. Let $t_{min} = X_r[k]$ be the current kth tuple in X_r, and let $V = (f', k')$ be a view from the corresponding cross view set. According to the definition, for any tuple $t \notin V$, we have $f'(t) \leq f'(V[k'])$, so the maximum value that can be achieved by any such tuple can be calculated using the following LP problem:

$$\begin{aligned} \max_{t} \quad & \phi = f(t) \\ \text{subject to:} \quad & f'(t) \leq f'(V[k']) \\ & 0 \leq t[i] \leq 1, i = 1, \ldots, m \end{aligned} \tag{6.2}$$

Let $f(t) = w_1 t[1] + \cdots + w_m t[m]$, and $f'(t) = w'_1 t[1] + \cdots + w'_m t[m]$. A careful inspection of the above LP formulation will reveal that it is exactly the *Fractional Knapsack Problem* (or *Continuous Knapsack Problem*) [71].

Algorithm 21: IVS-Lazy($\mathcal{I} = (\mathcal{T}, \mathcal{H}_V, \mathcal{H}_t)$, $Q = (f, k)$)

```
1  X_n ← an empty priority queue for kd-tree nodes;
2  X_r ← an empty priority queue for candidate results;
3  C_n ← an empty cache for partial leaf nodes;
4  X_n.enqueue(g_root, f(t^⊣_{g_root}));
5  τ ← ∞;
6  while ¬X_n.isEmpty() do
7      g ← X_n.dequeue();
8      τ ← min(τ, f(t^⊣_g));
9      if isLeaf(g) then
10         foreach t ∈ g do
11             X_r.enqueue(t, f(t));
12         X_r.keepTop(k);
13         if ¬κ(g) then C_n.add(P_g) ;
14         if |X_r| = k ∧ f(X_r[k]) ≥ f(t^⊣_g) then break ;
15     else
16         foreach g_c ∈ children(g) do
17             X_n.enqueue(g_c, f(t^⊣_{g_c}));
18 P ← consolidateCrossViewSets(C_n);
19 τ ← min(τ, LPSolve(P, Q));
20 return {t | t ∈ X_r ∧ f(t) ≥ τ};
```

In this problem, we are given a set of items $o_1, \ldots, o_m$, where each item o_i, $1 \leq i \leq m$, has weight w'_i and value w_i, and we are asked to pack them into a knapsack with maximum weight $f'(V[k'])$ such that the total value is maximized, while allowing fractions of an item to be put into the knapsack.

It is well known that a greedy algorithm which accesses items ordered by *utility* (value divided by weight) finds the optimal solution for the fractional knapsack problem, in linear time. Thus we utilize the following Algorithm FKP to find the maximum value ϕ which can be achieved by an unseen tuple with respect to a view V and a query Q. If this value ϕ is less than $f(t_{min})$ (the value of the current kth tuple in X_r), we can safely prune V from consideration in both IVS-Eager and IVS-Lazy when checking cross view sets.

Algorithm 22: FKP($Q = (f, k)$, $V = (f', k')$)

```
1 l ← {(i, u_i ← w_i / w'_i) | 1 ≤ i ≤ m};
2 Sort tuples in l based on utility;
3 φ ← 0, B ← f'(V[k']);
4 for (i, u_i) ∈ l do
5     if w'_i ≥ B then
6         φ ← φ + u_i B;
7         break;
8     else φ ← φ + w_i, B ← B − w'_i ;
9 return φ;
```

6.4.3 Discussion

Since we usually prefer the cached views to reflect the most recent and popular queries, and the memory consumption of the index structure needs to be bounded, a mechanism for cache replacement is necessary. There is much previous work on good strategies for cache/buffer replacement [42, 68], so in this work we will assume that a cache replacement strategy has been specified. Instead, we will only discuss how the basic operations of inserting and deleting a view might be implemented using the IV-Index.

To handle view insertion and deletion, we could associate with each *tuple* t cached in the memory a count $c(t)$, indicating how many views contain t. In addition, we could associate with each *node* g a count $c(g)$, specifying how many views cover g, or make g a complete node, according to Lemma 11.

First consider inserting a new top-k cached view $V = (f, k)$. For each tuple $t \in V$, we set $c(t) = c(t) + 1$ and insert t into the kd-tree if necessary. To change the completeness status of nodes affected by V in the kd-tree, we could use a best-first strategy to find each node g for which $f(t_g^{\dashv}) > f(V[k])$, and set $c(g) = c(g) + 1$. Similarly, when deleting a top-k cached view $V = (f, k)$, we could use a best-first strategy to find each node g for which $f(t_g^{\dashv}) > f(V[k])$, and set $c(g) = c(g) - 1$. In addition, we find each cached tuple $t \in V$ for which $f(t) > f(V[k])$, set $c(t) = c(t) - 1$ and remove it from cache when $c(t) = 0$.

6.5 Empirical Results

In this section, we study the performance of various algorithms for the kQAV problem based on one real dataset of NBA statistics and four synthetic datasets. The goals of our experiments are to study: (i) the performance of the LPTA-based algorithms, and by how much LPTA$^+$ improves the state-of-the-art LPTA algorithm; (ii) the relative performance of the lazy and eager versions of the IV-Index-based algorithm, and to what extent they outperform LPTA+; (iii) the effectiveness of the pruning process proposed in Section 6.4.2. We implemented all the algorithms in Python, and the linear programming solver is based on a variation of LinPro [23], and all experiments were run on a Linux machine with a 4 Core Intel Xeon CPU, OpenSuSE 12.1, and Python 2.7.2.

The NBA dataset is collected from the Basketball Statistics website [6], which contains the career statistics information of NBA players until 2009; each attribute of the dataset corresponds to one major statistics for NBA paperly, e.g., points per game. The NBA dataset has 3705 tuples and we selected 10 attributes to be used in our experiments. The synthetic datasets are generated by adapting the benchmark generator proposed in [25]. The *uniform* (UNI) dataset and the *powerlaw* (PWR) dataset are generated by considering each attribute independently. For UNI, attribute values are sampled from a uniform distribution, and for PWR, attribute values are sampled from a power law distribution with $\alpha = 2.5$ and normalized into the range $[0, 1]$. In the *correlated* (COR) synthetic dataset, values from different attributes are correlated with each other, while in the *anti-correlated* (ANT) synthetic dataset, values from different attributes are anti-correlated with each other. Each synthetic dataset is over 10 attributes and has 100000 tuples.

[23] http://www.cdrom.com/pub/MacSciTech/programming/ (visited on 03/18/2013)

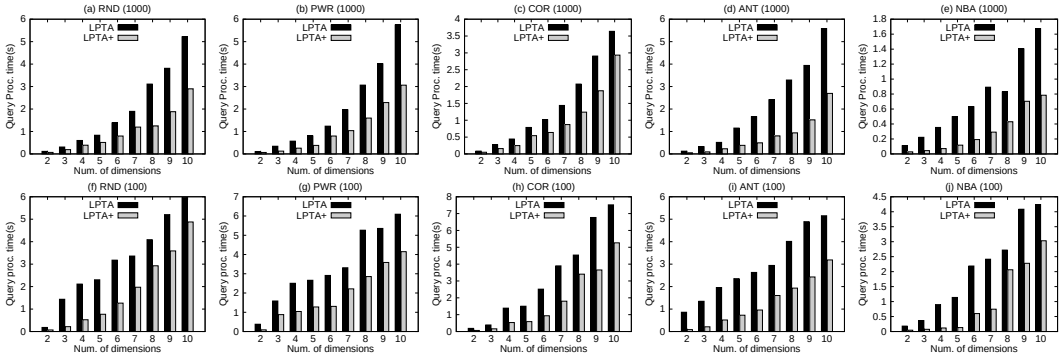


Figure 6.7: LPTA vs. LPTA$^+$: (a–e) results on 5 datasets with each view containing 1000 tuples; (f–j) results on 5 datasets with each view containing 100 tuples.

Weights for the score functions in all views are generated randomly, and all views are cached in memory. Similar to previous work on LPTA-based algorithms [45], the size of the histograms used for estimation is set to be roughly 1% of the size of the corresponding dataset. For the IV-Index-based approach, we set the number of tuples in the leaf nodes of the kd-tree to be less than or equal to 50. Alternative configurations for the kd-tree were also tested with similar results and so are omitted here for lack of space. Finally, the query score functions are also generated randomly, and all results reported here are based on an average of the results from processing 100 queries.

6.5.1 LPTA-based Algorithms

In Figure 6.7, we compare the performance of LPTA and LPTA$^+$ for queries which ask for the top-100 tuples using a set of 100 views. Figure 6.7 (a–e) considers the setting in which each view contains 1000 tuples. We can see that, for all five datasets, LPTA$^+$ is much faster than LPTA in most cases. Similar results are obtained for the setting in which each view contains 100 tuples (Figure 6.7 (f–j)). However, we note that for this setting, query processing time is longer because now the views contain fewer tuples, so we need to check more additional views in order to guarantee that we will find all the certain answers in $Q(\mathcal{V})$ w.r.t. the query Q.

In Figure 6.8 (a), we compare the performance of LPTA and LPTA$^+$ when varying the number of views in the cache pool. Here we fix the number of dimensions at 5, and consider queries where k is randomly selected from 10

to 100. As can be seen from this figure, the performance of both algorithms degenerates as the number of views increases. However, LPTA$^+$ is still twice as fast as LPTA in most settings. This result was obtained using the RND dataset. Very similar results were obtained for the other datasets and for different dimensionality settings, and are thus omitted.

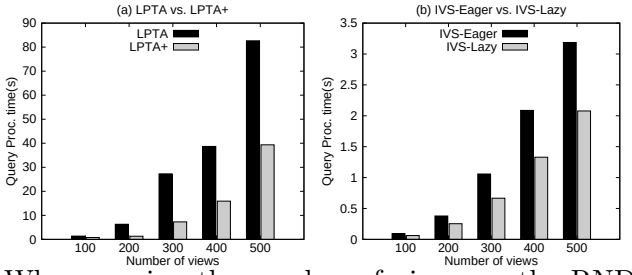


Figure 6.8: When varying the number of views on the RND dataset, the performance comparison between: (a) LPTA and LPTA$^+$; (b) IVS-Eager and IVS-Lazy.

In Figure 6.9 (a), we compare the performance of LPTA and LPTA$^+$ when varying the value k in each query from 10 to 100, given 100 views and by fixing the the dimensionality at 5. Similar to the previous results, the performance of both algorithms degenerates as k increases, but LPTA$^+$ is still faster than LPTA for all settings. The results obtained for datasets other than RND are very similar. We discuss Figures 6.8(b) and 6.9(b) below.

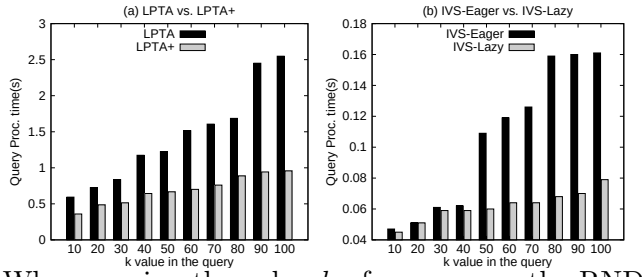


Figure 6.9: When varying the value k of a query on the RND dataset, the performance comparison between: (a) LPTA and LPTA$^+$; (b) IVS-Eager and IVS-Lazy.

Although LPTA$^+$ can greatly outperform LPTA, it can be observed that the query processing cost for LPTA$^+$ is still high.

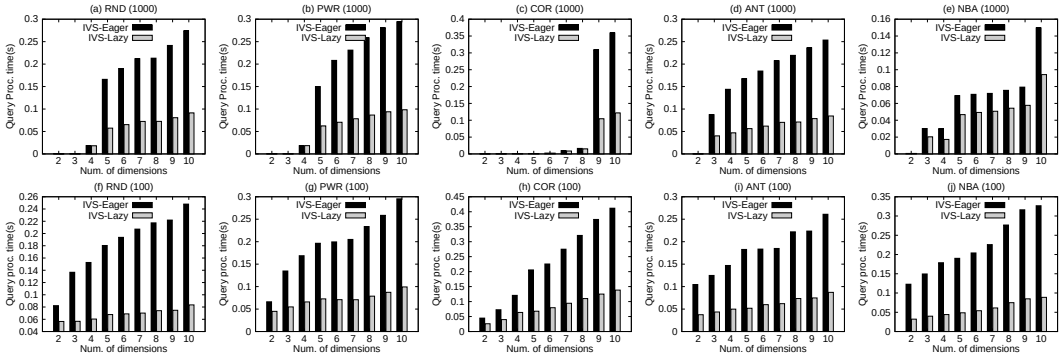


Figure 6.10: IVS-Eager vs. IVS-Lazy: (a–e) results on 5 datasets with each view containing 1000 tuples; (f–j) results on 5 datasets with each view containing 100 tuples.

6.5.2 IV-Index-based Algorithms

Figure 6.10 shows the experimental results of the IVS-Eager and IVS-Lazy algorithms under the same settings as in Figure 6.7. Compared with the results of LPTA-based algorithms in Figure 6.7, we can readily see that the IV-Index-based approaches are orders of magnitude faster than the LPTA-based approaches under all circumstances. From Figure 6.10 (a–j), we can also observe that, in most cases, IVS-Lazy is much faster than IVS-Eager, since it saves many calls to the LP solver. The only exception is for low-dimensional cases where both algorithms have a very small query processing cost. The advantage of IVS-Lazy especially applies for the high-dimensional cases where more nodes in the kd-tree are partial. Similar to the results of the LPTA-based algorithms, both IVS-Eager and IVS-Lazy are much faster on views which contain more tuples, simply because they need to check fewer partial nodes in the kd-tree.

When we vary the number of views and when we vary the number k in each query, as can be observed from Figure 6.8 (b) and Figure 6.9 (b), the performance of IV-Index-based algorithms are orders of magnitude faster than the LPTA-based algorithms. The running time of both IVS-Eager and IVS-Lazy increases as the number of views increases, or as k increases, as with all algorithms. However, IVS-Lazy has consistently better performance than IVS-Eager.

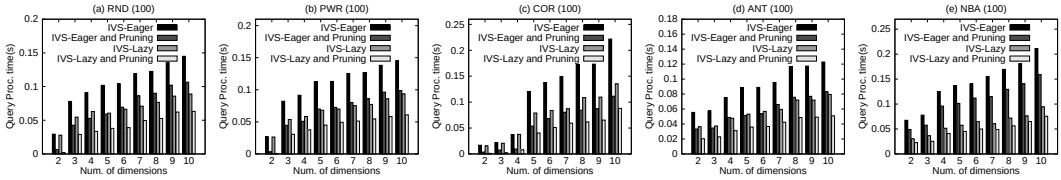


Figure 6.11: Pruning effectiveness test of IV-Search algorithms based on the five datasets.

6.5.3 Effectiveness of Pruning

Finally, in Figure 6.11, we show the effectiveness of the pruning techniques proposed in Section 6.4.2. In this experiment, we fix the number of tuples in each view to be 100, and for each query $Q = (f, k)$, k is a random number within $[10, 100]$; for other settings of these parameters, the results obtained are very similar. As can be seen from the figure, the pruning technique can improve the performance of both IV-Search algorithms. Notice that for various dimensionality settings, the overall performance of IVS-Lazy/Pruning is consistently the best on all five datasets.

6.6 Related Work

For general top-k query processing, the most popular approach is the *Threshold Algorithm* (TA) / *No Random Access Algorithm* (NRA) as proposed by Fagin et al. in [53]. While TA and NRA differ in whether random access to the database is allowed, this family of algorithms usually share a similar query processing framework which accesses tuples from the database in a certain order, while maintaining an upperbound on the maximum value that can be achieved by the tuples that have not yet been accessed. If the current top-k result has a value no less than the best value achievable by any unseen tuple, the algorithm can stop. Recently, various improvements to the original algorithms such as the *Best Position Algorithm* [18] have been proposed, while variations of top-k queries such as *Rank Join* [62] and *Continuous Top-k Queries* [128] have been studied. Finally, Li et al. study how top-k algorithms might be implemented in a relational database [86]. An excellent survey on top-k query processing can be found in [63].

Hristidis et al. [60] first considered the problem of using views to speed up top-k query processing. They focused on finding one best view which can be used for answering a query. As mentioned in [45], their setting is

quite restrictive as it cannot exploit multiple views, and it also assumes that all attributes of the underlying base table are always utilized for all top-k queries. Das et al. [45] propose a novel algorithm, called LPTA, which overcomes the limitations of [60] by utilizing multiple views for answering a top-k query.

It can be verified that the kQAV problem defined here is a generalization of the kQAV problem as considered in [45]. This is because the core techniques proposed in [45] rely on the assumption that either each top-k view $V_i = (f_i, k_i) \in \mathcal{V}$ is a complete ranking of all tuples in R, i.e., $k_i = |R|$; or the *base views*, which are complete rankings of all tuples in R according to the values of each attribute, are available. We make no such assumptions in our setting. That said, we can easily adapt our algorithms to work in settings where we do have base views available or all views are complete rankings of all tuples.

In [21], the authors consider the problem of whether a top-k query can be answered exactly using a set of top-k views, which resembles the classical query containment problem in databases [35]. However, this work does not address the general kQAV problem, i.e., return a maximum set of certain answers, in case $\mathcal{V}$ cannot answer a query Q exactly. Ryeng et al. [109] extend the techniques proposed in [45] and [21] to answer top-k queries in a distributed setting. They assume that access to the original database is available through the network interface, thus exact top-k answers can always been found by forming a "remainder" query which can be utilized to fetch tuples not available in the views. We note that the focus of our work is on efficient algorithms for finding answers to the kQAV problem, where the original database is not accessible. Should it be accessible, we can adapt the techniques proposed in [109] to find the additional answers in the case where our algorithms cannot find enough certain tuples from the cached views.

In addition to leveraging views, an alternative way of optimizing top-k query processing is through a *Layered Index* [36, 59, 81, 127]. These approaches try to organize tuples in the database into an layered index structure. We can quickly obtain the answers to a top-k query by accessing just the first few layers of the index. First, we note that our proposed IV-Index is significantly different from the layered index, since it is based on a standard space partitioning index such as kd-tree. Furthermore, these layered indexes all assume access to the original database is available, so are difficult to adapt to scenarios where we have no access to the database.

6.7 Application in Composite Recommendation

For composite recommendation with soft constraint as discussed in Chapter 5, we query top-k packages using algorithm TopPkg (Algorithm 15) for every weight sample w. Since there is no hard constraint, TopPkg only considers the value of every package w.r.t. w using its aggregate feature values. Thus given a static item dataset, we could treat each cached package as an atomic item, and directly apply the kQAV algorithms developed in this chapter to leverage cached top package results. Considering the fact that in the preference elicitation framework, we need to iteratively sample new weight vectors in order to accommodate new implicit feedbacks, the kQAV-based optimization can be extremely useful.

For composite recommendation with hard constraint as discussed in Chapter 3 and Chapter 4, because of the additional constraint, even though some previously cached top-k package results of query Q are still the top-k packages under the utility function of a new query Q', these top-k packages may not satisfy the constraint associated with Q'. In this section, we propose extensions to our proposed kQAV algorithms which can handle associated constraints with each query.

Consider composite recommendation with hard constraint and flexible schema discussed in Chapter 4, where we focus on the budget constraint (time budget and monetary budget). For IVS-based algorithms in Section 6.4, let every item here be a package which was cached due to a previous query, then we could easily associate each package p in the grid with its actual cost, thus when visiting p during query processing, we could simply check whether p satisfies the new query constraint. If it does, then we can continue as is with items. If it does not, as illustrated in Algorithm 23 line 13–14, we mark this current cell as partial, and add p to the set S_g which contains all packges in g which do not satisfy the constraint in Q. After all items in the current cell g have been processed, in line 17 of Algorithm 23, if the current cell is determined to be partial, we ignore those packages which do not satisfy the constraint of Q, and solve a LP problem to find an upperbound value on the possible utility value which can be achieved by any packages satisfying constraint of Q in this cell.

For composite recommendation with hard constraint and fixed schema as discussed in Chapter 3, we could create a separate IV-Index for every possible schema which will be used in the corresponding application, then since every query and package in an IV-Index has the same join condition, we could simply handle all remaining aggregation constraints in a similar way as in IVS-Eager-C.

Algorithm 23: IVS-Eager-C($\mathcal{I}$=($\mathcal{T}, \mathcal{H}_V, \mathcal{H}_t$),$Q = (f, k, B)$)

```
1  X_n ← an empty priority queue for kd-tree nodes;
2  X_r ← an empty priority queue for candidate results;
3  X_n.enqueue(g_root, f(t^⊣_{g_root}));
4  τ ← ∞;
5  while ¬X_n.isEmpty() do
6      g ← X_n.dequeue();
7      τ ← min(τ, f(t^⊣_g));
8      if isLeaf(g) then
9          S_g ← ∅;
10         foreach t ∈ g do
11             X_r.enqueue(t, f(t));
12             if t does not satisfy constraint B then
13                 κ(g) ← false;
14                 S_g.add(t);
15         X_r.keepTop(k);
16         if ¬κ(g) then
17             τ ← min(τ, LPSolve(P_g\S_g, Q));
18         if |X_r| = k ∧ f(X_r[k]) ≥ f(t^⊣_g) then break ;
19     else
20         foreach g_c ∈ children(g) do
21             X_n.enqueue(g_c, f(t^⊣_{g_c}));
22 return {t | t ∈ X_r ∧ f(t) ≥ τ};
```

Chapter 7

Summary and Future Research

7.1 Summary

Motivated by several applications such as trip planning, E-commerce, and course planning, we study the problem of composite recommendation in this thesis, in which instead of focusing on finding items which might satisfy users' needs, we consider how packages of items can be automatically recommended to the user. Based on different requirements from different applications, we explore in this thesis several possible variations of the composite recommendation problem.

In Chapter 3, we consider applications where schemas of the underlying packages are pre-defined, and formulate the composite recommendation problem as an extension to rank join with aggregation constraints. By analyzing their properties, we develop deterministic and probabilistic algorithms for their efficient processing. In addition to showing that the deterministic algorithm retains the minimum number of accessed tuples in memory at each iteration, we empirically showed both our deterministic and probabilistic algorithms significantly outperform the obvious alternative of rank join followed by post-filtering in many cases and that the probabilistic algorithm produces results of high quality while being at least twice as fast as the deterministic algorithm.

In Chapter 4, we consider applications where schemas of the underlying packages can be flexible, and focus on an important class of aggregation constraints based on budgets. We establish that the problem of finding the top package is intractable since it is a variant of the Knapsack problem, with the restriction that items need to be accessed in value-sorted order. We developed two approximation algorithms InsOpt-CR-Topk and Greedy-CR-Topk that are designed to minimize the number of items accessed. For InsOpt-CR-Topk, we show that every 2-approximation algorithm for the problem must access at least as many items as this algorithm. And for

Greedy-CR-Topk, though it is not guaranteed to be instance optimal, but is much faster. We experimentally evaluated the performance of the algorithms and showed that in terms of the quality of the top-k packages returned both algorithms are close to each other and deliver high quality packages; in terms of the number of items accessed Greedy-CR-Topk is very close to InsOpt-CR-Topk, but in terms of running time, Greedy-CR-Topk is much faster. We also showed that using histogram-based information about item costs can further reduces the number of items accessed by the algorithms and improves their running time. The proposed algorithm can be extended to handle cases where the budget may depend on the order of items being consumed by considering the underlying problem as an orienteering problem.

In Chapter 5, we study how composite recommendation is possible using soft constraints. Following [27, 34], we assume the system does not have the complete information about user's utility function. We cab leverage the existing preference elicitation frameworks for eliciting preferences from users, however, the challenge here is how we can perform the elicitation efficiently, especially considering the fact that we are reasoning about utilities of combinations of items. We propose several sampling-based methods which, given user feedback, can capture the updated knowledge of the underlying utility function. Finally, we also study various package ranking semantics for finding top-k packages, using the learned utility function.

Finally, we discuss in Chapter 6 a general optimization procedure based on cached views which can benefit various proposed composite recommendation algorithms. We show that the state-of-the-art algorithm for answering top-k queries using cached views, LPTA [45], suffers because of iterative calls to a linear programming sub-procedure. This can be especially problematic when the number of views is large or if the dimensionality of the dataset is high. By observing an interesting characteristic of the LPTA framework, we proposed LPTA$^+$ which has greatly improved efficiency over LPTA. We adapted both algorithms so they work in our kQAV setting, where views are not complete tuple rankings and base views are not available. Furthermore, we proposed an index structure, called IV-Index, which stores the contents of all cached views in a central data structure in memory, and can be leveraged to answer a new top-k query much more efficiently compared with LPTA and LPTA$^+$. Using comprehensive experiments, we showed LPTA$^+$ substantially improves the performance of LPTA while the algorithms based on IV-Index outperform both these algorithms by a significant margin.

7.2 Future Research

This dissertation has made substantial progress in the study of composite recommendation, but several challenges still remain.

First of all, the focus of this dissertation is on locating interesting packages given user specified preferences/constraints, either explicit or implicit. However, most of the proposed algorithms do not leverage existing transactions in the system which can be used to further understand users' preference over packages of items. E.g., for trip planning, a user might have already planned some trips before, thus given his/her existing trips, we might be able to infer what types of POIs he/she might be interested in. Some initial works have attempted to solve this problem through supervised learning [57, 91]. However, the proposed models were only verified on an extremely small and non-public dataset.

Second, another problem of finding top-k recommended packages is that many items found in the top-k packages might be duplicates, rendering these top-k packages being very similar to each other. Thus another desirable property of the generated top-k package set is diversity [106]. However, there is still a lack of principled ways of trading-off between diversity and quality of the packages. The preference elicitation framework studied in Chapter 5 might be a potential fit for solving this problem. For example, we could elicit user's preference on diversity by presenting both diversified packages and non-diversified packages.

Third, the usual recommendation interface is good for presenting a list of items, but given a set of k packages, there is a need for a good and intuitive interface for presenting these packages. In [106], the authors propose to consider the visualization problem of packages, however, the focus in on diversification. We note that in order to maximize the utility of the presented top-k packages, a good interface should not only present information related to these packages, but also consider highlighting properties which can be used to differentiate different packages along with explanation.

Finally, evaluation of the presented top-k packages is also extremely challenging. Existing evaluation measures such as precision and recall might not be suitable for composite recommendation, given the nature of the objects under consideration. For example, although user might find package p the most interesting whereas p' is presented as the top-1 package, there might be significant overlap between p and p'. Thus proper measures which take the nature of the composite recommendation problem into consideration should be designed and which can then be leveraged to understand users' preferences over packages of items.

Bibliography

[1] Flickr. http://www.flickr.com, visited on 03/16/2015.

[2] IMDB. http://imdb.com, visited on 03/16/2015.

[3] Memcached. http://memcached.org, visited on 03/16/2015.

[4] Metacritics. http://www.metacritic.com, visited on 03/16/2015.

[5] Metascore. http://www.metacritic.com/about-metascores, visited on 03/16/2015.

[6] Nba basketball statistics. http://www.databasebasketball.com, visited on 03/16/2015.

[7] Rottentomatoes. http://www.rottentomatoes.com, visited on 03/16/2015.

[8] Twitter. http://twitter.com, visited on 03/16/2015.

[9] U.s. news best collage rankings. http://www.usnews.com/rankings, visited on 03/16/2015.

[10] Wikipedia. http://www.wikipedia.org, visited on 03/16/2015.

[11] World university rankings. http://www.timeshighereducation.co.uk/world-university-rankings/, visited on 03/16/2015.

[12] Youtube. http://www.youtube.com, visited on 03/16/2015.

[13] Auto123 consumer car ratings. http://www.auto123.com/en/car-reviews/consumer-ratings/, visited on 03/18/2013.

[14] U.s. news best cars. http://usnews.rankingsandreviews.com/cars-trucks/rankings/cars/, visited on 03/18/2013.

[15] Serge Abiteboul and Oliver M. Duschka. Complexity of answering queries using materialized views. In *PODS*, pages 254–263, 1998.

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

[17] Alfred V. Aho et al. The transitive reduction of a directed graph. *J.Comp.*, 1(2):131–137, 1972.

[18] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Best position algorithms for top-k queries. In *VLDB*, pages 495–506, 2007.

[19] Albert Angel, Surajit Chaudhuri, Gautam Das, and Nick Koudas. Ranking objects based on relationships and fixed associations. In *EDBT*, pages 910–921, 2009.

[20] Greg J. Badros, Alan Borning, and Peter J. Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, 2001.

[21] Eftychia Baikousi and Panos Vassiliadis. View usability and safety for the answering of top-k queries via materialized views. In *DOLAP*, pages 97–104, 2009.

[22] Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

[23] G. R. Bitran and J.-C. Ferrer. On pricing and composition of bundles. *Production and Operations Management*, 16:93–108, 2007.

[24] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.

[25] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[26] N.D. Botkin and V.L. Turova-Botkina. An algorithm for finding the chebyshev center of a convex polyhedron. *Applied Mathematics and Optimization*, 29(2):211–222, 1994.

[27] Craig Boutilier. A pomdp formulation of preference elicitation problems. In *AAAI/IAAI*, pages 239–246, 2002.

[28] Craig Boutilier, Ronen I. Brafman, Carmel Domshlak, Holger H. Hoos, and David Poole. Cp-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *J. Artif. Intell. Res. (JAIR)*, 21:135–191, 2004.

[29] Sylvain Bouveret, Ulle Endriss, and Jérôme Lang. Conditional importance networks: A graphical language for representing ordinal, monotonic preferences over sets of goods. In *IJCAI*, pages 67–72, 2009.

[30] Ronen I. Brafman, Carmel Domshlak, Solomon Eyal Shimony, and Y. Silver. Preferences over sets. In *AAAI*, 2006.

[31] Gerhard Brewka, Miroslaw Truszczynski, and Stefan Woltran. Representing preferences among sets. In *AAAI*, 2010.

[32] Alexander Brodsky, Sylvia Morgan Henshaw, and Jon Whittle. CARD: a decision-guidance framework and application for recommending composite alternatives. In *RecSys*, pages 171–178, 2008.

[33] Zhe Cao et al. Learning to rank: from pairwise approach to listwise approach. In *ICML*, pages 129–136, 2007.

[34] Urszula Chajewska, Daphne Koller, and Ronald Parr. Making rational decisions using adaptive utility elicitation. In *AAAI/IAAI*, pages 363–369, 2000.

[35] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[36] Yuan-Chi Chang, Lawrence D. Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R. Smith. The onion technique: Indexing for linear optimization queries. In *SIGMOD*, pages 391–402, 2000.

[37] Yuan-Chi Chang, Chung-Sheng Li, and John R. Smith. Searching dynamically bundled goods with pairwise relations. In *ACM Conference on Electronic Commerce*, pages 135–143, 2003.

[38] Chandra Chekuri and Martin Pál. A recursive greedy algorithm for walks in directed graphs. In *FOCS*, pages 245–253, 2005.

[39] Gang Chen, Sai Wu, Jingbo Zhou, and Anthony K.H. Tung. Automatic itinerary planning for traveling services. *TKDE*, 26(3):514–527, 2014.

[40] Haiquan Chen, Wei-Shinn Ku, Min-Te Sun, and Roger Zimmermann. The multi-rule partial sequenced route query. In *GIS*, page 10, 2008.

[41] Jan Chomicki. Preference formulas in relational queries. *ACM Trans. Database Syst.*, 28(4):427–466, 2003.

[42] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *VLDB*, pages 127–141, 1985.

[43] George Dantzig. *Linear Programming and Extensions.* Princeton University, 1998.

[44] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Nikos Sarkas. Ad-hoc top-k query answering for data streams. In *VLDB*, pages 183–194, 2007.

[45] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. Answering top-k queries using views. In *VLDB*, pages 451–462, 2006.

[46] Mark de Berg, M. van Krefeld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer, 2000.

[47] Munmun De Choudhury, Moran Feldman, Sihem Amer-Yahia, Nadav Golbandi, Ronny Lempel, and Cong Yu. Automatic construction of travel itineraries using social breadcrumbs. In *ACM Hypertext*, pages 35–44, 2010.

[48] Sven de Vries and Rakesh V. Vohra. Combinatorial auctions: A survey. *INFORMS Journal on Computing*, 15(3):284–309, 2003.

[49] Rina Dechter. *Constraint Processing.* Morgan Kaufmann, 2003.

[50] Rina Dechter and Irina Rish. Mini-buckets: A general scheme for bounded inference. *J. ACM*, 50(2):107–153, 2003.

[51] Marie desJardins and Kiri Wagstaff. Dd-pref: A language for expressing preferences over sets. In *AAAI*, pages 620–626, 2005.

[52] Ronald Fagin et al. Comparing top k lists. In *SODA*, pages 28–36, 2003.

[53] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[54] Jonathan Finger and Neoklis Polyzotis. Robust and efficient algorithms for rank join evaluation. In *SIGMOD*, pages 415–428, 2009.

[55] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[56] Robert S. Garfinkel, Ram D. Gopal, Arvind K. Tripathi, and Fang Yin. Design of a shopbot and recommender system for bundle purchases. *Decision Support Systems*, 42(3):1974–1986, 2006.

[57] Yong Ge, Qi Liu, Hui Xiong, Alexander Tuzhilin, and Jian Chen. Cost-aware travel tour recommendation. In *KDD*, pages 983–991, 2011.

[58] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[59] Jun-Seok Heo, Junghoo Cho, and Kyu-Young Whang. The hybrid-layer index: A synergic approach to answering top-k queries in arbitrary subspaces. In *ICDE*, pages 445–448, 2010.

[60] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD*, pages 259–270, 2001.

[61] Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, 1975.

[62] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765, 2003.

[63] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[64] Ihab F. Ilyas, Rahul Shah, Walid G. Aref, Jeffrey Scott Vitter, and Ahmed K. Elmagarmid. Rank-aware query optimization. In *SIGMOD*, pages 203–214, 2004.

[65] Yannis E. Ioannidis. The history of histograms. In *VLDB*, pages 19–30, 2003.

[66] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, 2002.

[67] Bin Jiang et al. Mining preferences from superior and inferior examples. In *KDD*, pages 390–398, 2008.

[68] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, pages 439–450, 1994.

[69] Yaron Kanza, Roy Levin, Eliyahu Safra, and Yehoshua Sagiv. An interactive approach to route search. In *GIS*, pages 408–411, 2009.

[70] Ralph L. Keeney and Howard Raiffa. *Decisions with Multiple Objectives: Decisions with Preferences and Value Tradeoffs*. Cambridge University Press, 2003.

[71] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, 2004.

[72] Werner Kießling. Foundations of preferences in database systems. In *VLDB*, pages 311–322, 2002.

[73] Benny Kimelfeld and Yehoshua Sagiv. Finding and approximating top-k answers in keyword proximity search. In *PODS*, pages 173–182, 2006.

[74] Georgia Koutrika, Benjamin Bercovitz, and Hector Garcia-Molina. FlexRecs: expressing and combining flexible recommendations. In *SIGMOD*, pages 745–758, 2009.

[75] Chung T. Kwok and Daniel S. Weld. Planning to gather information. In *AAAI/IAAI, Vol. 1*, pages 32–39, 1996.

[76] Poole David L. and Mackworth Alan K. *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, New York, NY, USA, 2010.

[77] P. Lamere and S. Green. Project aura: recommendation for the rest of us. In *Sun JavaOne Conference*, 2008.

[78] Theodoros Lappas, Kun Liu, and Evimaria Terzi. Finding a team of experts in social networks. In *KDD*, pages 467–476, 2009.

[79] Javier Larrosa and Thomas Schiex. In the quest of the best form of local consistency for weighted csp. In *IJCAI*, pages 239–244, 2003.

[80] Eugene L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Man. Sci.*, 18(7):401–405, 1972.

[81] Jongwuk Lee, Hyunsouk Cho, and Seung won Hwang. Efficient dual-resolution layer indexing for top-k queries. In *ICDE*, pages 1084–1095, 2012.

[82] Justin J. Levandoski et al. Recstore: an extensible and adaptive framework for online recommender queries inside the database engine. In *EDBT*, pages 86–96, 2012.

[83] Roy Levin, Yaron Kanza, Eliyahu Safra, and Yehoshua Sagiv. Interactive route search in the presence of order constraints. *PVLDB*, 3(1):117–128, 2010.

[84] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *PODS*, pages 95–104, 1995.

[85] Alon Y. Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. Query optimization by predicate move-around. In *VLDB*, pages 96–107, 1994.

[86] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. Ranksql: Query algebra and optimization for relational top-k queries. In *SIGMOD*, pages 131–142, 2005.

[87] Chengkai Li, Nan Zhang, Naeemul Hassan, Sundaresan Rajasekaran, and Gautam Das. On skyline groups. In *CIKM*, 2012.

[88] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. On trip planning queries in spatial databases. In *SSTD*, pages 273–290, 2005.

[89] Greg Linden et al. Interactive assessment of user preference models: The automated travel assistant. In *UM*, pages 67–78, 1997.

[90] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. Practical selectivity estimation through adaptive sampling. In *SIGMOD*, pages 1–11, 1990.

[91] Qi Liu, Yong Ge, Zhongmou Li, Enhong Chen, and Hui Xiong. Personalized travel package recommendation. In *ICDM*, 2011.

[92] Eric Hsueh-Chan Lu, Chih-Yuan Lin, and Vincent S. Tseng. Trip-mine: An efficient trip planning approach with travel time constraints. In *Mobile Data Management (1)*, pages 152–161, 2011.

[93] Alberto Marchetti-Spaccamela and Carlo Vercellis. Stochastic on-line knapsack problems. *Math. Program.*, 68:73–104, 1995.

[94] Davide Martinenghi and Marco Tagliasacchi. Proximity rank join. *PVLDB*, 3(1):352–363, 2010.

[95] Denis Mindolin and Jan Chomicki. Discovering relative importance of skyline attributes. *PVLDB*, 2(1):610–621, 2009.

[96] Danupon Nanongkai, Ashwin Lall, Atish Das Sarma, and Kazuhisa Makino. Interactive regret minimization. In *SIGMOD*, pages 109–120, 2012.

[97] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang. Exploratory mining and pruning optimizations of constrained association rules. In *SIGMOD*, pages 13–24, 1998.

[98] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[99] Aditya Parameswaran and Hector Garcia-Molina. Recommendations with prerequisites. In *ACM RecSys*, pages 353–356, 2009.

[100] Aditya Parameswaran, Petros Venetis, and Hector Garcia-Molina. Recommendation systems with complex constraints: A CourseRank perspective. Technical report, 2009.

[101] Aditya G. Parameswaran, Hector Garcia-Molina, and Jeffrey D. Ullman. Evaluating, combining and generalizing recommendations with prerequisites. In *CIKM*, pages 919–928, 2010.

[102] Aditya G. Parameswaran, Petros Venetis, and Hector Garcia-Molina. Recommendation systems with complex constraints: A course recommendation perspective. *ACM Trans. Inf. Syst.*, 29(4):20, 2011.

[103] Jian Pei and Jiawei Han. Can we push more constraints into frequent pattern mining? In *KDD*, pages 350–354, 2000.

[104] Robert Price and Paul R. Messinger. Optimal recommendation sets: Covering uncertainty over user preferences. In *AAAI*, pages 541–548, 2005.

[105] Kenneth A. Ross, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. Foundations of aggregation constraints. *Theor. Comput. Sci.*, 193(1-2):149–179, 1998.

[106] Senjuti Basu Roy, Sihem Amer-Yahia, Ashish Chawla, Gautam Das, and Cong Yu. Constructing and exploring composite items. In *SIGMOD*, pages 843–854, 2010.

[107] Senjuti Basu Roy, Gautam Das, Sihem Amer-Yahia, and Cong Yu. Interactive itinerary planning. In *ICDE*, pages 15–26, 2011.

[108] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, third edition, 2010.

[109] Norvald H. Ryeng, Akrivi Vlachou, Christos Doulkeridis, and Kjetil Nørvåg. Efficient distributed top-k query processing with caching. In *DASFAA*, pages 280–295, 2011.

[110] Karl Schnaitter and Neoklis Polyzotis. Evaluating rank joins with optimal cost. In *PODS*, pages 43–52, 2008.

[111] Mehdi Sharifzadeh, Mohammad R. Kolahdouzan, and Cyrus Shahabi. The optimal sequenced route query. *VLDB J.*, 17(4):765–787, 2008.

[112] Mohamed A. Soliman et al. Ranking with uncertain scoring functions: Semantics and sensitivity measures. In *SIGMOD*, pages 805–816, 2011.

[113] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB*, pages 648–659, 2004.

[114] Dimitri Theodoratos and Timos K. Sellis. Data warehouse configuration. In *VLDB*, pages 126–135, 1997.

[115] Olivier Toubia et al. Polyhedral methods for adaptive choice-based conjoint analysis. *Journal of Marketing Research*, 41(1):pp. 116–131, 2004.

[116] Quoc Trung Tran, Chee-Yong Chan, and Guoping Wang. Evaluation of set-based queries with aggregation constraints. In *CIKM*, pages 1495–1504, 2011.

[117] Panayiotis Tsaparas, Themistoklis Palpanas, Yannis Kotidis, Nick Koudas, and Divesh Srivastava. Ranked join indices. In *ICDE*, pages 277–288, 2003.

[118] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[119] Jianshu Weng, Ee-Peng Lim, Jing Jiang, and Qi He. TwitterRank: finding topic-sensitive influential twitterers. In *WSDM*, pages 261–270, 2010.

[120] Min Xie, Laks V. S. Lakshmanan, and Peter T. Wood. Breaking out of the box of recommendations: From items to packages. In *RecSys*, pages 151–158. ACM, 2010.

[121] Min Xie, Laks V. S. Lakshmanan, and Peter T. Wood. Comprec-trip: A composite recommendation system for travel planning. In *ICDE*, pages 1352–1355, 2011.

[122] Min Xie, Laks V. S. Lakshmanan, and Peter T. Wood. Efficient rank join with aggregation constraints. *PVLDB*, 4(11):1201–1212, 2011.

[123] Min Xie, Laks V. S. Lakshmanan, and Peter T. Wood. Efficient rank join with aggregation constraints. *PVLDB*, 4(11):1201–1212, 2011.

[124] Min Xie, Laks V. S. Lakshmanan, and Peter T. Wood. Efficient top-k query answering using cached views. In *EDBT*, pages 489–500, 2013.

[125] Min Xie, Laks V. S. Lakshmanan, and Peter T. Wood. Ips: An interactive package configuration system for trip planning. *PVLDB*, 6(12):1362–1365, 2013.

[126] Min Xie, Laks V. S. Lakshmanan, and Peter T. Wood. Generating top-k packages via preference elicitation. *PVLDB*, 7(14):1941–1952, 2014.

[127] Dong Xin, Chen Chen, and Jiawei Han. Towards robust indexing for ranked queries. In *VLDB*, pages 235–246, 2006.

[128] Albert Yu, Pankaj K. Agarwal, and Jun Yang. Processing a large number of continuous preference top-k queries. In *SIGMOD*, pages 397–408, 2012.

[129] Xi Zhang and Jan Chomicki. Preference queries over sets. In *ICDE*, pages 1019–1030, 2011.

[130] Feng Zhao, Gautam Das, Kian-Lee Tan, and Anthony K. H. Tung. Call to order: A hierarchical browsing approach to eliciting users preference. In *SIGMOD*, pages 27–38, 2010.