UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Improving User Efficiency in Structured Data Exploration

Permalink
https://escholarship.org/uc/item/01h9w1fc

Author
Kashyap, Abhijith Ramesh

Publication Date
2013

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Improving User Efficiency in Structured Data Exploration

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

Doctor of Philosophy

in

Computer Science

by

Abhijith Ramesh Kashyap

June 2013

Dissertation Committee:
Dr. Evangelos Christidis, Chairperson
Dr. Vassilis Tsotras
Dr. Eamonn Keogh
Dr. Harsha V. Madhyastha

Copyright by
Abhijith Ramesh Kashyap
2013

The Dissertation of Abhijith Ramesh Kashyap is approved:

____

____

____

____

Committee Chairperson

University of California, Riverside

Acknowledgements

This PhD would not have been possible without a lot of support and guidance. I would like to thank my adviser Dr. Vagelis Hristidis for his constant encouragement and excellent guidance throughout this PhD. He introduced me to the research ideas that eventually led to this dissertation. I would also like to thank Dr. Michalis Petropoulos who advised me during my Master's degree and encouraged me to pursue a PhD. I thank other members of my committee, Dr. Vassilis Tsotras, Dr. Eamonn Keogh and Dr. Harsha Madhayastha for their insightful comments and guidance.

Finally, I would like to thank my parents who stood by me during this long journey, and to all my friends who encouraged me to finish my dissertation and for their understanding when I ignored them for long periods of time.

ABSTRACT OF THE DISSERTATION

Improving User Efficiency in Structured Data Exploration

by

Abhijith Ramesh Kashyap

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2013
Dr. Evangelos Christidis, Chairperson

While the number, sizes and complexity of databases have increased, the query interfaces that facilitate exploration of these databases have largely remained inadequate.

This dissertation takes a principled approach to user data exploration and proposes techniques that simplify access to large and complex structured and semi-structured databases. Various factors that affect usability such as user preference, intuitiveness of interfaces, and effort expended by users are taken into account, in addition to database structure. The interplay of these factors is studied and the proposed methods effectively utilize them to deliver maximum benefit to users. The most common tasks in a data exploration scenario are query formulation, results navigation and presentation. We propose and evaluate methods to improve usability for all these tasks. The techniques proposed are often complementary to each other, and exploit domain properties of the data. The effectiveness of the approaches is demonstrated with experiments on real-life datasets and comprehensive user-studies, wherever applicable.

The first part of the dissertation presents an auto-completion style query formulation interface, which enables users to augment keyword queries by adding structured conditions. The resulting

queries are focused and more likely to return results that the user finds relevant. The next two parts of the dissertation focus on challenges in two commonly used scenarios of results navigation: Categorization and Faceted Navigation. To model and quantify the effort incurred by a user, navigation and cost models are proposed for both navigation scenarios. Techniques to estimate this effort, taking into account preferences, are proposed and algorithms developed to compute the minimal set of suggested options that, if followed by the user, minimize the expected effort required to navigate the results. The final part focuses on the results presentation. They present a method to construct result snippets, which complements existing methods that consider solely the importance of the selected attributes. This method considers the user effort required to read and comprehend the snippets.

Contents

List of Figures **ix**

List of Tables **xi**

1 Introduction 1
1.1 Background and Motivation… 1
1.2 Contributions… 6
2 Effective Query Formulation on Complex Catalogs **11**
2.1 Introduction… 11
2.2 Framework and Definitions… 15
2.3 Structured Suggestions… 18
2.3.1 Category Suggestions : HCD… 18
2.3.2 Attribute Suggestions… 20
2.4 Computing Highest Common Descendants… 21
2.4.1 Keyword Inverted List (KIL) Index… 21
2.4.2 TopkHCD: Efficient Computation of HCDs… 23
2.5 Synopsis and Query Estimation… 25
2.5.1 Background and Our Synopsis Model… 26
2.5.2 KIL on Synopsis… 28
2.5.3 HCD Computation on Synopsis… 29
2.6 Experimental Evaluation… 33
2.6.1 Experimental Setup… 33
2.6.2 Quantitative Experiments… 34
2.6.3 Qualitative Experiments… 38
2.7 Related Work… 40
2.8 Summary… 42
3 Results Navigation Using Concept Hierarchies **43**
3.1 Introduction… 43
3.2 Framework and Overview… 50
3.3 Navigation and Cost Model… 56
3.4 Estimation of Navigation Probabilities… 59
3.5 Complexity Results… 61
3.6 Algorithms… 64
3.6.1 Optimal Algorithm for Best EdgeCut… 64
3.6.2 Heuristic-ReducedOpt Algorithm… 66
3.6.3 The TopKLevelWise Method… 67
3.7 Implementation and Experimental Evaluation… 68
3.7.1 System Architecture and Implementation… 69
3.7.2 Navigation Cost Evaluation… 71
3.7.3 *Opt-EdgeCut* Comparison… 75
3.7.4 Performance Evaluation… 76
3.8 Related Work… 77
3.9 Summary… 79

4 Cost-Driven Exploration of Faceted Query Results **80**
4.1 Introduction…… 80
4.2 Framework and Definitions…… 84
4.3 Navigation and Cost Model…… 86
4.3.1 Faceted Navigation Model…… 86
4.3.2 Faceted Cost Model…… 87
4.4 Estimating Probabilities…… 90
4.5 Algorithms…… 91
4.5.1 Complexity Results…… 92
4.5.2 Cost Model Analysis…… 93
4.5.3 ApproximateSetCover Heuristic…… 95
4.5.4 UniformSuggestions Heuristic…… 97
4.6 Experimental Evaluation…… 101
4.6.1 Setup…… 101
4.6.2 Experiments with Navigation Cost…… 104
4.6.3 Execution Time Evaluation…… 106
4.7 User Evaluation…… 107
4.8 Related Work…… 110
4.9 Summary…… 111
5 Comprehension Based Result Snippets **112**
5.1 Introduction…… 112
5.2 Framework and Definitions…… 115
5.3 Comprehension Model…… 118
5.3.1 Comprehension Model and Factors…… 118
5.3.2 User Study Setup…… 121
5.3.3 User Study Results…… 123
5.4 Informativeness…… 124
5.5 Complexity Results…… 127
5.6 Snippet Construction Algorithm…… 128
5.7 Experimental Evaluation…… 134
5.7.1 Experimental Setup…… 134
5.7.2 Results…… 136
5.8 Related Work…… 139
5.9 Summary…… 140
6 Conclusions and Future Work **141**
6.1 Conclusions…… 141
6.2 Future Work…… 143

References **145**

List of Figures

1.1 Components of a Query Interface 2

2.1 Navigation states, before and after refinements.. 12

2.2 Navigation hierarchies of states (queries) in Figure 2.1.................................. 13

2.3 KWalker Query Interface.. 13

2.4 An Example Navigation Hierarchy.. 16

2.5 Data-Tree representation of a hierarchically organized dataset....................... 17

2.6 TopkHCD Algorithm.. 24

2.7 Database Synopsis of Data-Tree in Figure 2.5.. 26

2.8 S-TopkHCD Algorithm... 30

2.9 Top-k Spearman's Footrule score for Category Suggestions (Electronics Dataset)..... 34

2.10 Top-k Spearman's Footrule score for Attribute Suggestions (Electronics Dataset)...... 35

2.11 Average Relative Error for Category Suggestions (Electronics Dataset)................ 36

2.12 Average Execution Times (in seconds) for Category and Attribute Suggestions (Electronics Dataset) (Summary-1).. 36

2.13 Experiments with Navigation Cost (Electronics Dataset)................................ 37

2.14 Average number of refinements for queries (Electronics Dataset)................. 37

2.15 Experiments with Navigation Cost (Books Dataset).. 38

3.1 Static Navigation on the MeSH Concept Hierarchy.. 46

3.2 Dynamic navigation steps to reach the concept "Histones" for the query "prothymosin" 47

3.3 The BioNav Interface.. 48

3.4 (a) Navigation Tree, EdgeCut and Component Subtrees, (b) Visualization of the EdgeCut on the user interface.. 52

3.5 The Active Tree Before and After the EdgeCut in Figure 3.4........................... 53

3.6 Top-Down Navigation Model... 55

3.7 BioNav System Architecture 68

3.8 Overall Navigation Cost Comparison for Biochemistry and Medicine 72

3.9 Number of Expand Actions Comparison …… 73

3.10 Number of Concepts Revealed Comparison …… 74

3.11 Overall Navigation Cost Comparison …… 74

3.12 Heuristic-ReducedOpt EXPAND Performance …… 76

4.1 The FACeTOR Interface …… 82

4.2 Faceted Navigation Model …… 85

4.3 Result Set R_Q, All Facet Conditions $C(R_Q)$, and Three Alternative Sets of Suggested Conditions $C_S(R_Q)$…… 94

4.4 ApproximateSetCover Heuristic …… 95

4.5 UniformSuggestions Heuristic …… 99

4.6 For the UsedCars Dataset: (a) Average Navigation Cost, and (b) Average Number of REFINE and EXPAND Actions, and Average Number of Suggested Conditions per Navigation Step (numbers on top of the bars), for $B = 1$ …… 102

4.7 Average Overlap per Navigation Step for the UsedCars Dataset, for $B = 1$…… 103

4.8 Average Navigation Cost for B=5 and B=10 (UsedCars Dataset) …… 104

4.9 For the IMDB Dataset: (a) Average Navigation Cost, and (b) Average Number of REFINE and EXPAND Actions, and Average Number of Suggested Conditions per Navigation Step (numbers on top of the bars), for B=1 …… 106

4.10 Average Execution Time of UniformSuggestions Heuristic (UsedCars dataset & B=1) …… 106

4.11 Actual User Navigation Time for 8 Result Sets …… 108

4.12 Average number of Suggestions and Actions …… 108

4.13 Actual Time vs. Estimated Navigation Cost …… 108

4.14 Users Perception (Questionnaire) …… 109

5.1 A heterogeneous result-set for query ‘acer laptop 3gb’ and three snippets with different characteristics …… 114

5.2 Comprehension Cost User Study Results …… 122

5.3 Reduction of Set Cover to the Fixed Snippet Construction (FSC) Problem …… 127

5.4 Snippet Construction Algorithm …… 132

5.5 Goodness Score ($\mathcal{F}$) of Result-set Snippets …… 136

5.6 Total Informativeness of Attributes in Snippets …… 137

5.7 Total Result-set Snippet Comprehension Cost …… 137

5.8 Snippet Construction Performance …… 138

List of Tables

2.1 Dataset Characteristics………………………………………………………………… 32

2.2 Sample Query Workload…………………………………………………………….... 33

3.1 BioNav Evaluation Query Workload ……………………………………………….... 71

4.1 Symbol Reference…………………………………………………………………….. 84

4.2 FACeTOR Evaluation Query Workload…………………………………………….... 101

5.1 Combinations of $nOccur(A_i)$ and $nPos(A_i)$ Each cell is a task……………………. 121

5.2 Snippet Construction Evaluation Queries……………………………………………. 134

Chapter 1

Introduction

1.1 Background and Motivation

The number and sizes of databases published online have increased significantly over the recent years. These databases are typically structured or semi-structured, and have some form of meta-data (such attributes, concept hierarchy, annotations, relations, etc.) associated with data values, which are organized by a database schema. Examples of such databases include product catalogs (such as Amazon.com[1], eBay.com[2] etc.), local businesses (e.g. Yellow Pages[3], Yelp[4] etc.), biomedical databases (e.g. Entrez Gene[5], OMIM[6] etc.), bibliographies (e.g. DBLP[7], PubMed[8]) among many others.

Access to these databases by *everyday common (as opposed to expert) users* is facilitated by means of a query interface. These query interfaces shield users from the complexities of the underlying database schema (the structure of the data and their relationships) and the intricacies of a language (e.g. SQL, XQuery) used to query the data.

Example 1.1: As an example, consider the query interface of the popular e-commerce website Amazon.com as shown in Figure 1.1. The product catalog of Amazon.com has a complex structure where each item has a type (e.g. *Laptop*, *Desktop* etc.) and a set of type-specific attributes and associated values (e.g. *Brand: Dell*, *Memory: 3Gb* etc. for a *Laptop* item). In order to

formulate a query "*laptops with Brand = Dell*" the user must be aware of the underlying database structure (XML, Relational etc.) and schema (e.g. a universal relation or tables for each type etc.).

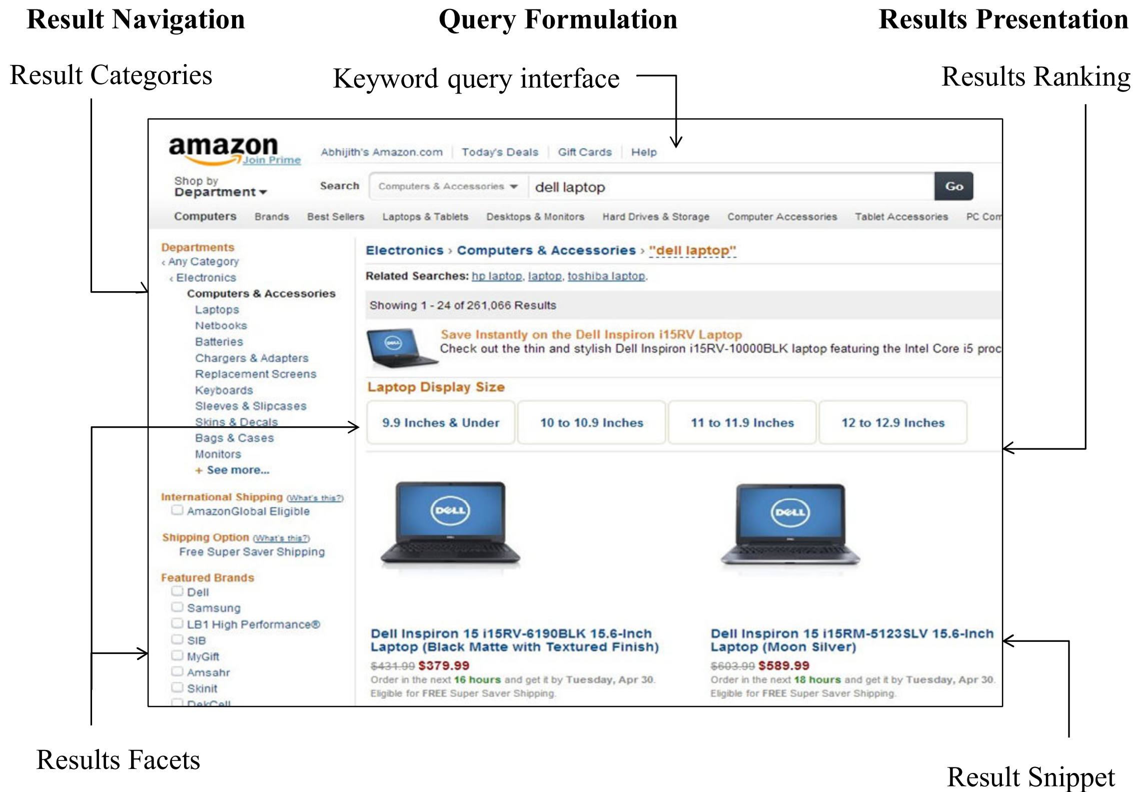


Figure 1.1. Components of a Query Interface.

Although query interfaces are widely utilized by users to query and explore databases, the *usability* of query interfaces has received relatively less attention [9] and the techniques used in the realization interface components are mostly ad-hoc in nature and often leads to additional and redundant effort on part of the users. In this dissertation, we focus on methods and techniques to reduce the effort incurred by users in data exploration tasks. We focus on components of a query interface which facilitate data-exploration on structured (or semi-structured) databases by providing users with tools to formulating queries and then navigate the query results.

In this section, we begin by introducing the components of a query interface and identify the limitations and challenges that we address in the rest of the dissertation. The underlying theme in all scenarios introduced is the effort on part of the users that can be reduced or entirely eliminated. Figure 1.1 shows an example of a typical query interface that is commonly available to users, who typically explore the dataset by issuing queries and then navigating the returned results. It shows the three main components: (a) Query Formulation, (b) Results Navigation and (c) Results Presentation.

(a) Query Formulation: The query interface in Figure 1.1 provides a simple keyword-based search to query the underlying product catalog. This simple yet intuitive query paradigm has become very popular method, over other methods such as Query Forms [10, 11], to query complex databases. These unstructured keyword queries are then processed by the underlying search engine either by utilizing index structures such as Lucene[12] or translating them into structured queries [13-16] based on the ways the query keywords are connected to each other.

Limitations: Keyword queries are typically ambiguous and typically return a large number of results from multiple interpretations of the query. The primary sources of ambiguity are ambiguity in keywords (e.g. keyword "fox" can refer to *Fox News* channel or the *Fox Software* company etc.) or due to ambiguity in inferring the relevant structural relationships in the underlying database - for example, the keyword query *"dell laptop"* shown in Figure 1.1 could refer to *Laptops* manufactured by *Dell*, or some accessories or components compatible with *Dell* laptops, among many others. The users then have to navigate the results using other components of the query interface to find the results that are relevant to her. The metadata and structure in structured databases can be exploited to build a more focused query. For example, the query "dell laptop" can be augmented with *structured conditions* such as $\{Type = Laptop,\ Brand = Dell\}$

and the resulting query would return only *Laptops* manufactured by *Dell.* However, the primary difficulty is the selection of structured conditions to add to the query, which are dependent on users query intent[17], which is difficult to determine.

(b) Results Navigation: To navigate large result-sets returned by queries, query interfaces help manage the exploration of these returned query results. Common methods of exploring a large result-set are by using *Categorization* or *Faceted Navigation.* With categorization the results are organized into a fixed navigation hierarchy or taxonomy and the user navigates the results by filtering the results by terms or concepts in the taxonomy. Analogously, faceted navigation allows the user to filter a result-set based on multiple classifications or facets, as opposed to a single classification as in the case of categorization. The user explores the results of the query iteratively by refining the results by filtering it by more specific categories (in case of a taxonomy) or by filtering it by one or more facet conditions.

Example 1.2: The results of the query "*dell laptop*" in Figure 1.1 are categorized into a pre-defined taxonomy of product types. A user who is interested solely in *Laptops* can filter the results by selecting this category. This taxonomy is also usually very large and the query interface displays only a portion of it to the user. If the user wishes to explore a different section of the taxonomy, for e.g. *Memory* (not shown in the figure) then she has to navigate the category hierarchy to reach the appropriate category. The figure also shows two (*Brands* and *Screen Size*) of the many attribute facets that can be used to refine the query.

Limitations: While Facets and Categorization provide an efficient way to navigate and find the results of interest, the existing methods follow ad-hoc methods to select and display options, such as preselecting facets to be displayed regardless of the query or the user's current context and ignoring user preferences. For example, the query interface in Figure 1.1, like most query

interfaces, expands the classification hierarchy level by level and selects a fixed set of facets to be displayed at each step. This often leads to additional effort on the part of the user who, now has to process (read and select) a multitude of options before deciding on the ones suitable to her. The set of suggestions revealed at each step do not take into account user-preferences and are not dependent on the state of user's navigation. Furthermore, many options are mutually redundant, e.g. Screen Size and RAM (not displayed in Figure 1.1) which are correlated, and displaying them together adds to the effort in perusing them but lead to the same results upon filtering by any one of these conditions.

(c) Results Presentation: A query interface provides a way to present and visualize the query results. For large result-sets, the interface typically paginates the results and presents a small subset to the user. To enable the user to quickly find the results, the interface orders or *ranks* the results in some pre-defined way so as to display more relevant results before irrelevant ones. In addition, the query interface presents only a small portion or a *snippet* of a given result to the user. This is because a result might contain a large amount of information which can easily overwhelm the user.

Example 1.3: The example in Figure 1.1 shows only the *first* 24 of the 260,000+ results of the query *"dell laptop"* ordered by *relevance* (as decided by the search engine). To access desired results which are not in the first page, the user has to either navigate additional pages or filter the query using categories or facet conditions. For each result, only a small subset of information (attribute and values) is displayed in Figure 1.1. The *Model, Make, Screen size, Color* and *Price* are displayed for the *Laptops* and a large number of attributes such as *CPU type, Memory Type* etc. are hidden.

Limitations: There has been a large body of work, including but not limited to [18-20], on ranking of results for structured data, but the important problem of presentation has been largely ignored. While snippets are an important component of a query interface, there has been very little work on their construction and presentation, and most works [21, 22] have focused on selecting the most important elements (attributes, values etc.) to be displayed in a snippet. A snippet helps the user to decide on the relevance of a result and also to compare a result with others, by comparing the attribute values in the snippet. Therefore, in addition to selecting important elements, it is also necessary to arrange and present them in a way that makes it easy for users to read and digest the information in snippets.

1.2 Contributions

In this dissertation, we propose techniques to address some of the aforementioned drawbacks in various components of the query interfaces. We take a principled approach to user data exploration and develop techniques that simplify access to large and complex databases. In addition structural-complexity, these techniques also take into account various factors that affect usability such as user preference, intuitiveness of interfaces, and effort expended by users which have been largely ignored in current works. We study the interplay of these factors and devise methods to effectively utilize them to deliver maximum benefit to users.

Modeling User Effort: The *effort* required by users in using a query interface is a subjective measure and depends on a number of factors such as the complexity of user-interface elements, complexity of data-elements and their number, among many others. For example, reading *Terms and Conditions* field requires much more effort than say, the *Price* field in the example interface of Figure 1.1. As we noted in Section 1.1, the number of options displayed to the user and their organization (sorting, visualization etc.) are factors that affect the user-effort in using these

interfaces. We focus our approaches around the user-effort and propose models to capture this effort. Our models are partly inspired by the work of Chakrabarti et. al.[23] which proposes an effort model based on aggregating the actions taken by a user in navigating results on a structured query interface.

On a typical query interface, such as the one shown in Figure 1.1, the user takes several actions to satisfy his/her information need. After executing the query, the user is presented with a (ranked) list of results and options viz. facets, categories etc. to navigate these results. The user then takes a number of actions to navigate the results to narrow down to results of interest, such as reading the results or its snippets to decide its relevance, reading the available options to decide if a particular option can be used to filter the query result. Each such action is an effort on the part of the user navigating the result set and counting all the actions taken by the user on a query interface gives an estimate of the overall effort.

In Chapter 2, we propose a query formulation interface which suggests structured conditions that can be added to a query allowing the user to quickly formulate a highly focused query. In Chapters 3 and 4, we construct *navigation models* that closely follow the actions of the user on a query interface with categorized or faceted results. In Chapter 5, we propose a model of a user *reading* a list of result snippets.

Quantifying and Estimating Effort: Based on the model of user, we construct a *cost model* that quantifies and estimates the effort incurred by users in data-exploration tasks. Using the cost model, we cast the problem of selecting options that minimize user-effort into an optimization problem thereby allowing us to use algorithmic techniques to solve them. The estimates are based on preferences for certain navigation and exploration patterns over others this allowing us to introduce user preferences into cost estimates.

Evaluation based on User Study: We evaluate our approaches with comprehensive experiments with real-life datasets based on simulations. Additionally, since most of the approaches are based on a model user behavior, it is requisite that the approaches be evaluated with real-life users. Most of the approaches we proposed are evaluated on real anonymous users (invited via Amazon Mechanical Turk [24]) and we show significant improvement in using our approaches as compared to state-of-the-art.

Next, we provide a summary of the contributions of this dissertation, which all aim at reducing the user-effort in data exploration tasks.

1. Query Formulation over Complex Catalogs

In Chapter 2, we present a novel holistic framework to formulate rich and more focused queries, starting from keywords that minimize the user time and effort. As the user is entering keywords, the system termed KWalker, almost instantly suggests the most promising hierarchy categories and attributes conditions to refine the initial keyword query. These suggestions are based on the novel principled concept of Highest Common Descendants (HCDs), which are the categories with the right amount of specificity, given the query. We propose efficient algorithms that use summarization, indexing and early termination to generate the *best* HCDs in order of milliseconds. Further, we propose techniques to select the best attribute conditions to display given the computed HCDs.

2. Navigation of Query Results Based on Concept Hierarchies

As discussed above in Section 1.1, the use of ad-hoc and unprincipled methods in navigating the results of a query using concept hierarchies leads to sub-optimal navigation in terms of user-effort. In Chapter 3, we propose techniques to decrease the user effort in navigating the

results using category hierarchies. Instead of using level-wise expansion, we propose a novel exploration model which at each expansion step, reveals a small subset of nodes from the descendants (not necessarily children) of the node being expanded. This subset of nodes is chosen such that the *estimated* effort required to reach all the query results is minimized.

We propose a model of navigation that closely mimics the actions of a user exploring a set of results using a category hierarchy, thereby allowing us to estimate the effort required to navigate the result-sets. Based on this navigation model, we propose a cost model to quantify the effort incurred by users during the navigation process. We analyze the complexity of the problem of constructing a subset of nodes that minimize the *expected* user effort for a given set of query results and show that it is NP-Hard and present efficient heuristic algorithms to compute an approximate subset that minimizes estimated expected effort. We evaluate our approaches on a bibliographic dataset from MEDLINE in which each result is annotated with terms from the MESH concept hierarchy and show that it significantly outperforms state-of-the-art approaches. Our approach was presented as a paper at ICDE-2009[25] and as an idea demonstration at SIGMOD-2010[26]. An extended version was published in the journal IEEE-TKDE[27].

3. Cost-Driven Exploration of Faceted Query Results

In Chapter 4 we propose navigation and cost model of a user exploring a faceted result-set. This method works by computing a small set of facet-conditions, across all facets of a query result, which minimize the user effort in navigating a set of query results. The analogous models proposed in Chapter 3 work for results that are categorized by a concept hierarchy or taxonomy and are not suitable for a faceted result-set in which there are multiple ways of categorizing and filtering a set of query results and therefore require a more sophisticated model to capture and estimate user effort.

We prove that the problem of computing a facet subset that minimizes the effort is NP-Hard and present efficient approximate algorithms to compute an approximate effort minimizing subset. We present the results of an extensive experimental evaluation of the system using real-life datasets and a comprehensive user study that demonstrates the efficiency and effectiveness of our approach. This approach was published as paper at CIKM-2010[28].

4. Comprehension-Based Snippets

In Chapter 5, we look into the problem of result snippets which are used by most search interfaces to preview query results. We propose novel techniques to construct snippets of structured heterogeneous results, which not only select the most informative attributes for each result, but also minimize the expected user effort (time) to comprehend these snippets. We create a comprehension model to quantify the effort incurred by users in comprehending a list of result snippets which is supported by an extensive user-study. We present efficient approximate algorithms, and experimentally demonstrate their effectiveness and efficiency. This work was published at CIKM-2012[29].

Chapter 2

Effective Query Formulation on Complex Catalogs

2.1 Introduction

Keyword queries have limited expressivity and are inherently ambiguous and therefore return a large number of results of different types (categories), many of which are irrelevant to the user. As an example, consider a user searching for a laptop who submits the query *'asus laptop 3gb'* on Amazon.com. As a first step, the system returns a hierarchy of all categories relevant to the query. Figure 2.1(a) shows the state of the hierarchy navigation after the user selects the *Electronics* category. As seen in the figure, due to ambiguity of keywords, the query returns results from several unrelated top-level categories such as *Computer&Accessories*, *Television&Video* and *Camera&Photo*. The category of interest, namely *Laptops*, is buried under the category *Computer&Accessories* (Figure 2.1(b)). Also, the laptops returned (not seen in the figure) do not all have *3Gb* of main memory nor are exclusively manufactured by *Asus*.

To reach the results of interest (*Asus Laptops* with *3GB RAM*), the user navigates this hierarchy in a *step-by-step* manner. The first series of steps involves navigating *down* the hierarchy to locate the category of interest. At each step, the system displays attribute facet conditions for the items in this category (Figure 2.1(b)). After locating the category of interest, the user selects some

conditions to further narrow down the results. Figure 2.1(b) shows the facets after the user selects category *Laptops* and then attribute condition *Brand=Asus*. Note how *Laptops*-specific attributes such as *Ram Size* and *Display Size* are revealed.

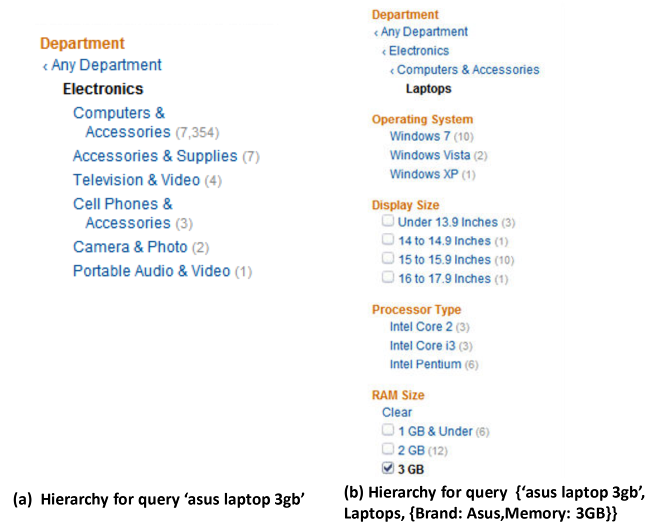
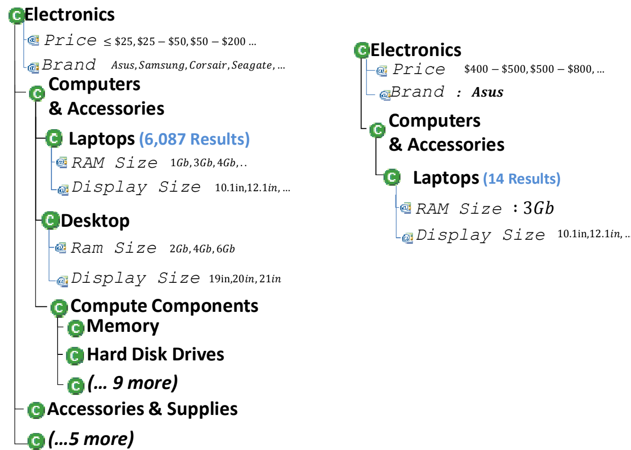


Figure 2.1. Navigation states, before and after refinements.

The user proceeds in this manner, until she has narrowed down the results sufficiently to reach the results of interest. The resultset in Figure 2.1(b) is much smaller (14 results) and the user can read them one by one to select the ones that satisfy her information need. Figure 2.2(a) shows the initial navigation hierarchy for query *'asus laptop 3gb'*, and Figure 2.2(b) shows the navigation hierarchy corresponding to the query state in Figure 2.1(b).

Instead of the two stage navigation described above (navigate hierarchy, then select category-specific attribute conditions), the system could directly suggest the most suitable categories namely, *Laptops* or *Netbooks,* and key attribute conditions for these categories like *Brand=Asus* and *Ram Size=3GB* to add to the original keyword query. Augmenting keyword queries with structured conditions helps disambiguate the query intent and create focused queries that return *only* the relevant results for the user.



(a) Navigation Hierarchy for query in Figure 1a. (b) Navigation Hierarchy for query in Figure 1b.

Figure 2.2. Navigation hierarchies of states (queries) in Figure 2.1.

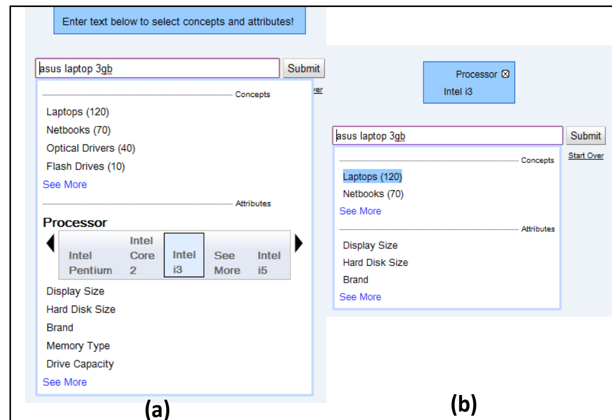


Figure 2.3. KWalker Query Interface.

KWalker: In this chapter, we propose KWalker which adaptively suggests the most promising categories along with the most promising attributes and conditions to add to the original keyword query. Figure 2.3 shows an example of our KWalker interface that enables this process. As the user is entering keywords, KWalker responds almost instantaneously, with contextual category and attribute conditions that can be added to the query. Figure 2.3(a) shows the suggestions generated by KWalker for the query *'asus laptop 3gb'*. As seen in the figure, KWalker generates a set of Category suggestions (*Laptops, etc.)* and a set of Attribute suggestions based on

category suggestions (*Processor* etc.). Once the user selects a suggestion, KWalker dynamically updates the list of suggestions to iteratively refine the query, as shown in Figure 2.3(b). For example, if the user selects condition *Processor=Intel i3* (Figure 2.3(b)). The user can either choose one of the suggestions or enter additional keywords to receive more suggestions. The user proceeds in this manner until she is satisfied with the constructed structured query and finally submits it.

The number of candidate suggestions to add to a keyword query on categorized hierarchical databases is quite large, due to the multiplicity of category and attribute conditions. To help users formulate a focused query using KWalker, we need to answer the following question: *What constitutes a good suggestion?* Specifically, we want to determine the good category suggestions, since attribute conditions can be generated based on them.

Commonly Used Approaches: A first solution would be to suggest all leaf categories that contain matching items, that is, items that contain all query keywords. However, this set can be too large, even for specific queries. For instance, for query '*asus laptop 3gb'*, there are more than 100 leaf categories in Amazon.com including *Servers*, *External Data Storage* and *Carrying Cases*. The reason is that query keywords can match any field of the product, including its reviews. Another commonly used approach is constructing suggestions based on the Lowest Common Ancestor (LCA) [30]. LCAs would be very effective in selecting individual matching results but not to suggest categories for navigation. For example, in Figure 2.2(a) the LCA of *'asus laptop 3gb'* is the root node *Electronics*, which is too general to be an effective suggestion that reduces navigation effort.

Highest Common Descendant (HCD): We introduce the novel concept of *Highest Common Descendants (HCDs)*, which are the categories with "just the right" amount of specificity,

given the query. That is, HCDs are not too specific or too general. For example, one of the HCDs for the query *‘asus laptop 3gb’* in Figure 2.2(a) is *Laptops* since it contains ‘*laptop*’ on its name, *‘3gb’* on its attributes and *‘asus’* on the attributes of its parent, *Computers&Accessories*. On the other hand, *Computers&Accessories* is not an HCD since it does not contain ‘*laptop*’ or *‘3gb’* on itself or its ancestors (according to the HCD definition (Section 2.3.1), at least one keyword must be contained on itself). Intuitively, *Computers&Accessories* is a bad suggestion for the user, since we can identify more specific categories that match the query. Children of *Laptops* are not HCDs either, since they are subsumed by *Laptops*; this avoids having huge number of possibly irrelevant HCDs. That is, HCD represents the “right granularity” of categories to display to the user. In some cases, the number of HCDs can still be too large to be displayed to the user. KWalker ranks the HCDs by estimated number of results and displays a small number of high ranked HCDs.

2.2 Framework and Definitions

Definition 2.1 (Category Hierarchy): The category hierarchy consists of a collection of *categories* $C = \{c_1, \dots, c_m\}$. Each category $c \in C$ is associated with a (possibly empty) set of *attribute*s $A_c \subseteq A$, where A is the set of all attributes used in C. An *attribute* $A_i \in A$ is associated with a domain $Dom(A_i)$ of un-interpreted constants. A database schema arranges the categories in C in a hierarchy $H[C]$.

In our framework, we consider *tree* hierarchies $T[C]$, where any category c can be a direct sub-category of at-most one category d. A DAG can be converted to a tree in a straight-forward manner by recursively replicating sub-trees with multiple parents, as is done in practice (e.g., gopubmed.com[31] , Amazon.com[1]). This is done because graph visualization does not scale well to large datasets and navigation over trees is more intuitive than general DAGs [32].

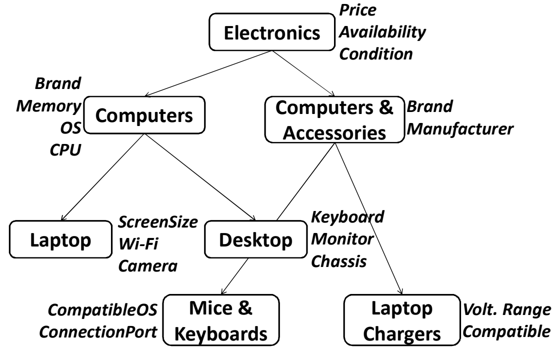


Figure 2.4. An Example Navigation Hierarchy.

Example 2.1: Figure 2.4 shows an example of a portion of tree hierarchy T[C] of 7 categories rooted at Electronics. Here, *Laptops* and *Desktops* are direct sub-categories of *Computers* and are sub-categories of the root *Electronics* category.

Definition 2.2 (Database): The *database D* is a collection of heterogeneous *data items* $D = (t_1, \dots, t_n)$. Each data item $t(id, c) \in D$ consists of an identifier *id* and a leaf category $c \in T[C]$. Further, $t(id, c)$ has attributes A_c.

Definition 2.3 (Data-Tree Model): To simplify the discussion below, we combine the category hierarchy $T[C]$ and the database *D* into a single abstract representation, called *data-tree* T_D. (Note that our work can be applied to any physical storage of the data.) According to the semantics of the sub-category relationship, we say that data item t *belongs* to every super-category of c, including *c* itself. A *data-node* $t[c]$ is the tuple of a data item t in category c, that is, the attribute values of *t* for the attributes A_c of *c*. Further, $t[c][a]$ is the value of attribute $a \in A_c$ of the tuple $t[c]$.

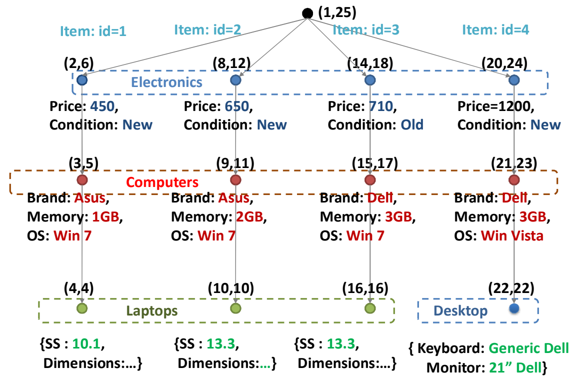


Figure 2.5. Data-Tree representation of a hierarchically organized dataset

Example 2.2: Figure 2.5 shows a data-tree, where three tuples (with *id*s 1, 2 and 3) belong to the leaf category *Laptops*, and the fourth (with *id* 4) belongs to the leaf category *Desktops*. For items with *id*s 1, 2 and 3, there is a data-node for the *Laptops, Computers* and *Electronics*. The *Computer* data-nodes store attribute values for *Brand, Memory* and *OS*, while *Laptop* data-nodes store *ScreenSize (SS)* and *Dimension* information.

Definition 2.4 (Query Model): The aim of KWalker interface is to help users add structured conditions to keyword queries. A *keyword query* $Q = (q_1, \dots, q_s)$ is a set of keywords that the user enters, such as *'asus laptop 3gb'*. The result $R(Q)$ of Q consists of all items in *D* that contain all the keywords in one of their categories (i.e., the categories they belong to), their attribute names and attribute values. Starting from the keyword query, the KWalker interface constructs, by interacting with the user, a *structured query* which consists of a keyword query augmented with *structured conditions.* A *structured condition,* which is a KWalker suggestion, is either a category $c \in T[C]$ or a predicate over an attribute $A_i \in A$ of the form $(A_i\ op\ v_i)$, where op is an operator (we use equality in our implementation) and v_i is a value. Structured conditions are combined conjunctively with the keyword query.

A *structured query* Q_S is a keyword query Q augmented with a set of structured conditions and has the following form:

$$Q_S = (Q, c \in (T[C] \cup \emptyset), AC = \{\ldots, (A_i = v_i), \ldots\})$$

where c is a category and AC is a set of attribute conditions. Note that Q_S only has one category condition; because typically, a user is interested in results of a particular category.

The result $R(Q_S)$ of $Q_S \coloneqq (Q, c, AC)$ is the set of all data items that satisfy the keyword query Q and belong to c and satisfy all the attribute conditions in AC. If c is undefined ($\emptyset$), then all items under the root are considered.

Example 2.3: The query result of the structured query (asus laptops, $Laptops$, $\{Brand =$ 'Asus'$\}$) on the sample dataset in Figure 2.5, will be data items with *id*s 1 and 2 since they both belong to *Laptops* and satisfy the attribute conditions of the query ($\{Brand =$ 'Asus'$\}$).

2.3 Structured Suggestions

In this section, we describe the suggestions generated by KWalker. We present category suggestions in Section 2.3.1 and attribute suggestions in Section 2.3.2.

2.3.1 Category Suggestions : HCD

Any category that has query results that belong to it is in principle a candidate category suggestion. For example, any of the 230+ categories in Figure 2.2(a) can be used to focus the query *'asus laptop 3gb'*. However, the number of such categories can be very large. To prune the list of relevant suggestions to include only *meaningful* categories at the right granularity, we follow a strategy based on the following observations:

1. Intuitively, the keywords entered by the user are a *partial* indication of a user's search goal. For example, the user searching with keywords *'asus 3gb'* could be potentially interested in, among other things, *Laptops* or *Desktops* with *3GB* of *RAM* or in *3GB MemorySticks* compatible with *Asus Laptops*. The set of such leaf categories that match such a query can be very large in practice as discussed in Section 2.1. To avoid this, we eliminate from consideration all categories that do not have matching keywords (on their label or attributes), thereby precluding too specific categories, with low confidence of matching the user's search intent.

2. On the other hand, we do not want to suggest categories that are too general. For example, for query *'asus laptop 3gb'*, *Computers&Accessories* in Figure 2.2(a) is too general, because it does not match any keywords ('asus' matches its parent, *Electronics*, 'laptop' and '3gb' match its child *Laptops*). Instead, *Laptops*, is a better suggestion for this query.

We formalize these aforementioned criteria with the notion of *Highest Common Descendent (HCD)*. Formally, an HCD of a keyword query Q is defined as follows:

$$HCD(Q) = \left\{ c \,\middle|\, \exists t \in R_c(Q), q_i \in Q \begin{pmatrix} occursIn(q_i, \mathrm{t}[\mathrm{c}]) \wedge \forall q_j \in \{Q - q_i\} \\ \left(\exists \mathrm{c} \prec^{*} d\left(occursIn(q_j, \mathrm{t}[d])\right)\right) \end{pmatrix} \right\}$$

where $R_c(\mathrm{Q})$ are the results of Q that belong to c, and $occursIn(q, t[c])$ is true if term q appears in the attribute names or values of t[c], or on the label of c. That is, a category c is an HCD if there is a result item t that belongs to c (else t[c] is undefined) and one of the keywords matches on c (on its label or attribute names or values for t on c).

Intuitively, HCDs are the roots of the sub-trees that are relevant to the query Q. Every data item returned by Q belongs to at least one HCD. To keep the number of suggestions manageable,

we rank category suggestions in decreasing order of (estimated) cardinality of results and only show at most k (e.g. 5) suggestions.

2.3.2 Attribute Suggestions

HCDs help in focusing the query Q to a specific sub-type of results, while attribute-conditions help in focusing the query to a specific aspect of the chosen category. Therefore, we suggest attributes, and corresponding attribute conditions, from the HCD categories. However, we also include attributes on ancestors (super-categories) of HCDs because the ancestor categories contain attributes relevant to the data items of the HCD categories. For instance, for the keyword query *'asus 3gb'* in Figure 2.5, the candidates are all the attributes in HCD *Laptops* (*ScreenSize* etc.) and its ancestors *Computers (Brand, Memory and OS)* and *Electronics* (*Price* and *Condition*). As in the case of HCD suggestions, the list of attribute suggestions can also be quite large, and we suggest a small list of attributes. However, we use the following approach to compute the benefit of attributes:

Let $|D_Q(a = v)|$ denote the number of data items in the result of query Q with the value v for attribute a. The indiscrimination score[33] of values of a, denoted by $ID^V(a)$, is:

$$ID^V(a) = \frac{1}{|Dom(a,Q)|} \sum_{v \in Dom(a,Q)} \frac{|D_Q(a = v)|\left(|D_Q(a = v)| - 1\right)}{2}$$

where *Dom(a,Q)* is the subset of *Dom(a)* for the results of Q.

The $ID^V(a)$ score differs from the one in [33], in that we divide by the total number of conditions for each attribute to normalize the scores to prevent attributes with a large number of conditions from having a high score. We order attributes by increasing values of indiscrimination score.

Example 2.4: The query result of the structured query (asus laptops, $Laptops$, $\{Brand =$ 'Asus'$\}$) on the sample dataset in Figure 2.5, will be data items with ids 1 and 2 since they both belong to *Laptops* and satisfy the attribute conditions of the query ($\{Brand =$ 'Asus'$\}$).

2.4 Computing Highest Common Descendants

The suggestions generated by KWalker are HCDs or attributes on HCD categories. These suggestions, as described in Section 2.3, can be directly computed using the results of the query. However, the large processing times precludes this straightforward approach for our system, which generates suggestions *on-the-fly*, before the query is executed. In this section, we describe techniques and algorithms to efficiently identify and compute Highest Common Descendants (HCDs). These approaches are based on indexing the data-tree (Section 2.4.2) and employing efficient algorithms to compute HCDs. Section 2.4.1 describes the Keyword Inverted List (KIL) index structure that is used to identify nodes in the *interval encoded* data-tree that contain the query keywords. In Section 2.4.2, we present an efficient Top-k pipelined algorithm to compute HCDs.

2.4.1 Keyword Inverted List (KIL) Index

The Highest Common Descendant (HCD) suggestions offered by KWalker are the most prominent categories (in terms of cardinality) in the results of the query that contain all the keywords on self or its ancestors. Computing HCDs on the data-tree database model (Section 2.2) involves checking the ancestor-descendant relationships amongst nodes in the data-tree that contain keywords. To efficiently support this operation, the data-nodes in the data-tree are encoded using interval encoding[34]. Using this encoding, a number of queries related to ancestor-descendent relationships can be efficiently answered. Such queries include finding all contained (descendants) or containing (ancestors) intervals of a given interval representing a data-node. The data-tree in

Figure 2.5 is encoded using interval encoding. The index structures proposed in this section help quickly locate nodes in the data-tree that contain a given keyword $q_i \in Q$. The inverted list for a keyword q_i consists of a list of entries $(e_1, \dots, e_r)$, one for each category that has the keyword and each entry is organized as follows:

$$[< id >; tf; < nodeList >]$$

- id: The identifier (name) of the category.
- tf: The *term frequency* or the total number of items classified into the category with identifier id that contain the keyword q_i.
- $nodeList$: A list of intervals identifying the data-nodes that contain the keyword q_i.

Example 2.5: The KIL entries for some keywords in the data-tree of Figure 2.5 are as follows:

$$KIL(asus) \rightarrow ([Computers; 2; (3,5), (9,11)])$$

$$KIL(laptop) \rightarrow ([Laptop; 3; (4,4), (10,10), (16,16)])$$

$$KIL(dell) \rightarrow ([Computers; 2; (15,17), (21,23)], [Desktop, 1, (22,22)])$$

The elements of the inverted list are sorted in decreasing order of tf. The number of entries in the inverted list of each keyword is not large, since it is bounded by the number of categories. However, the total number of interval entries in the *nodeList* can be quite large resulting in increased access and processing time. In Section 2.5, we describe techniques to reduce the size of these *nodeLists*.

Indexing KILs using Interval Tree Index: As we will see in Section 2.4.2, computing HCDs efficiently involves a join (in the relational sense) over the inverted lists $(L_1, \dots, L_n)$ of keywords in

the query Q. This join operation involves matching entries in the *nodeList* of a KIL entry for a keyword q_i with *nodeList* entries in other lists by ancestor-descendant relationships. To support such operations, we use an Interval Tree index over union of intervals in the *nodeList* of each keyword. For structured queries the process is identical except that an extra filtering step is required when reading from the KIL lists to check if the structured conditions are satisfied.

2.4.2 TopkHCD: Efficient Computation of HCDs

The HCDs and their corresponding cardinality for a given set of keywords can be computed by joining the inverted lists of query keywords. This join operation is essentially a merge-join over the inverted lists using ancestor relationship, where intervals in $nodeList$ of an entry in the inverted list ($say\ L_i$) are matched with ancestors in other lists $\left(L_{j\neq i}\right)$. The cardinality of intervals that match gives the cardinality of the HCD. As an example, consider the following KIL entries for the data-tree in Figure 2.5:

$$KIL(laptop) \rightarrow [Laptop; 3; (4,4), (10,10), (16,16)]$$

$$KIL(asus) \rightarrow [Computers; 2; (3,5), (9,11)]$$

To determine if the category $Laptop$ is an HCD for a keyword query *'asus laptop'*, the entries in *laptop* are *join*ed with entries of *asus*. Here two entries $(4,4)$ and $(10,10)$ have ancestors $(3,5)$ and $(9,11)$ respectively in the entry for $Computers$, therefore $Laptop$ is an HCD with cardinality 2. This procedure is very inefficient since it evaluates all entries in inverted lists of all keywords before returning the results, which can be large. However, we only need a small number ($k \leq 5$) of category suggestions ordered by cardinality and it is sufficient to retrieve and compute cardinalities only these categories.

To efficiently compute HCDs we build upon the Threshold Algorithm (TA) [35] for evaluating Top-k queries in scenarios where partial scores of items are available in m lists $(L_1, \ldots L_m)$ and the total score for an item I, $score(I)$ is computed using a monotonic aggregate function $score(I) = f(x_1, \ldots, x_m)$ that combines the partial scores of I from m lists of partial scores. The primary advantage of TA is that it computes Top-k items without (necessarily) fully processing the lists of partial scores and thus saves on computation and access costs. TA works by computing a threshold, at every step, which is an upper bound on the total score of items unseen by the TA. This threshold allows for *early-stopping,* and the algorithm terminates after seeing k items that have total scores, at-least as high as the threshold.

Algorithm: TopkHCD($L_1, \ldots, L_n$, k)

Input: Inverted lists $L_1, \ldots, L_n$, one for each keyword and k, the max number of HCDs

Output: A list of (at-most) k of HCDs and their cardinality.

---

1. Do sorted access on the lists in parallel to retrieve top entries $(e_1, \ldots, e_n)$.
2. **foreach** new category C retrieved in $(e_1, \ldots, e_n)$:
3. $card_C \leftarrow 0$
4. **foreach** interval $i \in e_c.nodeList$
5. Probe interval indices of all $L_{i \neq c}$ for ancestors and increment $card_C$ if an ancestor is found in all lists $L_{i \neq c}$
6. Set the threshold $t \coloneqq \max_{1 \leq i \leq n}[e_i.tf]$
7. **Stop** and return if the Top-k set has k items with cardinality at-least equal to the threshold t.
8. **Goto** Step 1.

Figure 2.6. TopkHCD Algorithm.

The Top-k algorithm for computing HCDs using KIL lists of n keywords is shown in Figure 2.6. In the first step (line 1) the algorithm retrieves the top entries $(e_1, \ldots, e_n)$ from each of

the n lists. For each category c retrieved, the algorithm computes its cardinality by probing for ancestors of data-node intervals in the $nodeList$ of the entry e_c of c (lines 2-5). To utilize the *Threshold Algorithm* effectively, it is necessary to define a threshold. It is important to note that TA assumes a monotonic function, whereas cardinality computation, in our scenario is not necessarily monotonic, since a category with a high tf might have data-nodes that do not have any ancestors in inverted lists of other keywords. However, since we order entries in KILs by tf, any HCD category retrieved in subsequent steps can have cardinality of at most $\max_{1 \le i \le n}[e_i.tf]$ and we use this value as the threshold. This value of the threshold could be an overestimate and the algorithm may process more nodes than necessary. However, this overestimation does not affect correctness and the algorithm always returns k HCDs with highest cardinality. For structured queries, the process is identical except that an extra filtering step is required when reading from the KIL lists to check if the structured conditions are satisfied.

2.5 Synopsis and Query Estimation

The techniques described in Section 2.4, even though efficient, operate on the entire database (data-tree) which makes them unsuitable for near instant-time user interaction promised by our system. In Section 2.5.1, we present summarization (synopsis) techniques that allow us to compute suggestions on a small but highly accurate summary of the data, based on the summarization model presented in [36]. In Sections 2.5.2 and 2.5.3, we show how we modify the KIL Index and the Top-k HCD Algorithm to work with the synopsis model. Note that this version of the Top-k HCD algorithm returns an approximate list of HCDs with estimated cardinalities. We evaluate the accuracy of the synopsis-based suggestions in Section 2.6. Since we use the synopsis model in [36], we refer the reader to the aforementioned paper for a detailed description about the synopsis.

2.5.1 Background and Our Synopsis Model

To speed-up computation of HCDs, their cardinalities and other quantities for suggestions described in Section 2.3, we evaluate suggestions on a compact synopsis or a summary of the database. Since our data-model is a heterogeneous tree of data items, we adopt the XCluster synopsis [36], which summarizes heterogeneous and hierarchical XML data, as the basis of our synopsis model.

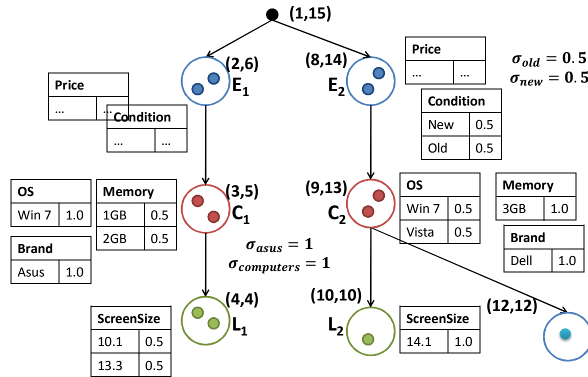


Figure 2.7. Database Synopsis of Data-Tree in Figure 2.5.

Data Tree Synopsis: The idea is to *summarize* data-nodes in the data-tree T_D (e.g., T_D in Figure 2.5) that are very *similar* to each other, both in hierarchy and attribute values, into the same partition node[36]. Each partition node stores aggregate information of all data-nodes grouped under it, thereby creating a synopsis T_S that approximates T_D. For each partition node c_P of T_S, the synopsis (akin to the XCluster Synopsis [36]) stores the count $count(c_P)$ of data-tree nodes that are grouped into c_P and a collection of histograms, one per attribute based on the category hierarchy type, that approximates the distribution of values of an attribute in a given partition node.

Example 2.5: Figure 2.7 shows the synopsis of the data-tree in Figure 2.5. Here, the 13 data-nodes have been compacted into a summary consisting of 7 nodes. Each partition node now summarizes one or more nodes of the data-tree, of the same category type. For example, nodes L_1

and L_2 summarize the 3 *Laptop* Nodes of Figure 2.5. The figure also shows the single attribute histograms which store the attribute values and their corresponding selectivity.

Evaluation of HCDs and their cardinalities on synopsis can be done by considering the aggregate information stored in the partitions and by using appropriate independence assumptions to estimate these values where information stored in synopsis is in-sufficient [36]. As an example, consider the computation of cardinality of *Laptops* for the query '*asus laptop'* on the synopsis in Figure 2.7. There are two partitions of category *Laptops,* L_1 and L_2, therefore the total cardinality is: $\sigma_{(asus,laptop)}(L_1) \cdot count(L_1) + \sigma_{(asus,laptop)} (L_2) \cdot count(L_2)$, where $\sigma_{(asus,laptop}(c_P)$ is the selectivity of the keywords in partition c_P. The joint selectivity of keywords (asus, laptop) can be estimated as $\sigma_{(asus,laptop)}(c_P) = \sigma_{asus}(c_P) \times \sigma_{laptop}(c_P)$ by assuming independence of distribution of keywords amongst the partitions. Now, since '*asus'* appears in the parent category of *Laptops* and to estimate its selectivity in a *Laptop* partition we can use the *path-value independence assumption*[36], which assumes independence between the value distribution in partitions of a parent (e.g. C_1) and those of its children (L_1). In our running example, the selectivity of the keyword query *'asus laptop'* can instead be estimated on parent partitions of type *Laptops* as follows: $\sigma_{asus}(C_1) \cdot \sigma_{laptop}(L_1) \cdot count(L_1) + \sigma_{asus}(C_2) \cdot \sigma_{laptop}(L_1) \cdot count(L_2) = 1 \cdot 1 \cdot 2 + 0 \cdot 1 \cdot 2 = 2$.

Typically, these independence assumptions are unreasonable and may cause huge errors during cardinality estimation of HCDs and for estimating distribution of attributes for a given query. However, these assumptions result in lower approximation errors if the partition-node c_P groups together *similar* (in data-values and hierarchy structure) nodes[37].

For performance purposes, we use a tree-synopsis, which is a simpler adaptation of the graph-synopsis generated by XCluster[36] for XML data. Graph-synopsis is more compact in terms of space but more expensive in terms of execution time, since multiple overlapping paths have to be considered. We adapt the algorithm to build the synopsis as described in [36] with the sole difference being that the data-tree is traversed bottom-up during synopsis construction.

2.5.2 KIL on Synopsis

The KIL Index of Section 2.4.1 is modified to store references to the synopsis partition nodes, instead of individual data-nodes. As described in Section 2.5.1, computing cardinalities using synopsis requires the selectivity (sel) of a given keyword in a partition in addition to its count (tf). Therefore, we now store two lists for each keyword. Each list consists of a list of entries ($e_1., \ldots, e_k$) with one entry for each category that contains the given keyword. The two lists are as follows:

- term-frequency list ($tfList$): each entry has category *id, tf* of keyword in the category, and list of partitions ($nodeList$) identifying partition nodes that contain the keyword. The list is ordered by descending category tf.

- selectivity list ($selList$): each entry has category *id*, the selectivity *sel* of keyword, and list of partitions ($nodeList$) identifying partition nodes that contain the keyword. The list is ordered by descending *sel*.

The organization of the *nodeList* is identical to that of KIL in Section 2.4.1, with one exception: we also store the *tf* or selectivity (*sel*) of the keyword in the partition, depending on whether the nodeList is for a $tfList$ or a $selList$.

Example 2.6: The KIL entries for some keywords in the synopsis of Figure 2.7 are shown below:

$$KIL(asus) \rightarrow \quad \begin{pmatrix} tfList: ([Computers, 2, (3{,}5; 2)]) \\ selList: ([Computers, 1.0, (3{,}5; 1)]) \end{pmatrix}$$

$$KIL(laptop) \rightarrow \quad \begin{pmatrix} tfList: ([Laptops, 2, (4{,}4; 2), (10{,}10; 1)]) \\ selList: ([Laptops, 1.0, (4{,}4{,}1), (10{,}10; 1)]) \end{pmatrix}$$

Also, to support random access, we create an interval tree index on both lists for each keyword, as described KIL of Section 2.4.1.

2.5.3 HCD Computation on Synopsis

Overview: Figure 2.8 presents S-TopkHCD, which is a modified version of TopkHCD (Section 2.4.2) to use synopsis. The algorithm does sorted and random accesses on the *tfList*s and *selList*s of the query keywords, in a manner similar to the way TopkHCD accesses the KIL list. The cardinalities of HCD categories are computed using the independence assumptions outlined in Section 2.5.1. A key intuition, from Equation 2.4, is that to estimate the cardinality of a category, we multiply its *tf* for one of the query keywords with the maximum selectivity for the other keywords of ancestor category. For each category *c* retrieved from a *tfList* we check if it is an HCD by doing random access (using the interval tree index) on the *selList*s of the other keywords to find ancestors of c that contain the other keywords. Similarly, for each category retrieved from a *selList*, we do random access on the *tfList*s of other keywords to find descendants that may be HCDs.

Cardinality estimation: The TopkHCD algorithm in Section 2.4.2 computes the cardinality of a category c in the data-tree T_D by counting the number of paths from the $root(T_D)$ to data-nodes of a category c that have all the keywords in $Q = \{q_1, .., q_n\}$. Since each item is along a different path in T_D, this technique accurately computes the cardinality of items for a

category c. In a synopsis T_S of T_D, multiple data-nodes of T_D of a given category c are merged into a single representative node, thereby collapsing the paths to the items the partition summarizes.

Algorithm: S-TopkHCD $(L_1, \ldots, L_n, k)$

Input: inverted lists $L_1, \ldots, L_n$, one for each query keyword k_i where each L_i consists of two lists, *selList* and *tfList*, and the number k of requested results.

Output: k HCDs with highest cardinality

// Let e_{is}, e_{it} be the last retrieved entries from the *selList, tflist* of k_i

1. Do a sorted access on the lists in parallel to retrieve entries $e_{1s}, e_{1t}, \ldots, e_{ns}, e_{nt}$ for each keyword.
2. For each new category c retrieved do one or both the following
3. **case 1:** if c is from the $tfList$ of L_i then
4. **foreach** entry e in the partitions of c in the $tfList$ of L_i:
5. Do a random access on the $selList$s of all $L_{j \neq i}$; looking for ancestors with maximum (for each keyword) selectivity.
6. Compute score using equation (2.1) and add to Top-k List
7. **case 2:** if c is from the selectivity list of L_i then
8. **foreach** a in $1 \leq a \leq n; a \neq i$
9. Do a random access on *tfList* of L_a looking for descendants of c. Let this list be c_{cand}.
10. **foreach** c' in c_{cand} do
11. Compute the scores of c' by joining with selectivity list of all $j(\neq k, \neq a)$ as in lines (5-6) above and add c' to Top-k list.
12. Set the threshold $t := \max_{1 \leq i \leq n}\left[e_{it}.tf \times \prod e_{js \neq i}.sel\right]$
13. **Return** if k^{th} item in Top-k list (ordered by score) has $score \geq t$.

Figure 2.8. S-TopkHCD Algorithm.

The cardinality $card(c,Q)$ of a category c for a keyword query Q, is now computed by aggregating the cardinalities, across all the partitions of type c:

$$card(c,Q) = \sum_{c_P \in partitions(c)} card(c_P, Q) \qquad (2.1)$$

where $card(c_P, Q) = count(c_P) \cdot sel(Q|c_P)$ and $sel(Q|c_P)$ is the selectivity of Q in c_P.

To compute $sel(Q|c_P)$, in addition to the *path-value independence assumption* [36], we observe that q_i can appear in multiple ancestors of the partition node c_P and hence we pick one ancestor node c_a of c_P per keyword that has the highest selectivity for the given keyword. The rationale for the assumption is that the tuples at other partition nodes with the same keyword will join with partition nodes with the highest selectivity for a given keyword and therefore will not affect the cardinality of the partition. Therefore,

$$sel(Q|c_P) = \prod_{q_i \in K} max_{\substack{c_a \in ancestors(c_P), \\ occurs-in(q_i, c_a)}} sel(q_i|c_a) \qquad (2.2)$$

where $occurs - in(q_i, c)$ means that q_i appears in partition *c*, and *ancestors(c)* is the set of ancestors of *c* including *c* itself. The cardinality of a partition c_P is therefore given by (from Equations 2.1 and 2.2):

$$card(c_P, Q) = count(c_P) \cdot \prod_{q_i \in K} max_{\substack{c_a \in ancestors(c_p), \\ occurs-in(q_i, c_a)}} sel(q_i|c_a) \qquad (2.3)$$

To use (2.3) for the sorted access on *tfList*s in S-TopkHCD, and particularly when we access c_P on the *tfList* of keyword q_i, we rewrite it as follows:

$$card(c_P, Q) = count(c_P) \cdot sel(q_i|c_P) \cdot \prod_{q_j \in Q-\{q_i\}} max_{\substack{c_a \in ancestors(c_p),\\ occurs-in(q_j, c_a)}} sel(q_j|c_a)$$

$$= tf(q_i, c_P) \cdot \prod_{q_j \in Q-\{q_i\}} max_{\substack{c_a \in ancestors(c_p),\\ occurs-in(q_j, c_a)}} sel(q_j|c_a) \qquad (2.4)$$

where $tf(q_i, c_P) = count(c_P) \cdot sel(q_i|c_P)$ is the term-frequency of keyword q_i in the partition c_P. When S-TopkHCD accesses c_P on the *tfList* of q_i it looks up the *selLists* of the other keywords to find the maximum selectivities of ancestor partitions of c_P.

Details: At the beginning of each iteration, the algorithm retrieves the top entries (categories) from each of the *2n* lists (line 1). For each new category retrieved, the algorithm computes the estimated cardinality (Equations 2.1 & 2.4) in a method analogous to TopkHCD in Section 2.4.2, by *join*ing with the other lists. Using Equation 2.4, we set the threshold to $t :=$ $\max_{1 \le i \le n}[e_{it}.tf \times \prod e_{js \ne i}.sel]$, i.e. the maximum possible cardinality of an (imaginary) HCD category which has all the keywords in the entries retrieved. It is easy to verify that any category retrieved in a subsequent iteration will have cardinality at-most t.

The score of the attributes is computed based on the value-distribution, which is computed by merging the histograms for the attribute across all partition nodes for the given query.

Table 2.1. Dataset Characteristics

	# paths	# attributes	# items	Dataset Size (XML)
Electronics	342	136	14,875	26.1 Mb.
Books	219	62	9,543	13.3 Mb.

2.6 Experimental Evaluation

In Section 2.6.1, we describe the experimental setup including the datasets used, the query workload and characteristics of the synopses structures. In Section 2.6.2, we describe the quantitative experiments to measure the accuracy and response times of the suggestions generated. Finally, in Section 2.6.3, we present a methodology for qualitative evaluation of KWalker and compare to existing faceted search approaches.

Table 2.2. Sample Query Workload

Query	Electronics Dataset	Books Dataset
Q1	toshiba laptop	harry potter children
Q2	intel laptop 3gb	biography Disraeli
Q3	dell optiplex	history time
Q4	dell laptop computers	autobiography
Q5	1TB hard drive	autobiography Franklin
Q6	dell intel processor	apple computer
Q7	asus	cooking baking bread
Q8	digital slr canon	recipe rice
Q9	sony recorder	business
Q10	seagate drive	business best practices

2.6.1 Experimental Setup

Datasets: We extracted a hierarchically categorized heterogeneous dataset of *Electronics* and *Books* products from a popular e-commerce website. The characteristics of the datasets are shown in Table 2.1. The *Electronics* hierarchy consists of 342 unique root-to-leaf paths (leaf categories) and 136 attributes distributed across these categories. Each category with attributes has

between 3 and 7 of them. The *Book* hierarchy is relatively sparse and contains 62 attributes across 219 categories. This is because the *Books* dataset has a large number of *Genre* categories.

Query Workload: A sample of the queries used in the experiments is shown in Table 2.2. The keywords appear in category names, attributes and attribute values. For example, query Q2 consists of keywords *laptop* which is a category name and *intel* and *3GB* which are attribute values for attributes *CPUType* and *RAM Size* respectively, which are in different categories in the hierarchy – *CPUType* is an attribute of *Computers* whereas *RAM Size* is an attribute of *Laptop.*

Synopsis Structure: The accuracy and performance of the synopsis depends on the space allotted. We built two synopses for the *Electronics* dataset in which the data-tree is compressed to 40% (Summary-1) and 70% (Summary-2) of its original size. For the *Books* dataset, we created a summary by compressing it to 50% of its original size.

2.6.2 Quantitative Experiments

In this section, we evaluate the impact of using a synopsis on the execution time and the accuracy of the category and attribute suggestions. We compare against suggestions generated with no summarization, which is termed the *reference synopsis*[36]. We report the results for *Electronics* dataset. The results for *Books* dataset are analogous and therefore omitted.

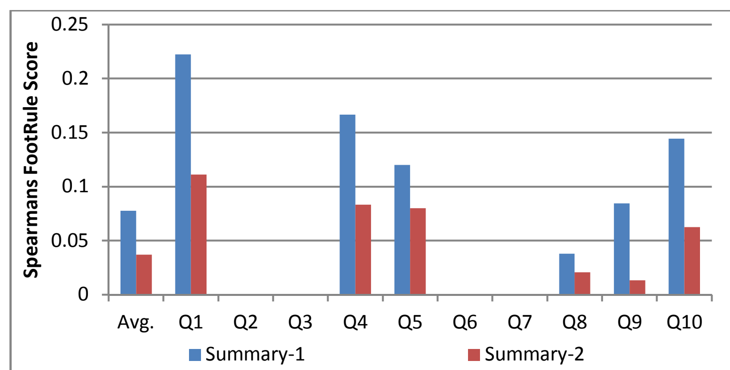


Figure 2.9. Top-k Spearman's Footrule score for Category Suggestions (Electronics Dataset)

Evaluation Metrics: To compare ordered lists of category or attribute suggestions, we use the extended normalized Spearman's footrule metric for Top-k lists [38]. Given two *Top-k* lists T_1 and T_2, the l −footrule distance, $F^l(T_1, T_2)$ is given by:

$$F^l(T_1, T_2) = \frac{1}{Z} \sum_{a \in T_1 \cup T_2} |T_1(a) - T_2(a)|$$

where $T_i(a)$ is the position of an element a in list T_i if a is present in T_i, and $l = (k + 1)$ otherwise and Z is the normalization factor.

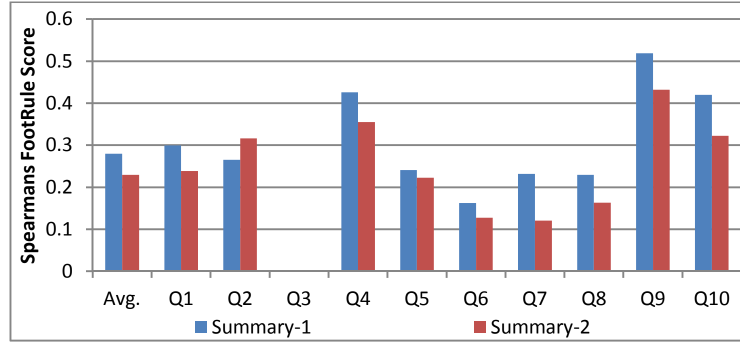


Figure 2.10. Top-k Spearman's Footrule score for Attribute Suggestions (Electronics Dataset)

In addition to the order, we evaluate the accuracy of the constructed synopses with the *average relative error* of estimates over the k HCD suggestions retrieved by the query. Given a query q and two lists of HCD suggestions T^c_{syn} and T^c_{ref} obtained by evaluating q over a *synopsis* and *reference* data-graph respectively, the average relative error is given by:

$$E_{avg}\left(T^c_{syn}, T^c_{ref}\right) = \frac{1}{\left|T^c_{ref} \cap T^c_{syn}\right|} \sum_{a \in T^c_{ref} \cap T^c_{syn}} \left(\frac{\left(\left|est_{ref}(a) - est_{syn}(a)\right|\right)}{est_{ref}(a)}\right)$$

where est_{ref} and est_{syn} denote the cardinality estimates of a category a under the summarized synopsis and *reference*, respectively.

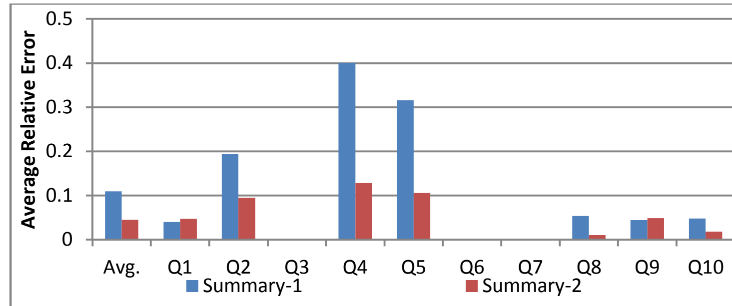


Figure 2.11. Average Relative Error for Category Suggestions (Electronics Dataset)

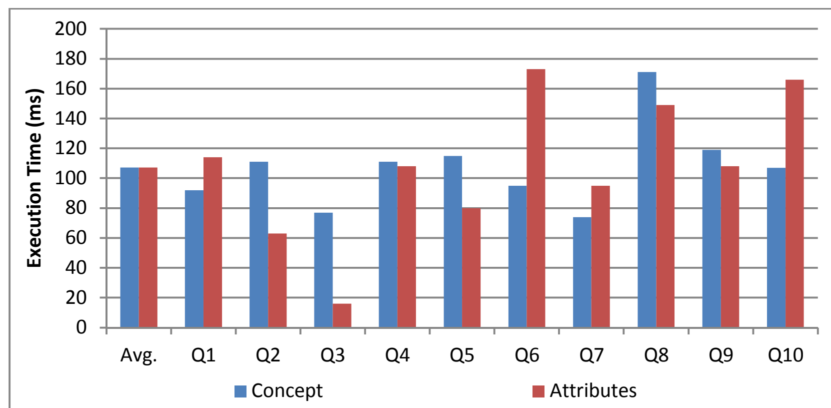


Figure 2.12. Average Execution Times (in ms) for Category and Attribute Suggestions (Electronics Dataset) (Summary-1).

KWalker Accuracy: Figures 2.9 and 2.10 show the Footrule scores of the Top-k category and attribute suggestions respectively. The Spearman's foot-rule scores for category suggestions is low (less than 0.08 on average), even for summaries that are compressed to 40% (Summary-1) of their original size. This shows that accuracy of Top-k lists does not vary even for small sizes of summaries. The scores for attribute suggestions (Figure 2.10) is much higher (about 0.28 on average for Summary-1 and 0.23 for Summary-2) and is as high as 0.51 for Q9 (Summary-1). Recall that the attribute suggestions are ranked based on the distribution of values in the query result and approximating the result on summaries causes higher errors for attribute suggestions than for HCD suggestions.

Figure 2.11 shows the average relative error for category suggestions for queries on the Electronics dataset. The errors on average range from 11% for Summary-1 to about 5% for Summary-2, which is low given rate of compression achieved in summaries. Also, as more space is allotted for summaries, the estimation error decreases as query approximations become more accurate. Furthermore, the accuracy in cardinalities is not as important as the relative order (Top-k) of suggestions.

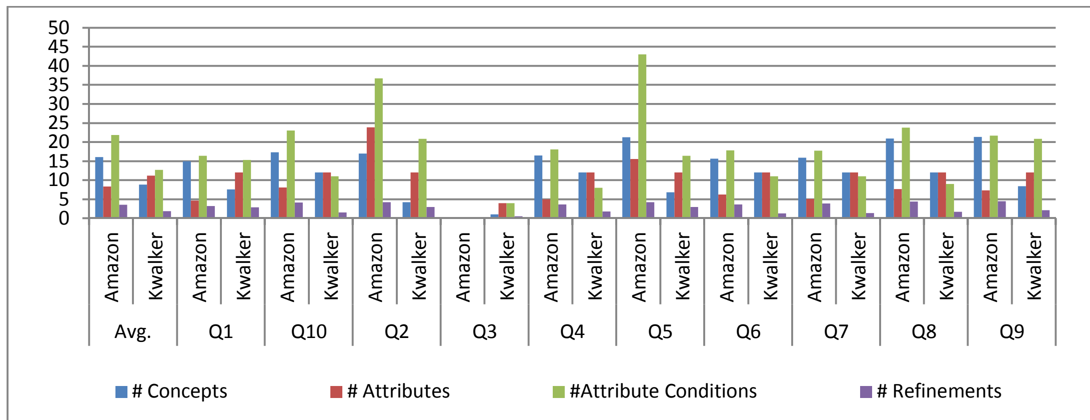


Figure 2.13. Experiments with Navigation Cost (Electronics Dataset).

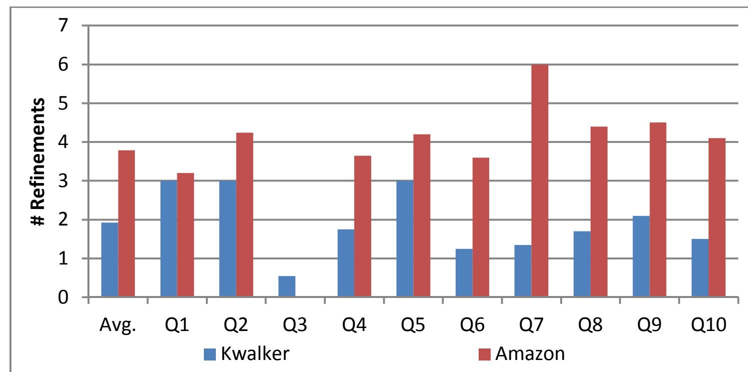


Figure 2.14. Average number of refinements for queries (Electronics Dataset).

KWalker Performance: Figure 2.12 shows the execution times (in seconds) of retrieving and evaluating categories and attributes suggestions for queries of the *Electronics* dataset, evaluated over Summary-1. The average execution times for both these summaries are low enough (~200ms on average for Summary-1) to support real-time interaction. We omit the execution times

over the *reference synopsis* which is large (avg. over 700ms). The drop in response times with summary synopses, as compared to reference synopsis, is due to the fact that computation of HCDs involves counting the number of paths in the data-tree and with summarization (Section 2.5.3) a number of these path are merged and therefore the HCD algorithms on synopsis evaluate far (nearly quadratically) fewer paths than those on the *reference synopsis*.

2.6.3 Qualitative Experiments

These experiments are designed to showcase the effectiveness and the ease of formulating structured queries using KWalker. Here we compare the navigation cost of KWalker against the only faceted interface we encountered (called *FacetedNav* henceforth) that combines both categories and attributes, and can be found most at commercial sites like Amazon.com[1]. Note that there is no previous research work on such faceted search.

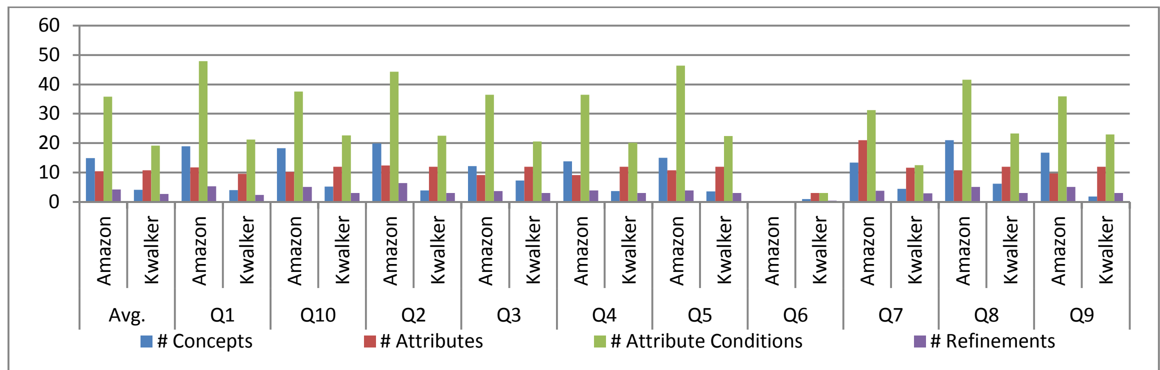


Figure 2.15. Experiments with Navigation Cost (Books Dataset).

FacetedNav: In response to a keyword query, this interface presents a list of results and a set of category and attributes conditions. Only the part of the hierarchy that contains one or more results is considered. The user navigates the hierarchy top-down starting from the root, that is, in the first step all the top-level categories are displayed. Further, all the attributes of the current category are displayed along with the at-most 10 conditions with the highest results cardinality for

the query. The user refines the query until the results list is short enough to inspect manually or if further refinements fail to refine the results. The cost of navigation consists of: (a) number of category labels, attribute names (b) the total number of times the query was refined by adding an attribute condition or drilling down the category hierarchy and (c) the total number attribute conditions displayed. Since we are comparing cost of query formulation, the number of returned results (which is constant for the two interfaces) is not included in the cost.

Simulation Setup: We assume that the *target* of a user exploration is a single result. For each query in Table 2.2, we designate randomly a result as the target of navigation and compare the cost of narrowing down to the given result using both the interfaces. Notice that a result can be reached my multiple paths. For example, the result with *id=3* in Figure 2.5 can be reached by navigating the hierarchy to the *Laptop* category from the root and inspecting the three results or by stopping at *Computers* and selecting '*Brand=Asus*', among others. The total number of paths can be quite large and each path has a different associated cost. To keep the number of paths manageable, we consider only 50 randomly chosen paths and only paths that lead to the target object are considered.

Results: The navigation cost incurred by FacetedNav and KWalker for the queries in Table 2.2 are shown in Figure 2.13 for *Electronics* and Figure 2.15 for the *Books* Dataset. The figure shows averages (across simulations) for each cost component viz. the number of categories, attribute and attribute conditions seen and the number of times the query was *refined* (by selecting a suggestion in FactedNav or HCD suggestions in KWalker). The overall navigation cost incurred by FacetedNav navigations is *higher* as compared to KWalker – by approximately 43% for the *Electronics* dataset. Note that KWalker reveals fewer categories during navigations as compared to FacetedNav. This is because KWalker suggests HCDs as refinement candidates instead of top-

down approach followed by FacetedNav and by selecting HCDs, the user is able to navigate to specific categories without encountering more generic (at higher levels) ones. For the same reason, navigations with KWalker require far fewer refinements as compared to FacetedNav (Figure 2.14) and therefore the simulation agent is able to construct a highly focused query in fewer iterations – on average KWalker required about 1.9 iterations as compared to 3.8 for FacetedNav. The query Q3 returns only one result and since a query with a small number (single page) of results is already quite focused and will not benefit from adding additional conditions. The number of query refinements for some queries (e.g Q1 in Figure 2.14) in FacetedNav is very close to those in KWalker. This is because the resultset for these queries is small and concentrated on a small section of the category hierarchy and therefore FacetedNav is able narrow down without too many refinements (Figure 2.13). The results for the *Books* dataset (Figure 2.15) are analogous to that of the *Electronics* Dataset and the discussion is therefore omitted.

2.7 Related Work

Keyword Search on (Semi-)Structured Data: Several works have studied keyword search on structured relational data [13, 14, 39-41] and semi-structured (heterogeneous) XML data [30, 42]. The goal of these works is to find how query keywords are connected to each other in the database, for example finding paths or subtrees (LCAs) that contain all the keywords. In Section 2.1, we discussed the problems associated with using LCAs as suggestions and proposed Highest Common Descendants (HCDs) as a superior alternative. Further, the problem of finding the HCDs is algorithmically fundamentally different from that of finding LCAs, as we show in Section 2.4. Advanced query forms [10, 11] are effective for searching in heterogeneous datasets. However, they would be tedious in our setting where the categories hierarchy is huge and the items are heterogeneous, that is, they have different attributes.

Keyword Query Enhancement and Query Refinement The GrowBag project [43] and Sarkas et. al [44] suggest additional search terms based on the co-occurrence patterns of these terms in the query result. These works neither consider structured attributes nor category hierarchies. Recently, some works[45, 46] proposed adding structure to keyword queries by annotating individual keywords with attribute names[46] or with terms from taxonomy[45]. The utility of these works is limited since they assume a single relation from which annotations are derived whereas we consider highly heterogeneous databases. Furthermore, these approaches generate full query suggestions, which can be large in number due to the ambiguity of keywords.

Query Autocompletion: Autocompletion has been adopted by a variety of applications such as Web search and e-commerce search. In most cases, autocompletion works by indexing a pre-defined set of strings and then suggesting keywords to be added to the query. Recently, there are a number of works on predictive autocompletion [47-49] which leverage information retrieval and machine learning techniques to suggest potential completions. These suggestions are based on aggregated user behavior on the results of the current query. Instead, KWalker suggests refining the query using categories and attribute conditions. The data model is also different. The demo system of Nandi and Jagadish [50] suggests structured conditions as completions of a given keyword in a way that the result will be non-empty. However, they do not consider a category hierarchy or the results' cardinality for each suggested attribute condition.

XML Summarization: The complexity and structure of synopsis used to summarize a dataset depends on its structure. For example, histograms[51] are sufficient to summarize flat relational data whereas summarizing XML is substantially difficult due to high structural variability. XML summarization has a number of variants such as path synopsis [52], structure-only [53] and structure-value synopsis [36, 54]. Our work models heterogeneous organized datasets

as XML and builds on the XCluster synopsis[36] to build compact summaries that can efficiently compute the quantities used to score suggestions.

2.8 Summary

We present a novel query formulation interface, KWalker to facilitate query formulation over hierarchically organized structured databases, which present unique challenges due to the heterogeneity of the stored objects. Starting with a keyword query, the user quickly formulates a structured query adding category and attributes conditions suggested by the interface. Efficient approximate algorithms are presented that use synopsis principles to estimate the benefit of each suggested condition, which is based on the number of result that this condition would return. The effectiveness of the solution is demonstrated with evaluation on real-life datasets.

Chapter 3

Results Navigation Using Concept Hierarchies

3.1 Introduction

As claimed in previous work [55], the ability to rapidly survey Biomedical literature constitutes a necessary step toward both the design and the interpretation of any large scale experiment. Biologists, chemists, medical and health scientists are used to searching their domain literature –such as PubMed– using a keyword search interface. Currently, in an exploratory scenario where the user tries to find citations relevant to her line of research and hence not known a priori, she submits an initially broad keyword-based query that typically returns a large number of results. Subsequently, the user iteratively refines the query, if she has an idea of how to, by adding more keywords, and re-submits it, until a relatively small number of results are returned. This refinement process is problematic because after a number of iterations the user is not aware if she has over-specified the query, in which case relevant citations might be excluded from the final query result. A substantial part of the chapter has been reprinted from and reformatted from our IEEE-TKDE paper [27] © IEEE 2011.

As an example, a query on PubMed for "cancer" returns more than 2 million citations. A more specific query, "breast cancer treatment", returns 111,000+ citations. Our running example

query for “prothymosin”, a nucleoprotein gaining attention for its putative role in cancer development, returns 313 citations. The size of the query result makes it difficult for the user to find the citations that she is most interested in, and a large amount of effort is expended searching for these results. Many solutions have been proposed to address this problem –commonly referred to as information overload [14, 18, 23, 56, 57]. These approaches can be broadly classified into two classes: ranking and categorization - which can also be combined. Ranking presents the user with a list of results ordered by some metric of relevance[14] or by content similarity to a result or a set of results [57]. In categorization [18, 23, 56], query results are grouped based on hierarchies, keywords, tags or attribute values. User studies have demonstrated the usefulness of categorization in finding relevant results of exploratory queries [19]. While ranked results are useful when the ranking function is aligned with user preferences or the result list is small in size, categorization is generally employed by users when ranking fails or the query is too “broad”[19].

BioNav belongs primarily to the categorization class, which is especially suitable for this domain given the rich concept hierarchies (e.g., MeSH [58]) available for biomedical data. We augment our categorization techniques with simple ranking techniques. BioNav organizes the query results into a dynamic hierarchy, the navigation tree. Each concept (node) of the hierarchy has a descriptive label. The user then navigates this tree structure, in a top- down fashion, exploring the concepts of interest while ignoring the rest. An intuitive way to categorize the results of a query on PubMed is by using the MeSH static concept hierarchy [58], thus utilizing the initiative of the US National Library of Medicine (NLM) to build and maintain such a comprehensive structure. Each citation in MEDLINE is associated with several MeSH concepts in two ways: (i) by being explicitly annotated with them, and (ii) by mentioning those in their text (see Section 3.7 for details). Since these associations are provided by PubMed, a relatively straightforward interface to navigate the query result would first attach the citations to the corresponding MeSH concept nodes

and then let the user navigate the navigation tree. Figure 3.1 displays a snapshot of such an interface where shown next to each node label is the count of distinct citations in the subtree rooted at that node. A typical navigation starts by revealing the children of the root ranked by their citation count, and is continued by the user expanding on or more of them, revealing their ranked children and so on, until she clicks on a concept and inspects the citations attached to it. A similar interface and navigation method is used by e-commerce sites, such as Amazon and eBay.

For this example interaction, we assume that some of the citations the user is interested in are available on the three indicated concepts corresponding to three independent lines of research related to prothymosin, and therefore the user is interested in navigating to these concepts. These include, “Histones”, which play a role in gene regulation and are essential for virus replication and tumor growth, “Cell Growth Processes” and “Transcription, Genetic”, a key process for synthesis and replication of RNA and thus plays an important role in the duplication of cancer cells.

The above static –same for every query result– navigation method is problematic when the MeSH hierarchy (or one with similar properties) is used for categorization for the following reasons:

First, the massive size of the MeSH hierarchy (over 48,000 concept nodes) makes it challenging for the users to effectively navigate to the desired concepts and browse the associated records. Even if we remove from the MeSH concept nodes with no citations attached to them, the 313 distinct query results for “prothymosin” are attached to 3,940 nodes, which is the actual size of the navigation tree in Figure 3.1. Combined with the fact that the MeSH hierarchy is quite bushy on the upper levels, this means that the user has to inspect, for example, a total of 152 concept nodes before she reaches the indicated concept “Histones”; a number comparable to the distinct citation count in the query result. A common practice [59] for hierarchy navigation is to show only a subset

of a node's children, which would be appropriate if only few nodes contain many results. Unfortunately, this is not the case for the MeSH navigation tree; most of the 98 children of the root in Figure 3.1 have many results (the first three shown have 310, 217 and 193).

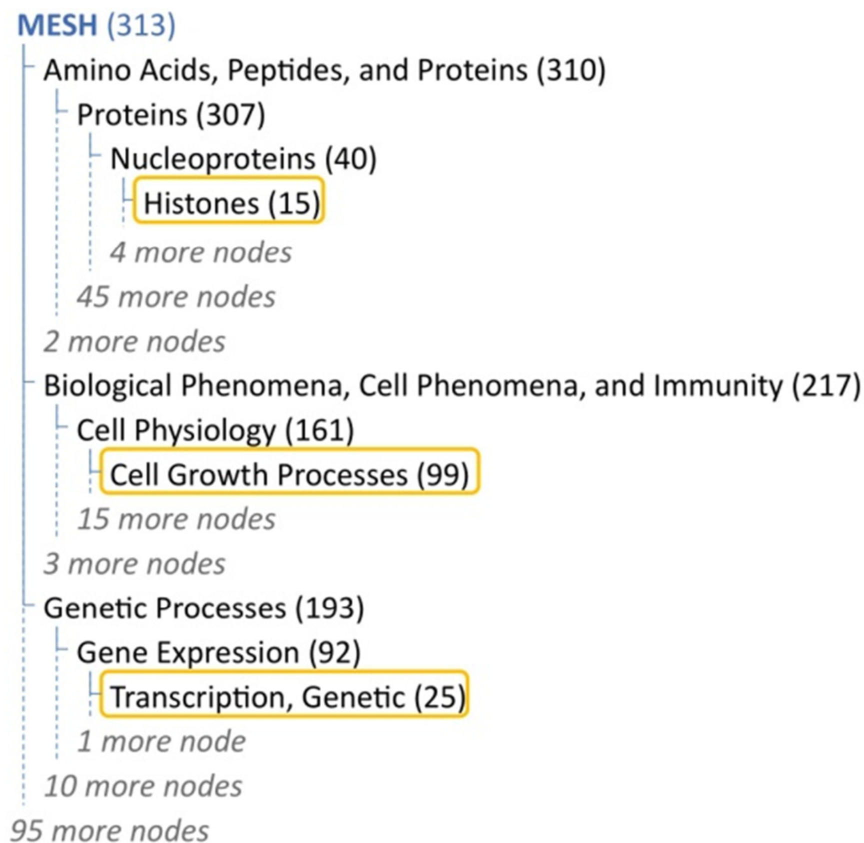


Figure 3.1. Static Navigation on the MeSH Concept Hierarchy.

Second, a substantial number of *duplicate* citations are introduced in the navigation tree of Figure 3.1, since each one of the 313 distinct citations is associated with several concepts. Specifically, the total count of citations in Figure 3.1 is 30,895. Naturally, the user would like to know which concepts fragment the query result into subsets of citations with as few duplicate citations as possible across them. Currently, the only way to figure this out using the interface in

Figure 3.1 is to click on different concept nodes and inspect the attached citations. As an example, the query results for "prothymosin" are related to three independent lines of research, represented by the three indicated concepts in Figure 3.1, which are hard to locate. Among the total 139 citations attached to the three indicated concept nodes, only 20 of them are duplicates.

Figure 3.2. Dynamic navigation steps to reach the concept "Histones" for the query "prothymosin".

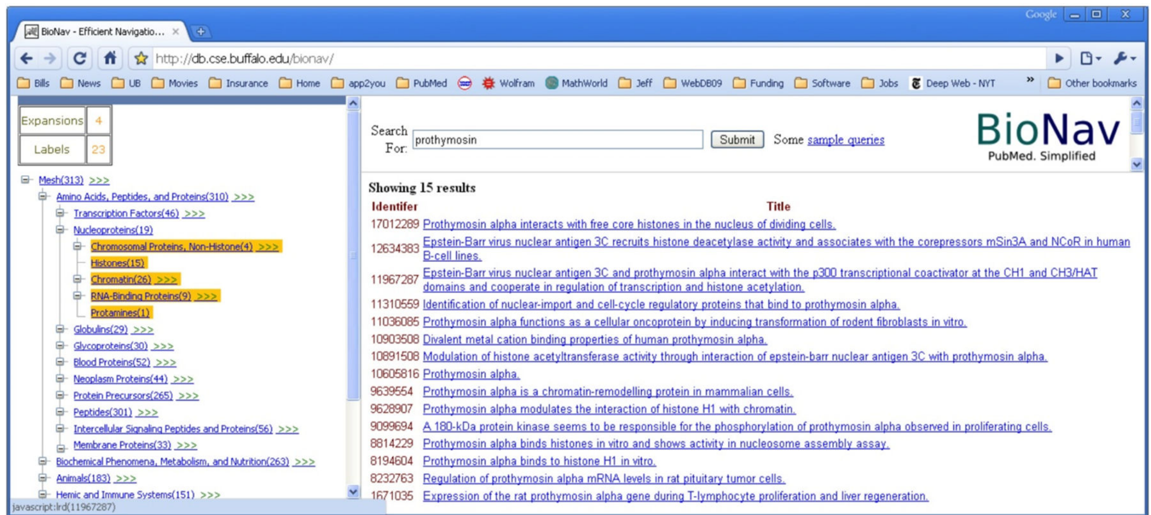


Figure 3.3. The BioNav Interface

Note that the user is not aware that the relevant results are available specifically on these nodes – she is only interested in narrowing down the results, using a familiar concept hierarchy, instead of examining all the results.

BioNav introduces a *dynamic* navigation method that depends on the particular query result at hand and is demonstrated in Figure 3.2. The query results are attached to the corresponding MeSH concept nodes as in Figure 3.1, but then the navigation proceeds differently. The key action on the interface is the *expansion* of a node that selectively reveals a ranked list of descendant (not necessarily children) concepts, instead of simply showing all its children.

Figure 3.2(a), for example, shows the initial expansion of the root node where only 8 (highlighted) descendants are revealed compared to 98 children shown in Figure 3.1. The concepts are ranked by their relevance to the user query and the number of them revealed depends on the characteristics of the query results. Next, assuming the user is interested in the "Amino Acids..." node and judging that the 310 attached citations is still a big number, she expands it by clicking on the ">>>" hyperlink next to it in Figure 3.2(b). The user inspects the 6 concepts revealed and

decides that she is not interested in any of them. Hence, she expands the “Amino Acids...” node one more time in Figure 3.2(c), revealing 4 additional concepts.

Note that “Nucleoproteins” is an example of a descendant node being revealed, since its parent node “Proteins” is not revealed in Figure 3.2(c). In Figure 3.2(d), the user expands the “Nucleoproteins” node and reveals “Histones”, one of the three key concepts for the query. In the last step of the interaction, the user clicks on the “Histones” hyperlink and the 15 corresponding citations are displayed in a separate frame as shown in Figure 3.3. To reach “Histones” using the BioNav navigation method only 23 concepts are revealed, after 4 node expansions, compared to 152 concepts, also after 4 expansions, with the static navigation method of Figure 3.1.

For each expansion, the displayed descendant concepts are chosen in a way that the expected navigation cost is minimized, based on an intuitive navigation cost model we present in Section 3.3. The cost model estimates the exploration probability for a node based on its selectivity, that is, the ratio of attached citations before and after the query. The navigation cost for a concept node is also proportional to the density of the navigation subtree rooted at this node in terms of citation count. Intuitively, the selection is done such that every expansion reduces maximally the expected remaining navigation cost. For example, the reason that “Proteins” is not displayed in Figure 3.2 is that it is too general given the query results and the original distribution of citations in the PubMed database (details in Sections 3.3 and 3.4), and hence displaying it would lead to an expected increase in the user navigation cost, based on the user navigation cost model.

In addition to the static hierarchy navigation works mentioned above, there are works on dynamic categorization of query results (e.g., the Clusty search engine [60], or [23, 56]), which create unsupervised query-dependent results clusters, but do not study how the clusters should be navigated. BioNav is distinct since it offers dynamic navigation on a predefined hierarchy, as is the

MeSH concept hierarchy. Another difference is that BioNav uses a navigation cost model to minimize the navigation cost.

Although we specifically target the biomedical domain in this work, the approach can be directly applied to datasets where tuples are classified using terms from a concept hierarchy. The BioNav system architecture and implementation is presented in Section 3.7.1. Related work is discussed in Section 3.8 and we summarize out findings in Section 3.9.

3.2 Framework and Overview

Definition 3.1 (Concept Hierarchy): A Concept Hierarchy $H(V, E, r)$ is a labeled tree consisting of a set V of concept nodes, a set E of edges and is rooted at node r. Each node $n \in V$ has a label l and a unique identifier id.

According to the semantics of the MeSH concept hierarchy [58], the label of a child concept node is more specific than the one of its parent. This also holds for most concept hierarchies.

Once the user issues a keyword query, PubMed –BioNav uses the Entrez Programming Utilities (eUtils) [61]– returns a list of citations, each associated with several MeSH concepts. BioNav constructs an *Initial Navigation Tree* by attaching to each concept node of the MeSH concept hierarchy a list of its associated citations. Formally, an Initial Navigation Tree $T_I(V_I, E_I, r)$ is a concept hierarchy, where every node (concept) $n \in V_1$ is additionally labelled with a *results (citations) list* $L(n)$.

In a given initial navigation tree, several concept nodes might have an empty results list. Since MeSH is a rather large concept hierarchy, BioNav reduces the size of the initial navigation

tree by removing the nodes with empty results lists, while preserving the ancestor/descendant relationships. Formally, the resulting structure is defined as follows.

Definition 3.2 (Navigation Tree): A Navigation Tree $T(V, E, r)$ is the maximum embedding of an initial navigation tree $T_I(V_I, E_I, r)$ such that no node $n \in V$ is labeled with an empty results list $L(n)$, excluding the root (in order to maintain the tree structure and avoid a forest).

An *embedding* $T(V, E, r)$ of a tree $T_I(V_I, E_I, r)$ is an injection from V to V_I such that every edge in E corresponds to a path (disjoint from all other such paths) in T_I. An embedding T of a tree T_I, where both trees are rooted at node r, is maximum if no other node n with a nonempty results list $L(n)$ can be added to V and T still be an embedding. The *maximum* embedding of the initial navigation tree is recursively computed in a single depth-first left-to-right traversal. If a non-leaf node n has an empty results list $L(n)$, then add all children of n to the parent of n and remove it. If n is a leaf, then remove it. Figure 3.4(a) shows part of the navigation tree for the "prothymosin" query, where the results lists are omitted for clarity.

The above procedure reduces the size of the initial navigation tree, but the structure is still too big (3,940 nodes for "prothymosin") to simply display it to the user and let her navigate it. BioNav minimizes her effort to reach the desired citations in the navigation tree by expanding in a way that minimizes the expected overall user navigation cost. Moreover, BioNav avoids information clutter by hiding unimportant concept nodes leading to interesting ones. This is achieved through a series of *expand* actions that reveal only a few descendants (not necessarily children) of the user selected node for further navigation.

We model a node expansion at a given navigation step as an *EdgeCut* in the navigation tree. In graph theory, an EdgeCut in a graph $G(V,E)$ is a set of edges $E_C \subseteq E$ such that the graph $G'(V,E\backslash E_C)$ has more components than G. For trees, any subset of the edges constitutes an EdgeCut, since the removal of any edge creates a new component.

In Figure 3.4(a), the dashed line illustrates the EdgeCut corresponding to the expansion of the node “Amino Acids...” and reveals the highlighted concepts of Figure 3.4(a). These revealed nodes are visualized on the interface as a tree shown in Figure 3.4(b). The EdgeCut consists of the edges (“Proteins”, “Transcription Factors”) and (“Proteins”, “Nucleoproteins”). Intuitively, an EdgeCut allows us to “skip” child nodes (“Proteins”), navigate directly to descendant nodes located deeper in the tree and show them as children of the node being expanded. Moreover, an EdgeCut can selectively reveal only a subset of a potentially large set of descendant nodes, as is the case in Figure 3.2(b) where only 6 out of the 52 descendants of “Amino Acids…” are revealed.

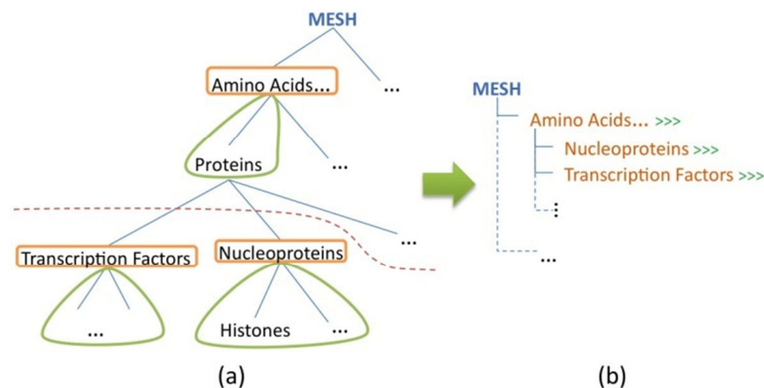


Figure 3.4. (a) Navigation Tree, EdgeCut and Component Subtrees, (b) Visualization of the EdgeCut on the user interface.

Definition 3.3 (Valid EdgeCut): A valid EdgeCut of a tree $T(V,E,r)$ is an EdgeCut $C \subseteq E$ such that no two edges in C appear in the same path from the root to some leaf node. We only consider valid EdgeCuts in the rest of the chapter, because invalid ones lead to unintuitive navigations.

Component Subtrees. An EdgeCut causes the creation of two types of *component subtrees*, *upper* and *lower*. Given an EdgeCut C of a tree $T(V, E, r)$, a *lower* component subtree $I(y_i)$ rooted at y_i is created by each node $y_i \in V$, such that $(x, y_i) \in C$ for some node x. In Figure 3.2(c), the expansion of "Amino Acids..." creates four lower component subtrees, two of which are shown in Figure 3.4(a), rooted at "Transcription Factors" and "Nucleoproteins". Moreover, for a given EdgeCut C, a single *upper* component subtree is created consisting of the nodes not in any *lower* component subtree, and is always rooted at the root of the tree being expanded. In Figure 3.4(a), the upper component subtree comprises of the nodes "Amino Acids..." and "Proteins".

The state of the navigation tree after an EdgeCut, and the component subtrees created, is captured by the *Active Tree*.

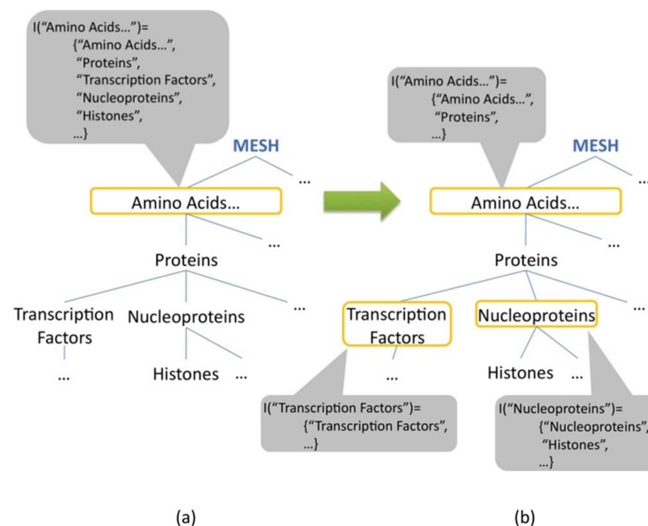


Figure 3.5. The Active Tree Before and After the EdgeCut in Figure 3.4.

Definition 3.4 (Active Tree): An Active Tree $T_A(V, E, r)$ is a Navigation Tree where each node $n \in V$ is annotated with a node set $I(n)$ consisting of the nodes in the component subtree rooted at n. If a node n is not a root of a component subtree, then $I(n) = \{n\}$. The non-singleton I sets are disjoint.

Before any EdgeCut, a navigation tree is trivially converted to an active tree by annotating the root node with an I set that includes all tree nodes. The rest of the nodes n_i are annotated with the node set $I(n_i) = \{n_i\}$. Figure 3.5(a) shows (part of) the active tree capturing the state of the navigation tree before the EdgeCut in Figure 3.4(a) (singleton I sets, such as "Histones", are not shown).

An EdgeCut (expansion) is an operation on the active tree, performed on the I set of a given node, and updates the sets $I(n_i)$ of the roots n_i of the upper and lower subtrees created by the EdgeCut based on the nodes included in these subtrees. It is denoted by EdgeCut: $I(n) \rightarrow S \subseteq I(n)$ and returns the set S of roots of the upper and lower subtrees that it creates. Figure 3.5(b) shows the effect of the EdgeCut operation in Figure 3.4(a) on the active tree in Figure 3.5(a). The active tree is closed under the EdgeCut operation.

Note that the set $I(n)$ of a node n is overloaded to also denote the "invisible" component subtree of the active tree that is rooted at n and only consists of the nodes in $I(n)$. For instance, the invisible subtree I("Amino Acids...") in Figure 3.5(b) is the one indicated as the upper component subtree in Figure 3.4a.

BioNav visualizes the active tree to the user by showing only the nodes that do not appear in any non-singleton I set organized as follows.

Definition 3.5 (Active Tree Visualization): The visualization of an active tree $\mathrm{T_A(V, E, r)}$ is the embedded tree $\mathrm{T_A{}'(V', E', r)}$. $\mathrm{V'}$ are the nodes not in any non-singleton $\mathrm{I(n)}$, for all $\mathrm{n \in V}$. Shown next to every node $\mathrm{n \in V'}$ is the number of distinct citations attached to nodes in $\mathrm{I(n)}$, given by $\mathrm{|L(I(n))| = \left|\bigcup_{n_i \in I(n)} L(n_i)\right|}$. If n has a non-singleton $\mathrm{I(n)}$, then an expand hyperlink is shown next to it.

The visualization of the active tree after the EdgeCut in Figure 3.4(a) is shown in Figure 3.4(b) and is a subset of the nodes revealed in Figure 3.2(c). The citation count $|L(I(n))|$ for "Nucleoproteins" in Figure 3.2(c) is 40 denoting the unique citations attached to it and its (invisible) component subtree. It is reduced to 19 in Figure 3.2(d), since its component subtree is getting smaller as descendant concept nodes are revealed.

EXPLORE($I(n)$)
  if n is the root
    S ←EXPAND $I(n)$ // that is S ←EdgeCut($I(n)$)
    For each n_i in S
      EXPLORE($I(n_i)$)
  else, if n is not a leaf-node, choose one of the following:
    1. SHOWRESULTS $I(n)$
    2. IGNORE $I(n)$
    3. S ←EXPAND $I(n)$
      For each n_i in S
        EXPLORE($I(n_i)$)
  else, choose one of the following: // n is a leaf node
    1. SHOWRESULTS $I(n)$
    2. IGNORE $I(n)$

Figure 3.6. TOPDOWN Navigation Model

An EdgeCut and the visualization of the resulting active tree are capable of reducing the navigation tree both height- and width-wise. The embedded tree in Figure 3.2(c), compared to the navigation tree in Figure 3.1, is narrower and shorter. Note that we do not make any assumptions about the user's preference over the tuples in the result and every citation in the result can be reached by a sequence of navigation actions, that is, there is no information loss in navigating the query results using our framework.

Using the ”>>>” hyperlinks, the user can trigger subsequent EdgeCut operations on component subtrees in a recursive fashion. Although we expect the user to trigger EdgeCut operations predominantly on the lower component subtrees, an EdgeCut is possible on the upper subtree as well. For example, an EdgeCut operation on the upper component subtree of Figure 3.4(a) would reveals the “Proteins” concept as parent of the previously revealed concept “Nucleoproteins”.

3.3 Navigation and Cost Model

The navigation model of BioNav is formally defined in this section. Then the navigation cost model is presented, which is used to devise and evaluate our algorithms.

Navigation Model. After the user issues a keyword query, BioNav initiates a navigation by constructing the initial active tree (which has a single component tree rooted at the MeSH root) and displaying its root to the user. Subsequently, the user navigates the tree by performing one of the following actions on a given component subtree $I(n)$ rooted at concept node n:

1. **EXPAND $\boldsymbol{I(n)}$**: The user clicks on the ″>>>″ hyperlink next to node n and causes an EdgeCut($I(n)$) operation to be performed on it, thus revealing a new set of concept nodes from the set $I(n)$.

2. **SHOWRESULTS $\boldsymbol{I(n)}$**: By performing this action, the user sees the results list $L(I(n))$ of citations attached to the component subtree $I(n)$.

3. **IGNORE $\boldsymbol{I(n)}$**: The user examines the label of concept node n, ignores it as unimportant and moves on to the next revealed concept.

4. **BACKTRACK**: The user decides to undo the last EdgeCut operation.

This navigation process continues until the user finds *all* the citations she is interested in. In order to define a cost model, we focus on a simplification of the general navigation model, which we call TOPDOWN, where only EXPAND, SHOWRESULTS and IGNORE are the available operations, that is, the user follows a top-down only navigation starting from the root. TOPDOWN is common in practice. When the user encounters a leaf node in TOPDOWN, the only option is SHOWRESULTS. The TOPDOWN navigation model is formally presented in Figure 3.6 and is a recursive procedure that is initially called on the root of the active tree.

TOPDOWN Cost Model. The cost model, which is inspired by a previous work [23], takes into consideration the number of concept nodes revealed by an EXPAND action, the number of EXPAND actions that the user performs and the number of citations displayed for a SHOWRESULTS action. In particular, the cost model assigns (i) cost of 1 to each newly revealed concept node that the user examines after an EXPAND action, (ii) cost of 1 to each EXPAND action the user executes, and (iii) cost of 1 to each citation displayed after a SHOWRESULTS action. For example, in the navigation of Figure 3.2 above, the cost for reaching the "Histones" concept and inspecting its attached citations is 42. That is, 4 EXPAND actions that reveal a total of 23 concept nodes, and a SHOWRESULTS action on the "Histones" concept that lists 15 citations. The user examines all concept nodes and all citations in order to select the ones of interest.

Since the exact sequence of actions of a user cannot be known *a priori*, we *estimate* the cost based on the following two probabilities:

- EXPLORE probability $P_E(I(n))$ is the probability that the user is interested in the component subtree $I(n)$ and will hence explore it. The IGNORE probability is $1 - P_E(I(n))$.

- EXPAND probability $P_C(I(n))$ is the probability that the user executes an EXPAND action on component subtree $I(n)$ given that she has chosen to explore $I(n)$. The SHOWRESULTS probability for $I(n)$ is $1 - P_C(I(n))$.

In Section 3.4, we show how we estimate probabilities $P_E(I(n))$ and $P_C(I(n))$. The cost of exploring component subtree $I(n)$, rooted at node n, is:

$$cost(I(n)) = P_E^N(I(n)) \cdot \left(\begin{array}{c} \left(1 - P_C(I(n))\right) \cdot \left|L(I(n))\right| \\ +P_C(I(n)) \cdot \left(B + |S| + \sum_{s \in S} cost(I_C(s)) \right) \end{array} \right)$$

where $P_E^N(I(n))$ is the normalized $P_E(I(n))$, such that the sum of P_E^N's of the component subtrees after an EdgeCut equals 1. P_E^N of the original tree is 1. The intuition for this normalization is that the probability that the user wants to explore a node n should not depend on the specific expansions sequence that revealed n.

The first operand of the addition inside the big parenthesis is the cost of executing SHOWRESULTS on n. The second operand is the cost of executing an EXPAND action on n. The constant B is the cost of executing the EXPAND action, and S is the set of concept nodes revealed by the action, or otherwise the roots of component subtrees returned by the EdgeCut operation. $I_C(s)$ is the updated I set of a node $s \in S$ after the EXPAND action on $I(n)$ has been performed.

Recall that $\left|L(I(n))\right|$ in the cost formula is the number of distinct citations attached to $I(n)$. Intuitively, creating a component subtree with large number of duplicates reduces the navigation cost if the SHOWRESULTS probability for that subtree is high. Moreover, the number of duplicates across component subtrees should be minimal; otherwise the user will pay the cost of inspecting a citation multiple times.

Finally, note that by changing B, the cost assigned to executing an EXPAND action we affect the number of revealed concepts after each EXPAND. In particular, increasing this cost leads to more concepts revealed for each EXPAND action. This cost can be thought of as a *cognitive measure* of a user's expectation of the system behavior as she navigates the query navigation tree. A small expand cost would decrease the number of concept nodes revealed during each EXPAND action, whereas the user can process more. It would also increase the number of EXPAND actions thus frustrating the user. In Section 3.7, we experiment with various values of B.

3.4 Estimation of Navigation Probabilities

We assume that each citation is *equally likely* to be of interest to the user. If more information about the "goodness" of the citations were available, our approach could be straightforwardly adapted using appropriate weighting for $L(I(n))$.

Estimating EXPLORE Probability P_E. Since all citations in the query result are assumed to be of equal importance, concept n is of higher interest if $L(n)$ is large. On the other hand, a concept that is associated with a very large number of citations $L_T(n)$ of MEDLINE, independently of the query, is probably not discriminatory or important. The latter is inspired by the inverse document frequency measure in Information Retrieval. Hence, $P_E(n)$ for a node n is proportional to $|L(n)|/|L_T(n)|$. We normalize $P_E(n)$ by dividing by the sum of all P_E's in the navigation tree T, that is:

$$P_E(n) = \left(\frac{\frac{|L(n)|}{|L_T(n)|}}{\left(\sum_{n_i \epsilon T} \frac{|L(n_i)|}{|L_T(n_i)|} \right)} \right)$$

For a component tree $I(n)$ rooted at node n:

$$P_E(I(n)) = \sum_{n_i \in I(n)} P_E(n_i)$$

Given the above formula, $P_E(I(n)) = 1$ for the initial active tree. The above P_E formulas, together with the cost model in Section 3.3, largely determine the characteristics of the component subtrees BioNav creates during an EXPAND action. In particular, the upper component subtree typically groups together (i) concepts with low P_E and a large number of attached citations, and (ii) concepts with high P_E and a small number of attached citations. The first group is dismissed as uninteresting and the second could lead to a large number of concepts being revealed. Intuitively, the two groups of concepts average each other out according to the $P_E(I(n))$ formula. The lower component subtrees typically group concepts with P_E and number of attached citations in-between the two extremes in a way that minimizes the average navigation cost.

Estimating EXPAND Probability P_C. $P_C(I(n))$ is 0, if n is a leaf concept node or has a singleton $I(n)$ set, since there is no other choice for the user. For internal nodes in the active tree with a non-singleton $I(n)$ set that have a large $L(I(n))$, a typical user will want to further narrow down when faced with the prospect of seeing too many citations, that is, $P_C(I(n))$ is 1, if $L(I(n))$ is greater than an *upper threshold*. $P_C(I(n))$ is 0, if $L(I(n))$ is smaller than an *lower threshold*. Currently, BioNav operates with 50 and 10 being the upper and lower threshold respectively.

In the remaining cases, a user might want to narrow down the search of $I(n)$ by executing an EXPAND action, if the citations under n are widely distributed among the subconcepts in $I(n)$. An objective measure for such a wide distribution (disorder) is *information entropy.* If the entropy of the subtree $I(n)$ is high, then the user would benefit from an EXPAND action. Hence, $P_C(I(n))$ is defined as:

$$P_c(I(n)) = E(I(n)) = \frac{-\sum_{n_i \in I(n)} \frac{|L(n_i)|}{|L(I(n))|} log \frac{|L(n_i)|}{|L(I(n))|}}{-log \frac{1}{|I(n)|}}$$

The sum can become greater than 1 because of the existence of duplicates. Hence, we normalize the entropy of $I(n)$ by dividing with the maximum entropy, where citations are uniformly distributed to all nodes in $I(n)$ and there are no duplicates.

P_C determines the impact of duplicates in a component subtree after a node expansion. If $P_C(I(n))$ is low, that is, the SHOWRESULTS probability is high, then the number of duplicates in $I(n)$ plays a bigger role in the way a component subtree is expanded.

3.5 Complexity Results

To prove that the problem of selecting the optimal valid EdgeCut for a given tree is NP-hard, where "optimal" means minimize the user navigation cost according to the navigation model of Section 3.3, we prove that the problem is NP-complete for a simplified navigation model, which we refer to as TOPDOWN-EXHAUSTIVE and is a special case of the TOPDOWN model shown in Figure 3.6.

In TOPDOWN-EXHAUSTIVE, BioNav performs an EXPAND action on the root of the initial active tree, and then the user selects randomly the root of one of the component subtrees created and performs a SHOWRESULTS action. The cost of TOPDOWN-EXHAUSTIVE navigation is the cost to read the root label of all component subtrees revealed by the EdgeCut plus the cost of SHOWRESULTS for the selected component subtree.

Intuition on the complexity of computing optimal valid EdgeCut: The "optimal" valid EdgeCut is the EdgeCut that will lead to the minimum expected navigation cost, that is, the

minimum average cost. In order to minimize the expected cost of TOPDOWN-EXHAUSTIVE navigation, we need to minimize the cost of EXPAND and of SHOWRESULTS. The cost of EXPAND is simply the number k of component subtrees produced by the EdgeCut. The average cost of SHOWRESULTS over all component subtrees equals the sum of unique elements (citations) in every subtree over k. This sum would be $|L(T)|$ where T is the navigation tree if there were no duplicates among the subtrees. However, due to the existence of duplicates (the same citation can be annotated with multiple MeSH concepts) this sum depends on the EdgeCut. Hence, the duplicates are the reason that the problem is NP-complete for TOPDOWN-EXHAUSTIVE, because we need to maximize the number of duplicates within the created subtrees, and at the same time create a relatively small number of component subtrees. Note that even for a given k, the problem of selecting the best EdgeCut is NP-hard as we show in Theorem 3.1.

Theorem 3.1. *Finding the optimal valid EdgeCut in TOPDOWN-EXHAUSTIVE is NP-complete.*

Proof. The decision problem corresponding to the problem of computing the optimal EdgeCut is as follows:

TOPDOWN-EXHAUSTIVE Decision (TED) Problem: Given a navigation tree T, where each node n contains a list $L(n)$ of elements from universe U (U are all the citations in the query result), that is, $L(n) \subseteq U$, there exists an EdgeCut C of T that creates k subtrees (including the upper subtree) with d duplicate elements within the created subtrees. That is, if $S_1, \dots, S_k$ are the subtrees and each S_i contains $b(S_i)$ duplicates, i.e., elements that appear somewhere in $S_1, \dots, S_{i-1}$ (if an element appears 3 times, then it counts as 2 duplicates), then $\sum_{i=1 \dots k} b(S_i) = d$.

Note that the cost of a TOPDOWN-EXHAUSTIVE navigation is computed as follows, if we solve the TED problem for every combination of k and d. If T has W unique results, then a subtree of the EdgeCut will have on average $(W + d)/k$ results. Hence the whole navigation cost is $k + (W + d)/k$, where k is the cost of reading the labels of the k subtrees.

TED is in NP since a solution can be verified in polynomial time. To prove that it is NP-complete, we will reduce the MAXIMUM EDGE SUBGRAPH (MES) problem, which is NP-complete [62], to TED.

MAXIMUM EDGE SUBGRAPH (MES) Problem: Given graph $G(V, E)$, a weight function $w: E \rightarrow N$ (N are the natural numbers) and positive integers d and k', there is a subset $V' \subseteq V$ with $|V'| = k'$ such that the sum of the edge weights of the edges between the nodes in V' is d, that is, $\sum_{(u,v) \in E \cap (V' \times V')} w(u, v) = d$.

Mapping of MES to TED: For each node $u \in V$, we create a node u' in T that is a child of the root of T. That is, the root r of T is empty ($L(r) = \emptyset$) and it has $|V|$ children.

The universe U is defined as follows: for each pair of edges $(u, v) \in E$ with weight $w(u, v)$, we add elements $B_{uv}^{1}, \ldots, B_{uv}^{w(u,v)}$ in U.

Each of the nodes of T is populated with elements from U as follows: For each edge $(u, v) \in E$, we add to nodes u' and v' of T the elements $B_{uv}^{1}, \ldots, B_{uv}^{w(u,v)}$. The intuition is that we map an edge weight in MES to the number of duplicates between two nodes in TED.We set $k = |V| - k' + 1$.

Note that the above reduction is linear on the maximum edge weight in G, which generally is less than $|V|$, hence the reduction is polynomial on $|V|$ and $|E|$. Now, a solution to MES is

mapped to a solution to TED, since selecting k' nodes in MES corresponds to expanding the tree into k subtrees in TED. The nodes of V corresponding to the nodes in the upper subtree of the EdgeCut (the one including the root) are the solution to MES. This set of nodes has maximum sum of edge weights in MES and maximum number of duplicates in TED.

3.6 Algorithms

Given the cost equation in Section 3.3, we can compute the optimal cost by recursively enumerating all possible sequences of valid EdgeCuts, starting from the root and reaching every concept in the navigation tree, computing the cost for each step and taking the minimum. However, this algorithm is also prohibitively expensive. Instead we propose an alternative algorithm *Opt-EdgeCut* that makes use of the *dynamic programming* technique to reduce the computation cost. As shown in Section 3.6.1 below, *Opt-EdgeCut* is still exponential and is just used to evaluate the quality of the heuristic we present in Section 3.6.2 (*Heuristic-ReducedOpt*). In Section 3.6.3, we consider an alternate navigation strategy (*TopKLevelWise*), which in several variations is used in existing systems, such as eBay and Amazon, and allows users to navigate query results using extensive concept hierarchies. In *TopKLevelWise*, a fixed-size subset of children is revealed during each EXPAND action on a concept node, where the subset is selected based on a fixed cost metric. We compare two variations of *TopKLevelWise* with *Heuristic-ReducedOpt* in Section 3.6.2 and show that the navigation cost incurred using our approach can be an order of magnitude lower than either of these approaches.

3.6.1 Optimal Algorithm for Best EdgeCut

The *Opt-EdgeCut* algorithm to compute the minimum expected navigation cost (and the EdgeCut that achieves it) traverses the navigation tree in post-order and computes the navigation cost bottom-up starting from the leaves. For each node n, the algorithm enumerates and stores the

list $\mathbb{C}(n)$ of all possible EdgeCuts for the subtree rooted at n, and the list $\mathbb{I}(n)$ of all possible $I(n)$ sets that node n can be annotated with. The algorithm then computes the minimum cost for each subtree in $\mathbb{I}(n)$ given the EdgeCuts in $\mathbb{C}(n)$ and the already computed minimum costs for the descendants of n. The complexity of *Opt-EdgeCut* is $O(|V| \cdot 2^{|E|})$.

Algorithm *Opt-EdgeCut*

Input: The navigation tree T

Output: The best EdgeCut

---

1 Traversing T in post-order, let n be the current node
2 **while** $n \neq root$ **do**
3   **if** n is a leaf node **then**
4     $mincost(n, \emptyset) \leftarrow P_E(n) * L(n)$
5     $optcut(n, \emptyset) \leftarrow (\emptyset)$
6   **else**
7     $\mathbb{C}(n) \leftarrow$ enumerate all possible EdgeCuts for the tree rooted at n
8     $\mathbb{I}(n) \leftarrow$ enumerate all possible subtrees for the tree rooted at n
9   **foreach** $I(n) \in \mathbb{I}(n)$ **do**
10   compute $P_E(I(n))$ and $P_c(I(n))$
11   **foreach** $C \in \mathbb{C}(n)$ **do**
12   **if** C is a valid EdgeCut for $I(n)$ **then**

$$cost(I(n), C) \leftarrow P_E\big(I(n)\big) \cdot \begin{pmatrix} \big(1 - P_C(I(n))\big) \cdot L\big(I(n)\big) \\ +P_C\big(I(n)\big) \cdot \big(B + |S| + \sum_{s \in S} mincost\big(I_C(s)\big)\big) \end{pmatrix}$$

13   **else**
14           $cost(I(n), C) = \infty$
15       $mincost(n, I(n)) \leftarrow \min_{C_i \in \mathbb{C}(n)} cost(I(n), C_i)$
16       $optcut\big(n, I(n)\big) \leftarrow C_i$
17 **return** $optcut(root, E)$ // E is the set of all tree edges

---

3.6.2 Heuristic-ReducedOpt Algorithm

The algorithm to compute the optimal navigation, *Opt-EdgeCut*, is exponential and hence infeasible for the navigation trees of most queries. We propose a heuristic to select a good EdgeCut for a node expansion. Note that the input argument to the heuristic is a component tree $I(n)$ and not the whole active tree T as in *Opt-EdgeCut*. The reason is that once *Opt-EdgeCut* is executed for T, the costs (and optimal EdgeCuts) for all possible $I(n)$'s are also computed and hence there is no need to call the algorithm again for subsequent expansions.

Algorithm *Heuristic-ReducedOpt*

Input: Component subtree $I(n)$, number z of partitions

Output: The best EdgeCut

---

1 $z' \leftarrow z$

2 **repeat**

3 $\quad k \leftarrow \sum_{n \in T} L(n) \cdot P_E(n)/z'$

4 $\quad$ *Partitions* $\leftarrow$ k-partition$(I(n), k)$ // call k-partition algorithm [63]

5 $\quad z' \leftarrow z' - 1$

6 **until** $|Partitions| \leq z$

7 construct reduced subtree $I'(n)$ from *Partitions*

8 *EdgeCut'* $\leftarrow$ *Opt-EdgeCut*$(I'(n))$

9 *EdgeCut* $\leftarrow$ corresponding of *EdgeCut'* for $I(n)$

10 **return** *EdgeCut*

For a given component subtree $I(n)$, *Opt-EdgeCut* enumerates a large number of EdgeCuts on $I(n)$ and repeats this recursively on its subtrees. We propose to run *Opt-EdgeCut* on a reduced version $I'(n)$ of $I(n)$. The reduced tree $I'(n)$ has to be small enough so that *Opt-EdgeCut* can run on it in "real-time". We select the size z of $I'(n)$ according to the processing power of our system. We set $z = 15$ in our experiments. Also, $I'(n)$ should approximate $I(n)$ as closely as possible.

$I'(n)$ is the tree of "supernodes" created by partitioning $I(n)$. Each supernode in $I'(n)$ corresponds to a partition of tree $I(n)$. Then, *Opt-EdgeCut* is executed on $I'(n)$.

The algorithm we use to partition the tree is based on the k-partition algorithm [63] that processes the tree in a bottom-up fashion. For each tree node n, the algorithm removes the "heaviest" children of n one-by-one until the weight of n falls below k. For each of the removed children, it creates a partition. The result is a tree-partitioning with the minimum cardinality. The complexity of the k-partition algorithm is $O(|V| \cdot \log|V|)$.

We adopt the k-partition algorithm to our needs as follows. For each node in $I(n)$, we assign weight equal to $|L(n)| \cdot P_E(n)$, which is an estimation of its navigation cost. We run the k-partition algorithm by setting k, the weight threshold, to $\sum_{n_i \in I(n)} L(n_i) \cdot P_E(n_i)/z$, where z is the number of desired partitions. However, this might result in more than z partitions, due to some non-full partitions. Therefore we repeatedly run k-partition algorithm on $I(n)$, gradually increasing k (by decreasing z) until up to z partitions are obtained. Note that z is the maximum tree size on which *Opt-EdgeCut* can operate in "real-time".

3.6.3 The TopKLevelWise Method

In *TopKLevelWise*, the navigation model has the following key difference to our expansion model: the component subtree generated by an EXPAND on a node n are all rooted at one of the children of n. The size of the EdgeCut is limited by a parameter K, and the component subtree are chosen using a simple cost metric – the number of distinct results in a given component subtree. We consider two variations of *TopKLevelWise*. The first, which we call *static*, is employed by GoPubMed [59] and Amazon and uses $K = \infty$, that is, it selects the entire set of children to be included in the EdgeCut. The second, *Top10LevelWise*, is used by e-commerce websites such as

eBay. Here, a set of $K = 10$ children, with the highest number of results, are displayed. We compare these two strategies to *Heuristic-ReducedOpt* in Section 3.7 and show that our approach outperforms both of them.

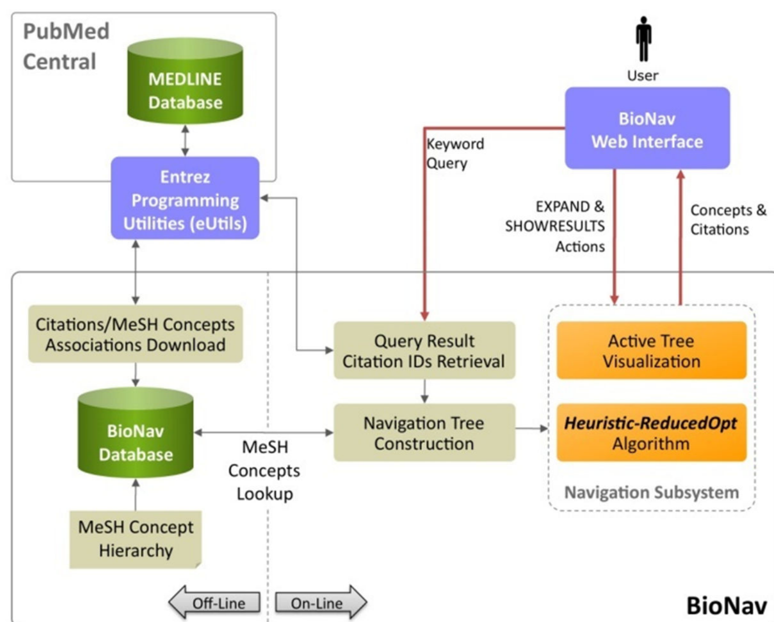


Figure 3.7. BioNav System Architecture.

3.7 Implementation and Experimental Evaluation

We evaluated the BioNav system in terms of both average navigation cost and expansion time performance. Other traditional measures of quality, such as precision and recall, are not applicable to our scenario as the objective is to minimize the tree navigation cost and not to classify.

In Section 3.7.1 we describe the system architecture and some implementation details. In Section 3.7.2, we show that the BioNav navigation method, which is evaluated using the *Heuristic-ReducedOpt* algorithm, leads to considerably smaller navigation cost for a set of real queries on the MEDLINE database and navigations on the MeSH hierarchy. In Section 3.7.3, we compare the

optimal algorithm (*Opt-EdgeCut*) with *Heuristic-ReducedOpt* and show that the heuristic is a good approximation of the optimal. These experiments were executed on a *reduced* navigation tree (~20 nodes), constructed from the original query navigation tree for each query, since *Opt-EdgeCut* is prohibitively expensive for most navigation trees. Finally, Section 3.7.4 shows that the execution time of *Heuristic-ReducedOpt* is small enough to facilitate interactive-time user navigation. The experiments were executed on a Dell Optiplex machine with 3Ghz CPU and 2 GB of main memory, running Windows XP Professional. All algorithms were implemented in Java and Oracle 10g was used as the database.

3.7.1 System Architecture and Implementation

The BioNav system architecture is shown in Figure 3.7 and consists of two parts. The off-line components populate the BioNav database with the MeSH concept hierarchy and the associations of the MEDLINE citations with MeSH concepts, while the on-line components support BioNav's web interface and the EXPAND-SHOWRESULTS actions of the user.

Off-Line Pre-Processing. The BioNav database is first populated with the MeSH hierarchy, which is available online [58] and has more than 48,000 concept nodes.

Then, the BioNav database is populated with the associations of the MEDLINE citations to MeSH concepts. These associations are not directly provided by the Entrez Programming PubMed so we had to implement the following method to infer these associations. For each concept in the MeSH hierarchy, we issued a query on PubMed using the concept as the keyword. For each citation ID in the query result, we added to a table in the BioNav database the tuple $<\text{concept}, \text{citationID}>$. Given the number of concepts in the MeSH hierarchy, the number of citations in MEDLINE (~18 million), and the PubMed eUtils restrictions on the number of queries that can be executed within a certain period of time, it took almost 20 days to collect all the

$< \text{concept}, \text{citationID} >$ tuples. In the end, there were almost 747 million such tuples. To improve the selection queries on this table, we de-normalized it by concatenating all concepts associated with each citation into a comma-separated list, that is:

$$< \text{citationID}, (\text{concept1}, \text{concept2}, \dots) >$$

In this work, we assume the dataset D to be fixed. However, in practice, D changes frequently as new citations are added and existing citations are updated to include new terms from the MeSH hierarchy. In this case, we assume that D is refreshed periodically by an offline process that issues queries to PubMed using the concept keyword and updates the concept counts and rows of retrieved citations. A newly added citation may not appear immediately in the query result, but we assume that such delays are acceptable to users. When executing queries using concepts as keywords, we also store the number of citations $L_T(n)$ in the query result needed for the computation of P_E in Section 3.4.

On-Line Operation. Upon receiving a keyword query from the user, BioNav executes the same query against the MEDLINE database and retrieves only the IDs (PubMed Identifiers) of the citations in the query result. This is done using the ESearch utility of the Entrez Programming Utilities (eUtils) [61]. eUtils are a collection of web interfaces to PubMed for issuing a query and downloading the results with various levels of detail and in a variety of formats. Next, the navigation tree is constructed by retrieving the MeSH concepts associated with each citation in the query result from the BioNav database. This is possible since MeSH concepts have tree identifiers encoding their location in the MeSH hierarchy, which are also retrieved from the BioNav database. This process is done once for each user query. The navigation tree is trivially converted to an active tree (see Section 3.2) and passed on the *Navigation Subsystem* that supports the user's actions on the BioNav web interface.

Initially, the navigation subsystem just visualizes the active tree on the web interface, that is, it simply shows its root node. Subsequently, the user requests an EXPAND action on the root. Then, the navigation subsystem executes the *Heuristic-ReducedOpt* algorithm on the tree $I(r)$ of the root r, and the resulting active tree is visualized on the web interface. When the user makes a SHOWRESULTS request, BioNav uses the Entrez ESummary utility to download high level information of the citations to be shown, such title and authors.

Table 3.1. Bionav Evaluation Query Workload.

#	Keyword(s)	# of Citations in Query Result	Navigation Tree Size	Max Tree Width/Height	Tree Citations w/ Duplicates	Target Concept	MeSH Level of Target Concept	|L(n)| of Target Concept	|LT(n)| of Target Concept
Biochemistry									
Q1	LbetaT2	116	1947	1009/10	14927	Mice, Transgenic	5	11	90804
Q2	melibiose permease	160	1324	722/8	14419	Substrate Specificity	3	31	79470
Q3	Na+/I symporter	163	2596	1367/6	17146	Perchloric Acid	3	7	4250
Q4	ibogaine	287	3020	1656/11	28148	Serotonin	5	43	101567
Q5	prothymosin	313	3941	2113/10	30897	Histones	4	15	22741
Q6	ice nucleation	474	3181	1776/9	27440	Plants, Genetically Modified	3	2	12330
Q7	dyslexia genetics	517	3056	1691/9	45079	Polymorphism, Single Nucleotide	4	18	18843
Q8	syntaxin 1A	1115	6589	3764/10	105503	GABA Plasma Membrane Transport Proteins	7	11	650
Q9	follistatin	1183	6446	3656/10	102946	Follicle Stimulating Hormone	6	157	34540
Q10	norepinephrine transporter	1681	6482	3816/11	124199	Protein Kinase C	7	18	46928
Medicine									
Q11	varenicline	162	1830	962/6	11370	Nicotinic Agonists	7	81	18277
Q12	vardenafil	486	3424	2014/8	40987	Phosphodiesterase Inhibitors	5	401	69984
Q13	duloxetine	695	3884	2323/10	57979	Fibromyalgia	3	28	4683
Q14	ebola virus	1062	5187	2992/11	83602	Ebola Vaccines	5	25	27
Q15	asperger's syndrome	1126	3884	2323/9	57979	Early Diagnosis	2	28	4683
Q16	nocturia	1297	4646	2660/11	77083	Nocturnal Enuresis	5	39	1397
Q17	oxaluria	1727	5097	2913/10	85536	Celiac Disease	4	2	12871
Q18	blepharosmasm	1329	5603	2145/9	72419	Blepharospasm	3	984	1313
Q19	cadmium poisoning	1882	6217	3628/11	79808	Infertility, Male	4	2	18839
Q20	tourette syndrome	3029	5196	1977/9	76835	Tourette Syndrome	5	36	2289

3.7.2 Navigation Cost Evaluation

To evaluate the navigation cost benefit of BioNav, we asked two researchers, who use PubMed regularly, to create a set of 10 queries each. The first researcher was a biochemist and the second a medical doctor. We asked them to consider queries that cover topics within their fields and are of exploratory nature, that is, queries that return more than just a few citations. For each query, we also asked them to designate a *target* MeSH concept in the corresponding navigation tree that they would subjectively consider as most interesting. The two sets of queries we received consist our workload and is show in Table 3.1. Apart from the queries ("Keywords" column), listed are statistics on the initial navigation trees, the target concepts and information regarding their

location depth in the MeSH hierarchy, the number of citations $|L(n)|$ attached to them for the given query, and the total number of citations $|L_T(n)|$ attached to them in MEDLINE.

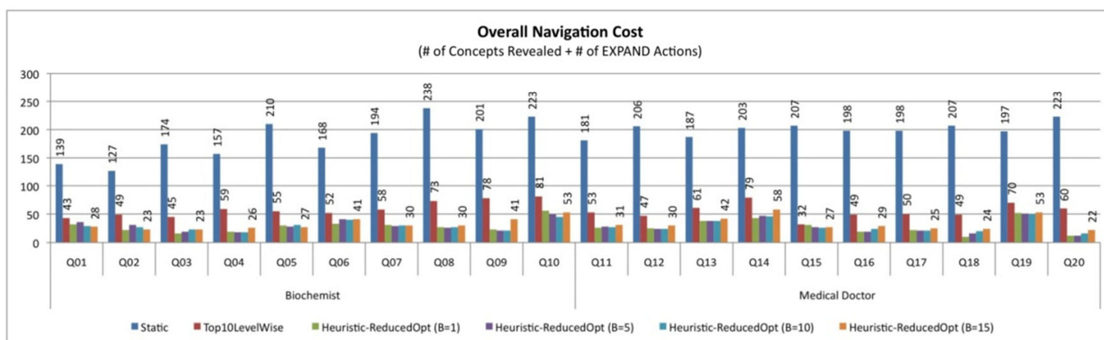


Figure 3.8. Overall Navigation Cost Comparison for Biochemistry and Medicine.

"Follistatin" and "LbetaT2" are terms that mainly interest biochemists studying reproductive endocrinology and gynecology. The "dyslexia genetics" query accumulates results related to genes associated with dyslexia. "Melibiose permease" and "Na+/I- symporter" are transport proteins related to bacterial growth and thyroid function respectively. On the other hand, "vardenafil" (Levitra), used for the treatment of erectile dysfunction, and "varenicline" (Chantix), used for quitting smoking, are two new drugs that interest many medical doctors.

Interestingly, some queries correlate with quite a few fields of research and others concentrate in more specific topics. For example, the literature for "prothymosin", although not particularly broad in number of citations in the query result (313), is associated with several topics such as cancer, cell proliferation, apoptosis, chromatin remodeling, transcriptional regulation and immunity. In contrast, "vardenafil" retrieves a higher number of citations (486) but the literature is mostly targeted to erectile dysfunction and hypertension. This fact is reflected on the navigation tree characteristics for the two queries, also shown in Table 3.1. The navigation tree for "prothymosin" is bigger than the one for "vardenafil" in every respect, that is, tree size, maximum width and height.

In this experiment we assume that the user follows a top-down navigation where she always chooses the right node to expand in order to finally reveal the target concept. We compare the navigation cost of BioNav, where EXPAND is implemented using the *Heuristic-ReducedOpt* algorithm (with $z = 10$), to the two navigation strategies, *Static* and *Top10LevelWise*, described in Section 3.6.3.

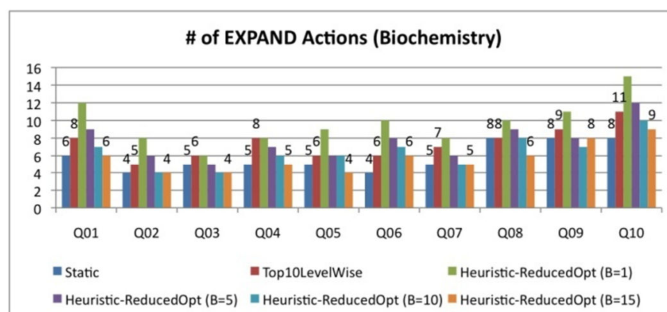


Figure 3.9. Number of Expand Actions Comparison.

Figure 3.8 compares the navigation cost for these three methods. We observe that BioNav often improves the navigation cost by an order of magnitude, over Static navigation. The average improvement of BioNav, over static navigation, is 82%, for $B = 15$. The improvement is high regardless of the navigation tree characteristics (87% for “prothymosin” (Q5), 85% for “vardenafil” (Q12)), and regardless of the number of citations in the query result (80% for “LbetaT2” (Q1), 90% for “tourette syndrome” (Q20)). The smallest improvement (71%) was observed for “ebola virus” (Q14). The reason is that its target concept (Ebola Vaccines) is located far away, in terms of navigation tree distance, from other query results. Most query results are distributed under a MeSH concept called “Viruses”, while the target concept is located under a sibling concept called “Complex Mixtures”. Hence, it takes several EXPAND actions until BioNav reveals the latter. Query “ice nucleation” (Q6) also exhibits small improvement (75%), but for a different reason. Its target concept (Plants, Genetically Modified) has an extremely low $|L(n)| = 2$. Hence, its P_E is quite low and so it takes several EXPAND actions until it is revealed.

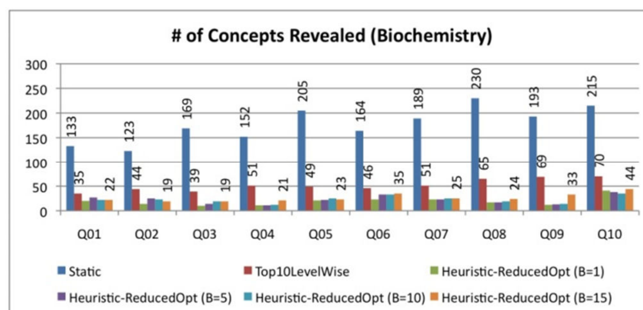


Figure 3.10. Number of Concepts Revealed Comparison.

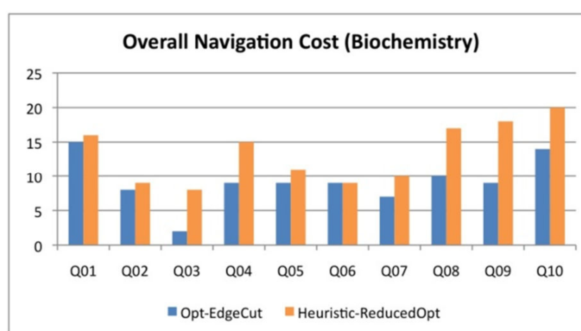


Figure 3.11. Overall Navigation Cost Comparison.

Consistent, but more modest, improvement in navigation cost is achieved by BioNav over *Top10LevelWise*. The average improvement is 41%, with a minimum of 16% for query "asperger's syndrome" (Q15) and a maximum of 63% for "tourette syndrome" (Q20). Since *Top10LevelWise* explores the navigation tree level-wise, a concept that is high up in the hierarchy, such as the target concept of "asperger's syndrome", can be reached as fast by *Top10LevelWise* as it does by BioNav. On the other hand, a concept that is deep inside the navigation tree but with high P_E, such as the target concept of "tourette syndrome", is reached much faster by BioNav.

Figure 3.9 shows the number of EXPAND actions for the three methods for the biochemistry query set only. Note that these numbers are relatively close, which means that the dramatic differences in Figure 3.8 are due to the fact that BioNav selectively reveals few descendant nodes for each EXPAND, instead of a possibly large number of child nodes. The worst case is the "ice nucleation" (Q6), where BioNav requires 6 EXPAND actions, compared to 4 of

static navigation, since the target concept is quite high in the MeSH hierarchy, and at the same time has a low P_E, as discussed above. A similar increase in the number of EXPAND actions is observed for query "ebola virus" (not shown in Figure 3.9) for the reason discussed above. Figure 3.10 shows the number of revealed concepts for each method and demonstrates the superiority of our approach.

Procedure *GenReducedTree*

Input: Initial Navigation Tree $I(n)$, the target concept c, and the desired number $maxN$ of nodes in the reduced tree

Output: A reduced tree with at most $maxN$ nodes, including c

---

1 collect all nodes of $I(n)$ in list L
2 create list L' to store the nodes of the reduced tree
3 add to L' a concept node in L with the same label as c and all its ancestors
4 **while** $(sizeof(L') \leq maxN)$ **repeat**
5   select a node c' uniformly at random from L
6   add c' and all its ancestors to L', excluding duplicates
7 create a tree $I'(n)$ from the nodes in L', preserving the parent-child relationship
8 return $I'(n)$

3.7.3 *Opt-EdgeCut* Comparison

To compare the optimal algorithm *Opt-EdgeCut* and *Heuristic-ReducedOpt*, we use the same query workload as in Section 3.7.1. As mentioned earlier, it is infeasible to execute *Opt-EdgeCut* on the navigation tree obtained for any query in Table 3.1. Therefore, we base our comparison on a *reduced* navigation tree $I'(n)$ obtained by applying the procedure *GenReducedTree* to an initial navigation tree $I(n)$. The procedure *GenReducedTree* ensures that a reduced navigation tree has (1) at least one concept node with the same label as the target concept

of the queries in Table 3.1, and (2) up to a maximum number $maxN$ of concept nodes. In this experiment, we set $maxN$ to 25.

Figure 3.11 compares the proportional navigation cost of *Opt-EdgeCut* over *Heuristic-ReducedOpt* for the biochemistry query set only. *Opt-EdgeCut* performs better than *Heuristic-ReducedOpt* for all queries. However, the improvement varies over a wide range (6% for “LbetaT2” (Q1), to 75% for “Na+/I symporter” (Q3)). This is because partitioning in *Heuristic-ReducedOpt* hides away the target nodes inside one of the partitions during an EXPAND action, effectively excluding their participation in an EdgeCut. Thus, more EXPAND actions are needed to reach the target concept, which increases the cost. The opposite is true for query “ice nucleation” (Q6). The target concept is relatively high up in the hierarchy and the partition algorithm creates a partition for the target concept during the first expansion. Thus the same number of expansions is needed to reach it, resulting in the same overall cost.

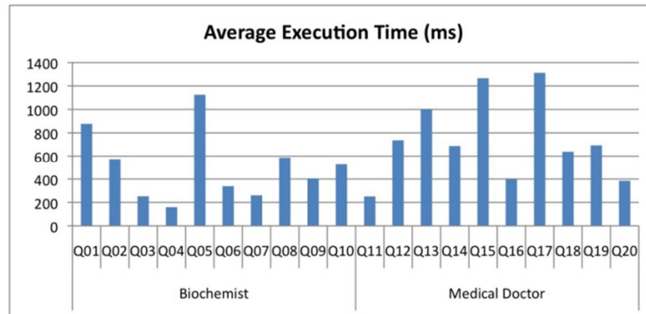


Figure 3.12. Heuristic-ReducedOpt EXPAND Performance.

3.7.4 Performance Evaluation

Figure 3.12 shows the average time of *Heuristic-ReducedOpt* to execute an EXPAND action for each query in Table 3.1. The average was taken over the number of EXPAND actions partially shown in Figure 3.9. For an input tree $I(n)$, *Heuristic-ReducedOpt* first creates a reduced tree $I'(n)$, and then runs the *Opt-EdgeCut* algorithm on it. The execution time is dominated by *Opt-*

EdgeCut as it is an exponential algorithm and depends on the size of the input tree. As stated earlier, we restrict the size of the reduced tree $I'(n)$ to 10 nodes and the EXPAND cost B is set to 15. However, $I'(n)$ can have a smaller size (see Section 3.6.2), in which case *Opt-EdgeCut* executes faster but with reduced accuracy.

For example, the reduced tree $I'(n)$ for "oxaluria" (Q17), in both EXPAND actions, had sizes 10 and 9 respectively, which explains the highest average execution time, and also among the highest improvements in Figure 3.8. On the other hand, for "Na+/I symporter" (Q3), the first three EXPAND actions resulted in an $I'(n)$ of sizes 8, 8 and 7, respectively. Hence, the average execution time in Figure 3.12 is lower, as is the improvement in navigation cost.

3.8 Related Work

Pubmed Search: Several systems have been developed to facilitate keyword search on PubMed using the MeSH concept hierarchy. Pubmed itself allows the user to search for citations based on MeSH annotations. This interface poses significant challenges, even to experienced users, since the annotation process is manual and thus prone to errors. The closest to BioNav is GoPubMed [31, 59], which implements a static navigation method on Pubmed results. GoPubMed lists a predefined list of high-level MeSH concepts, such as "Chemicals and Drugs" etc., and for each one of them displays the top-10 concepts. After a node expansion, its children are revealed and ranked by the number of their attached citations, whereas BioNav reveals a selective and dynamic list of descendant (not always children) nodes to the user's query.

Biomedical Search: Other systems that tackle PubMed search using the MeSH concept hierarchy include PubMed PubReMiner [64] and XplorMed [65, 66]. Both of them are query refinement tools and do not implement a particular navigation method. In particular, PubMed

PubReMiner outputs a long list of all MeSH concepts associated with each query along with their citation count. The user can select one or more of them and refine her query. XplorMed performs statistical analysis of the words in the abstracts of the citations in the query result and proposes query refinements/extensions to the user in a multi-step process. Ali Baba [67] displays the results on a graph where edges denote associations between the result nodes, which are typically genes and proteins. iHOP [68, 69] shows to the user the genes associated to a query gene, where the association is measured through co-occurrence in a sentence. LSLink [70] uses the physical links between objects in the query result to find *meaningful* associations between pairs of terms in different controlled vocabularies annotating objects in multiple datasources.

Categorization: Two academic proposals [23, 56] dynamically categorize SQL query results by inferring a hierarchy based on the characteristics of the result tuples. Their domain is the tuple attributes and their problem is how to organize them hierarchically in order to minimize the navigation cost. One of the systems [56] takes into consideration the user's preferences during the inference for a more personalized experience. Once the hierarchy is inferred, they follow static navigation. BioNav is distinct since it offers dynamic navigation on a predefined hierarchy. Hence, BioNav is complementary to these systems- it can be used to optimize the navigation, after these systems construct the initial navigation tree.

Clustering: Clustering systems [60, 71, 72] create unsupervised query-dependent clusters. PubMatrix[73] takes as input two sets of keywords terms, in addition to query keyword, and generates a co-occurrence frequency matrix of each pair of terms from the two lists, in the query result. The user can then browse this matrix and perform independent searches on pairs of terms. The Clusty [60] search engine clusters keyword-based query results on the web and operates on top of other search engines. HighWire Press [72] uses Clusty's algorithms to cluster query results in

the biomedical domain. [74] clusters PubMed documents by the drug they refer to based on the UMLS [75] drugs classification. Once the clusters are created, a static navigation method is followed. BioNav could be adapted to work on top of the (typically shallow) hierarchy created by clustering systems.

3.9 Summary

We address this problem of navigating results using the Mesh Concept Hierarchy by organizing the query results according to their associations to concepts and propose a dynamic navigation method on the resulting navigation tree. We formally stated the underlying framework and the navigation and cost models used for evaluation of our approach. We prove that the problem of selecting the set of nodes that minimize the navigation cost is NP-complete, we propose an efficient heuristic, and we validate it for diverse sets of queries and navigation trees.

Chapter 4

Cost-Driven Exploration of Faceted Results

4.1 Introduction

In a very common search scenario, when users are not familiar with the content or the structure of the underlying database, or they are not experienced with sophisticated search interfaces, they issue queries that are exploratory in nature and may return a large number of results. In other cases, users often issue broad (underspecified) queries in fear of missing potentially useful results. As a consequence, users end up spending considerable effort browsing long lists of query results. This phenomenon, known as *information overload*, is a major hurdle in querying large databases.

Information overload has been tackled from two directions – ranking and categorization. There are many recent works on ranking database results for both keyword [13, 40] and structured queries [20]. Ranking is effective when the assumptions used by the ranking function are aligned with user preferences. Ranking may not perform well for exploratory queries, since it is hard to judge which result is better than the other when the query is broad. Moreover, no summary (grouping) of the query result is provided for the user to refine her query. In categorization, query results are grouped based on hierarchies, keywords, tags, or attribute values. For instance, consider

the MEDLINE database of biomedical citations [8], whose articles are tagged with terms from the MeSH concept hierarchy [58]. Categorization systems propose a method for users to effectively explore the large results by navigating the MeSH sub-hierarchy relevant to the particular query result [27]. Wider adoption of such hierarchical categorization systems is limited, as building these concept hierarchies requires an intense manual effort, and automatically assigning terms to tuples afterwards is not always a successful process [76].

A popular variant of categorization, which is the focus of this chapter, is *faceted navigation* [33]. Here, the tuples in a query result are classified into multiple independent categories, or *facets*, instead of a single concept hierarchy. For an example car dataset, the result for keyword query "$honda$" shown in Figure 4.1(a) is categorized based on $Year$, $City$ and $State$ facets, among others. Each facet is associated with a set of *facet conditions*, each of which appears in the number of tuples shown in parenthesis (cardinality). For instance, the $Year$ facet in Figure 4.1(a) is associated with the set $(2000, 2001, \dots)$ of facet conditions. The user can narrow down or *refine* this result set by selecting a facet condition (e.g., $Year = 2003$) and clicking on it. User studies have shown that faceted navigation improves the ability of users to explore large query results and identify tuples of interest when compared to single concept hierarchies [77].

Faceted navigation has been studied extensively by the Information Retrieval community, where the challenge is to dynamically determine the facets for a given set of documents. The drawback of these systems is the unpredictability and counter-intuitiveness of the resulting facets [76, 78]. In contrast, faceted navigation is much more intuitive and predictable for structured databases, where each attribute is a facet describing a particular characteristic of the tuples in the dataset.

(a) Initial Query Results and Suggested Facet Conditions (b) REFINE Action (c) EXPAND Action

Figure 4.1. The FACeTOR Interface.

The following are key concerns that need to be addressed to achieve effective faceted navigation when the number of facets and facet conditions are large:

1. Which facets and facet conditions should be suggested (displayed) to the user? For example, the query result in Figure 4.1(a) consists of 789 tuples that can be categorized using 41 facets and 234 facet conditions. Suggesting "familiar" facets and facet conditions would help the user make a refinement decision without requesting additional facet conditions (by clicking the "More" hyperlinks in Figure 4.1(a)). For example, if users are more familiar with the $Year$ facet than the $Mileage$ facet in Figure 4.1(a), it is intuitive to suggest conditions from the *Year* facet. Most current solutions, try to address the facet conditions selection problem in an *ad hoc* manner by ranking the facet conditions using results cardinality or other ad-hoc factors.

2. Which facet conditions will lead to the tuples of interest in fewer navigation steps? For example, although the facet condition $Make = Honda$ has the highest cardinality for the query result in Figure 4.1(a) (the approach followed by most current systems), selecting

this facet will not refine the query result by a large margin, thereby forcing the user to perform additional refinements to narrow down the query result.

3. The overlap of the query results among the set of suggested conditions is another critical concern, since a low overlap can reduce the suggestions inspected and shorten the navigation. In Figure 4.1(a), if most of the $City = Dallas$ cars were made in $Year = 2001$, it is not wise to suggest both facet conditions. If the user chooses one of them in one navigation step, she would have to inspect the other in the next step anyway.

In this chapter, we present the FACeTOR system that takes a cost-based approach to selecting the set of facet conditions to suggest to the user at each navigation step. These facet conditions are selected using an intuitive cost model that captures the expected cost of navigating a query result. At each navigation step, FACeTOR first computes the applicable facet conditions. However, instead of showing all of them or ranking them by an *ad hoc* function, FACeTOR suggests a subset of them based on an intuitive navigation cost model, which considers factors including the user's familiarity with the suggested conditions, their overlap, and the expected number of navigation steps. The suggested facet conditions are chosen such that they minimize the expected navigation cost until the tuples of interest are reached, although these are not known *a priori*.

Recent works on faceted navigation of database query results [23, 33] have limitations that we address in this chapter. In both works, the navigation algorithm selects one facet (or possibly multiple ones [33]) and displays *all* its facet conditions to the user. Instead, we suggest a mix of facet conditions from several facets, that is, our algorithm operates at the facet condition level and not the facet level. Further, our cost model more closely estimates the actual user navigation cost.

These improvements introduce novel algorithmic challenges, due to the explosion of the search space and the interactive time requirement of exploration systems.

Table 4.1. Symbol Reference.

Symbol	Meaning
S_R	The schema of the initial result set with attributes $A_1, \ldots, A_m$
R	The initial result set
Q	The query formulated during a faceted navigation
R_Q	$R_Q \subseteq R$, the result of a query Q over R
c	A facet condition of the form $A_i = a_j$
$C(R_Q)$	All possible facet conditions for R_Q
$C(A_i)$	All possible facet conditions for attribute A_i
$C_S(R_Q)$	$C_S(R_Q) \subseteq C(R_Q)$, the suggested conditions given R_Q

4.2 Framework and Definitions

The starting point of the FACeTOR framework is a *result set* that the user explores.

Definition 4.1 (Result Set): A result set is a relation R with schema $S_R = (A_1, \ldots, A_m)$. Each attribute $A_i \in S_R$ has an associated active domain $ADom(A_i, R)$ of un-interpreted constants.

The initial result set R could be the whole database or more realistically, the result of a keyword query. In this work, we assume that the user first submits a keyword query (e.g., "$honda$" in Figure 4.1(a)). At each step of a faceted navigation, FACeTOR classifies the tuples of a result set R according to their *facets*. Each attribute $A_i \in S_R$ of R contributes a facet to the classification which in turn, contributes a set of conditions.

Definition 4.2 (Facet Condition): Given a result set R, a facet condition is an equality predicate $c: A_i = a_i$, where $A_i \in S_R$ and $a_i \in ADom(A_i, R)$. The set of all possible facet conditions for a result set R is $C(R)$.

Our running example considers a cars result set R whose tuples are classified by their $Year$, $City$, $Exterior\ Color$ and 37 more facets. As shown in Figure 4.1(a), FACeTOR displays the name of each facet along with a list of facet conditions as hyperlinks, followed in parenthesis by the number of tuples in R satisfying the condition (cardinality).

When the user clicks on a hyperlink corresponding to a facet condition c_i, FACeTOR filters the result set R to the tuples that satisfy c_i, thus yielding a new result set $R_Q \subseteq R$, and the faceted navigation proceeds to the next step where R_Q is now being classified. FACeTOR captures the progression of the faceted navigation using a query Q. When the user clicks on a facet condition c_i, then the equality predicate is added conjunctively to Q, thus forming a refined query $Q \wedge c_i$. At each navigation step, FACeTOR *suggests* only a subset $C_S(R_Q)$ of all possible facet conditions in $C(R_Q)$.

```
NAVIGATE(Q)
1  Choose one of the following:
2     SHOWRESULT(R_Q)
3     Examine all suggested conditions C_S(R_Q)
4        Choose one of the following:
5           REFINE(Q, c)
6              Q = Q ∧ c
7           EXPAND(A_i, R_Q)
8              Examine all remaining conditions in C(A_i)\C_S(R_Q)
9              Choose a condition c' ∈ (C(A_i)\C_S(R_Q))
10             Q ← Q ∧ c'
11       NAVIGATE(Q)
```

Figure 4.2. Faceted Navigation Model

Definition 4.3 (Suggested Conditions): For a result set R_Q, a set of facet conditions $C_S(R_Q) \subseteq C(R_Q)$, are suggested if $\bigcup_{c \in C_S(R_Q)}(R_{Q \wedge c}) = R_Q$, that is, every tuple in R_Q satisfies at least one suggested condition.

In this work, we are interested in minimizing the overall expected navigation cost incurred by the user, by choosing the *best* set of suggested conditions for a given R_Q, without making any assumptions about the user's preference over the tuples in R_Q. The navigation cost is based on an intuitive model of user navigation.

4.3 Navigation and Cost Model

The faceted navigation model is formally presented in Section 4.3.1 and forms the basis for the navigation cost model defined in Section 4.3.2.

4.3.1 Faceted Navigation Model

At each faceted navigation step, FACeTOR displays to the user the set of suggested conditions $C_S(R_Q)$ for the current result set R_Q. The user then explores R_Q by examining all conditions in $C_S(R_Q)$ and proceeds to the next navigation step by performing one of the following actions:

1. **SHOWRESULT($\boldsymbol{R_Q}$):** The user examines all tuples in the result set R_Q. If, in Figure 4.1(a), the user chooses to stop navigation and read all the results, she would have to read a total of 789 result tuples and 21 labels.
2. **REFINE($\boldsymbol{Q, c}$):** The user chooses a suggested condition $c \in C_S(R_Q)$ and refines query Q, that is, Q becomes $Q \wedge c$. The result of $REFINE(Q, c{:}\, year = 2003)$ is shown in Figure 4.1(b). As

a consequence of this action, the result set has now been narrowed down to 200 tuples and the new set of suggested conditions is available for this *refined* result set.

3. **EXPAND**$(\boldsymbol{A_i}, \boldsymbol{R_Q})$: The user is dissatisfied with (rejects) *all* suggested conditions in $C_S(R_Q)$. Instead, she EXPANDs an attribute A_i, by clicking on its "More" hyperlink, which reveals the remaining facet conditions for A_i in R_Q, and selects one of them to REFINE the query Q. This occurs when the user is not familiar with any of the suggested conditions. The effect of EXPAND is shown in Figure 4.1(c), where the remaining facet conditions for $State$ are revealed.

The formal navigation model is presented in Figure 4.2. It is a recursive procedure and is initially called on the entire result set R and the identity query Q, and terminates when the user finds all the tuples of interest, i.e. when the user executes SHOWRESULT(R_Q). The set R_Q and the suggested conditions $C_S(R_Q)$ is computed at the beginning of each NAVIGATE step.

4.3.2 Faceted Cost Model

The cost model measures the navigation cost incurred by the user when exploring a query result set R_Q, using the navigation model described in Section 4.3.1. The navigation cost is the sum of costs of the actions performed by the user, which is, examining suggested conditions, SHOWRESULT, REFINE and EXPAND actions.

The cost of examining all tuples in a result set R_Q, that is, the cost of SHOWRESULT(R_Q) is $|R_Q|$, and the cost of examining all suggested conditions is $|C_S(R_Q)|$.We assume that the REFINE and the EXPAND actions have a cost B associated with them, that is, B is the cost of "clicking" on a suggested condition or executing an EXPAND action on the attribute $A_i \in S_R$.

If the exact sequence of actions followed by the user in navigating R_Q were known *a priori*, we could accurately determine the cost of navigation. Since this sequence cannot be known in advance, we estimate the navigation cost, taking into account the inherent uncertainty in the user navigation. To estimate the navigation cost, we introduce four probabilities:

- **SHOWRESULT Probability $\boldsymbol{P_{SR}(R_Q)}$** is the probability the user examines all tuples in the result set R_Q and thus terminates the navigation. If no facet conditions can be suggested, then $P_{SR}(R_Q) = 1$.

- **REFINE Probability $\boldsymbol{P(c)}$** is the probability the user refines the query Q by a suggested condition $c \in C_S(R_Q)$.

- **Attribute Preference Probability $\boldsymbol{P_A(A_i)}$** is the probability the user prefers suggestions from attribute A_i.

- **EXPAND Probability $\boldsymbol{P_E(R_Q)}$** is the probability the user does not choose a suggested condition and instead performs an EXPAND action is $P_E(R_Q) = \prod_{c \in C_S(R_Q)} (1 - P(c))$.

Since the navigation model is recursive, the expected navigation cost can be estimated by the following recursive cost formula:

$$
\begin{aligned}
&cost(Q) \\
&= P_{SR}(R_Q) \cdot |R_Q| + \left(1 - P_{SR}(R_Q)\right) \\
&\cdot \left[\begin{array}{c} B + \left|C_S(R_Q)\right| + (1 - P_E(R_Q)) \cdot refine\left(Q, C_S(R_Q)\right) + \\ P_E(R_Q) \cdot \displaystyle\sum_{A_i \in S_R} P_A(A_i) \cdot \left(\left|C(A_i) \backslash C_S(R_Q)\right| + refine\left(Q, C(A_i) \backslash C_S(R_Q)\right)\right) \end{array}\right]
\end{aligned} \quad (4.1)
$$

where $refine(Q, C) = \sum_{c \in C} \left(P_{norm}(c) \cdot cost(Q \wedge c)\right)$ (4.2)

The first line of Equation 4.1 captures the fact that the user has two options, when presented with a set of suggested conditions. One is to execute a SHOWRESULT action with probability $P_{SR}(R_Q)$ and cost $|R_Q|$. The other is to execute a REFINE or EXPAND action with probability $1 - P_{SR}(R_Q)$. The cost consists of the following parts shown in the square brackets of cost formula:

1. A fixed cost B of a REFINE action.

2. The user reads the suggested conditions with cost $|C_S(R_Q)|$.

3. With probability $1 - P_E(R_Q)$ the user decides to REFINE. The cost of REFINE, shown in Equation 4.2, is the sum of all possible REFINE choices weighted by their probabilities. These probabilities are normalized to sum to 1, as follows:

$$P_{norm}(c) = P(c)/\Sigma_{c \in C} P(c)$$

4. With probability $P_E(R_Q)$, the user does not choose any of the suggested conditions and performs an EXPAND action instead (third line of Equation 4.1). With probability $P_A(A_i)$, the user prefers attribute A_i over all other attributes and EXPANDs it. She examines all the non-suggested conditions for A_i, $|C(A_i) \backslash C_S(R_Q)|$ in total, chooses one of them and refines query Q. The estimated cost for the last step is also given by the refine formula in Equation 3.2 above, where $C = C(A_i) \backslash C_S(R_Q)$.

The cost formula (Equation 4.1) quantizes the effort incurred by the user navigating the results R_Q of the query Q. The challenge now is to choose the set of conditions $C_S(R_Q) \subseteq C(R_Q)$, that minimizes the overall navigation cost.

4.4 Estimating Probabilities

Our aim here is to present a framework for effort based navigation of faceted query results. The problem of estimating probabilities, $P_{SR}(R_Q), P(c), P_A(A_i)$ and $P_E(R_Q)$, is orthogonal to the solution and can be estimated in various ways viz. information theoretic approaches such as entropy, user navigation logs etc. However, for the sake of completion and evaluation of the framework, we present a method to estimate these probabilities.

Estimating $\boldsymbol{P_{SR}(R_Q)}$, the probability the user executes SHOWRESULT on a given result set R_Q. We use the information theoretic measure of *Entropy* to estimate P_{SR}. The rationale behind this decision is that the user would choose to further refine the query Q and narrow down the result set R_Q if the tuples in R_Q are widely distributed among all possible facet conditions $C(R_Q)$. The entropy of a result set R_Q distributed amongst the facet conditions in $C(R_Q)$ is given by:

$$H\left(R_Q, C(R_Q)\right) = -\sum_{c \in C(R_Q)} \left(|R_{Q\wedge c}|/N\right) ln\left(|R_{Q\wedge c}|/N\right)$$

where $N = \sum_{c \in C(R_Q)} |R_{Q\wedge c}|$ is the sum of the number of tuples over all facet conditions.

Since the value of entropy can be greater than 1, we normalize it with the maximum value of entropy for a given result set R_Q distributed over $|C(R_Q)|$ facet conditions. Entropy is maximal when N tuples are distributed equally amongst $|C(R_Q)|$ facet conditions, that is, each facet condition is satisfied by $N/|C(R_Q)|$ tuples. The total entropy of such a system is:

$$H_{max}\left(R_Q, C(R_Q)\right) = -\sum_{c \in C(R_Q)} \left(\frac{N/|C(R_Q)|}{N} ln \frac{N/|C(R_Q)|}{N}\right) = ln|C(R_Q)|$$

Hence,

$$P_{SR}(R_Q) = \frac{-\sum_{c \in C(R_Q)} |R_{Q \wedge c}|/N \, ln \, |R_{Q \wedge c}|/N}{ln|C(R_Q)|}$$

Estimating $P_A(A_i)$: This is the probability the user knows or likes attribute A_i. We estimated this probability using a survey of 10 users (students and faculty in our institutions) who rated each attribute A_i in the dataset on a scale from 0 to 1. These values are taken to be the user preference $P_A(A_i)$ for attribute A_i.

Estimating $P(c)$: $P(c)$ is the probability the user executes a REFINE action on suggested condition c. A user would REFINE by c, if she knows or likes the attribute of c and is also familiar with the value of the attribute in c. Therefore, we used a two-pronged approach to compute $P(c)$. To estimate the popularity of a value of a facet condition, we computed the frequency $freq(A_i.v_i)$ of each value for each attribute in R. Then, we multiply each frequency with the attribute preference to obtain the attribute/value preferences $P(c{:}A_i = v_i) = freq(A_i.v_i) \cdot P_A(A_i)$, which we then normalize by dividing by the maximum frequency for each attribute.

4.5 Algorithms

Given the intractability of the *Facet Selection* problem (Section 4.5.1), we have to rely on heuristics to compute the set of suggested conditions. To develop these heuristics, we analyzed the cost model presented in Section 4.3.2 to determine the characteristics of suggestions that form good candidates for suggested conditions.

Next, we present two heuristics to efficiently compute the best set of suggested conditions. The first, ApproximateSetCover (Section 4.5.3), is inspired by an approximation algorithm for the weighted set cover problem [79], and attempts to find a relatively small set of suggestions that have

a high probability of being recognized by users (high P(c)). The second heuristic, UniformSuggestions (Section 4.5.4), follows Equation 4.1 more closely and greedily selects facet conditions based on a heuristic assumption that is derived from the analysis of the cost model.

4.5.1 Complexity Results

We prove that the problem of finding the suggested facet conditions that minimize the expected navigation cost given by Equation 4.1 is NP-Hard, by showing that a simplified version of the problem is also NP-Hard. The *Simplified Facet Selection (SFS)* problem considers a simpler navigation model than the one in Section 4.3.1, called NAVIGATE-SINGLE and defined next.

NAVIGATE-SINGLE: In this model, the system performs a single REFINE action, where the user randomly selects one of the suggested conditions, and then performs a SHOWRESULT action. The cost of NAVIGATE-SINGLE navigation is the cost to examine all suggested conditions displayed ($|C_S(R_Q)|$) plus the cost $|R_{Q \wedge c}|$ of performing the SHOWRESULT action for the randomly-selected suggested condition c.

Suppose that the dominant cost of our cost model is that of examining a suggested condition. That is, suppose the cost to examine a suggested condition is 1 and the cost of SHOWRESULT is 0. Also suppose that all attributes of R_Q are Boolean (0, 1) and that the suggested conditions in $C_S(R_Q)$ are always positive, that is, $|C(A_i)| = 1$. Recall that facet conditions only specify a single attribute.

Theorem 4.1: *The SFS problem is NP-Hard.*

Proof (sketch): SFS is clearly in NP. To prove the NP-Hardness we reduce from the HITTING-SET problem. An instance of the HITTING-SET problem consists of:

- a hypergraph $H = (X, E)$, where X is a finite set of vertices and $E = (E_1, \dots, E_n)$ is a set of hyperedges, that is, subsets of X, and

- a positive integer $l \leq |X|$.

The problem is to determine whether there is a hitting set $H \subseteq X$ of size l such that $\forall i \in \{1, \dots n\}: H \cap E_i \neq \emptyset$.

We reduce HITTING-SET to SFS as follows. A node u_i in X becomes a facet condition $A_i = 1$. A hyperedge $E_i \in E$ becomes a tuple t_i in the result set R_Q. E_i connects the vertices corresponding to the attributes that have value 1 for the result t_i. The solution of HITTING-SET translates naturally to a solution to NAVIGATE-SINGLE and vice versa.

4.5.2 Cost Model Analysis

Consider a sample result set R_Q shown in Figure 4.3. Also shown, are three *alternative* sets of suggested conditions (Figure 4.3(a), 4.3(b) and 4.3(c)) selected from the set of all facet conditions $C(R_Q)$. Which one of the alternative set of suggestions shown in Figures 4.3(a), 4.3(b) and 4.3(c) has the lowest cost, and therefore is more likely to be selected by the navigation cost model?

The suggested conditions shown in Figure 4.3(a) are highly selective, since each one of them appears in a small number of results (low cardinality). Therefore, a large number of such conditions are required to *cover* the result set R_Q causing the navigation cost to increase as the user now has to read all the labels before proceeding to the next navigation step.

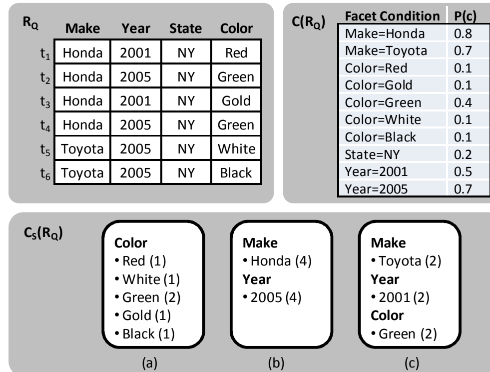


Figure 4.3. Result Set R_Q, All Facet Conditions $C(R_Q)$, and Three Alternative Sets of Suggested Conditions $C_S(R_Q)$.

A set of suggested conditions where each condition has low selectivity (Figure 4.3(b)) also leads to a high overall expected navigation cost. Such conditions typically have a high overlap and do not effectively narrow down the result set and therefore, the user has to execute more REFINE actions to narrow down the result set. For example, refining by either $Make = Honda$ or $Year = 2005$, in Figure 4.3(b), reduces the number of results from the initial six to four, and the resulting result set may need to be refined further before reaching the desired result(s). Conditions with low selectivity can potentially lead to redundant navigation steps. For example, refining by $State = NY$ does not narrow down the result but still adds to the navigation cost.

Based on the above discussion, we observe that the facet conditions selected by the cost model as suggested ones should neither have high nor low selectivity. The suggested conditions in Figure 4.3(c) are facet conditions with such desired characteristics. The conditions $Make = Toyota$, $Year = 2001$ and $Color = Green$ are moderately selective and thus have minimum overlap and do not require a large number of conditions to cover R_Q.

Another factor that increases the navigation cost is the EXPAND action, since the user can potentially see a large number of conditions, thereby increasing the navigation cost. The expected

cost of EXPAND is multiplied by $\prod_{c \in C_S(R_Q)}(1 - P(c))$, which is minimized when all the conditions in $C_S(R_Q)$ have a high $P(c)$.

Algorithm: ApproximateSetCover(Q, R_Q)

Input: A query Q, a result set R_Q

Output: The suggested conditions $C_S(R_Q) \subseteq C(R_Q)$

---

1 $C_S(R_Q) \leftarrow \emptyset$
2 $V \leftarrow \emptyset$ // V are results covered so far
3 while $V \neq R_Q$ // while not all results covered
4 $\quad c \leftarrow argmax_{c \in C(R_Q)}\left(P(c) \cdot |R_{Q \wedge c} \backslash V|\right)$
5 $\quad C_S(R_Q) \leftarrow C_S(R_Q) \cup \{c\}$
6 $\quad V \leftarrow V \cup R_{Q \wedge c}$
7 $\quad Q \leftarrow Q \wedge c$
8 return $C_S(R_Q)$

Figure 4.4. ApproximateSetCover Heuristic.

4.5.3 ApproximateSetCover Heuristic

Given a result set R_Q and its facet conditions $C(R_Q)$, the objective is to compute the set of suggested conditions $C_S(R_Q)$ such that the expected navigation cost, based on our cost model, is minimal and the set $C_S(R_Q)$ *covers* R_Q, that is, $\bigcup_{c \in C_S(R_Q)} R_{Q \wedge c} = R_Q$, where each facet condition c covers $|R_{Q \wedge c}|$ results in R_Q.This problem closely resembles the well-known NP-hard weighted set cover problem – given a set system (U, S), such that $\bigcup_{s \in S} s = U$, and weights $w: S \rightarrow \mathbb{R}_+$, find a subfamily $\mathcal{F} \subseteq S$ such that $\bigcup_{s \in \mathcal{F}} s = U$ and $\sum_{s \in \mathcal{F}} w(s)$ is minimal. The approximation algorithm for weighted set cover [79] adds at every step the set s that maximizes the number of newly covered items divided by the weight $w(s)$. In order to apply the approximation algorithm for weighted set cover to our problem, we need to define the weight function $w: C(R_Q) \rightarrow \mathbb{R}_+$. By

observing the cost formula in Equation 4.1, each facet condition in the suggested set $C_S(R_Q)$ should have a high probability $P(c)$ of being selected for REFINEment. Otherwise, the probability that the user does not select a suggested condition and chooses EXPAND would be high, resulting in a high overall cost. To achieve this objective, we set the weight function to be $w: c \in C(R_Q) \rightarrow 1/P(c)$.

Note that the overlap among conditions and number of elements covered by a selected condition do not need to be part of w, since they are considered directly in the approximation algorithm.

Figure 4.4 presents the ApproximateSetCover heuristic, which is an adaptation of the weighted set cover approximation algorithm [79] using the above defined weight function, and has a running time of $O(|C(R_Q)| \cdot |R_Q|)$ and an approximation ratio of $O(\log(|C(R_Q)|))$. Note that this approximation ratio assumes that the quantity we want to minimize is the sum of the weights ($1/P(c)$) of the selected conditions. However, the real objective of ApproximateSetCover is to minimize the navigation cost, which is much harder to bound, given that ApproximateSetCover does not capture all the details of Equation 4.1. Also note that this approximation ratio can be large if the number of conditions in $C(R_Q)$ is large. However, the number of facet conditions is generally small and this algorithm performs reasonably well in practice, as demonstrated by the experiments in Section 4.6.

Example 4.1: Figure 4.3(b) shows the result of the ApproximateSetCover heuristic on the result set R_Q in Figure 4.3. The algorithm requires two iterations of the while loop (lines 3-7) before terminating with the set of suggested conditions in Figure 4.3(b). In the first iteration, the algorithm selects $Make = Honda$, since this facet condition covers 4 results and has the maximum

value of $P(c) \cdot |R_{Q \wedge c}| = 3.2$ amongst all the conditions in $C(R_Q)$ and V is empty. In the next iteration, two results (t_1 & t_3) remain uncovered and are covered by facet condition $Year = 2005$.

4.5.4 UniformSuggestions Heuristic

In this heuristic we follow the cost formula in Equation 4.1 more closely, which leads to a more robust heuristic. Computing the optimal suggested conditions involves recursively evaluating Equation 4.1 for each combination of facet conditions in $C(R_Q)$. This translates to a very large (in both height and width) recursion tree. UniformSuggestions replaces this recursion tree with a set of very small recursion trees, one for each condition in $C(R_Q)$. For that, we evaluate the expected cost of each facet condition independently, assuming that all future suggested conditions will have identical properties, and then select the facet conditions with minimal expected cost, until all results in R_Q are covered.

In particular, the *uniform-condition* heuristic assumption states that for a given condition $c \in C(R_Q)$, evaluate the navigation cost using Equation 4.1, while assuming that every other condition in $C(R_Q)$ has the same *characteristics* as c. The characteristics of c are *(a)* its probability $P(c)$, and *(b)* the ratio $r(c) = |R_{Q \wedge c}|/|R_Q|$ of the uncovered results that c covers. This heuristic assumption reduces the search space of suggestions to $\left|C(R_Q)\right|$ as each condition is now evaluated independently. It also allows us to simplify the cost formula in Equation 4.1 as follows.

If each suggested condition in $C_S(R_Q)$ covers a ratio r of the results in R_Q, we need a total of $n = 1/r$ conditions to cover all the results in R_Q. Also, REFINEment by c narrows down R_Q to an estimated $|R_Q|/n$ number of results. On the other hand, if the user does not select a suggested condition and instead EXPANDs an attribute A_i, she views an additional $|C(A_i) \backslash C_S(R_Q)| \approx |C(A_i)|$ facet conditions. Also, in the absence of any prior knowledge about the selectivity of facet

conditions in $C(A_i)$, we assume that each $c' \in C(A_i)$ narrows down R_Q to an estimated $|R_Q|/|C(A_i)|$. Thus, we can simplify the recursion in Equation 4.1 as follows:

$$cost(c, |R_Q|) = P_{SR}(R_Q) \cdot |R_Q| + \left(1 - P_{SR}(R_Q)\right) \cdot \left[\begin{array}{c} B + n + \left(1 - P_E(c)\right) \cdot \sum_{i=1}^{n} \left(P_{norm}(c) \cdot cost\left(c, \frac{|R_Q|}{n}\right) \right) + \\ P_E(c) \cdot \sum_{A_i \in S_R} P_A(A_i) \cdot \left(\begin{array}{c} |C(A_i)| + \\ \sum_{c' \in C(A_i)} \left(P_{norm}(c') \cdot cost\left(c', \frac{|R_Q|}{|C(A_i)|}\right) \right) \end{array} \right) \end{array} \right] \quad (4.3)$$

where $P_E(c) = \left(1 - P(c)\right)^n$

Observe that instead of Q, the cost function in Equation 4.3 above uses c and $|R_Q|$ as arguments for this heuristic, since a cost is computed for each c, and only the number of results $|R_Q|$ is important. The parameter Q in the original cost formula (Equation 4.1) captured the query progression with REFINE actions, which is not required in this heuristic. In Equation 4.3 above, $P_{norm}(c)$ is the normalized probability of following one condition of *type* c. Since all n suggested conditions have the same $P(c)$, then $P_{norm}(c) = 1/n$. Therefore the cost component in Equation 4.3 for navigating all n suggested conditions can be rewritten as:

$$\sum_{i=1}^{n} P_{norm}(c) \cdot cost\left(c, |R_Q|/n\right) = cost\left(c, |R_Q|/n\right)$$

By a similar argument, and since every facet condition c' has the same characteristics as c in Equation 4.3, we can simplify the summation last line of Equation 4.3 as follows, where A_c is the attribute of c:

$$\sum_{A_i \in S_R} P_A(A_i) \cdot \begin{pmatrix} |C(A_i)| + \\ cost(c, |R_Q|/n) \end{pmatrix} = |C(A_c)| + cost(c, |R_Q|/|C(A_c)|)$$

Algorithm: UniformSuggestions(Q, R_Q)

Input: A query Q, a result set R_Q

Output: $C_S(R_Q) \subseteq C(R_Q)$, the suggested conditions.

---

1 $Q' \leftarrow Q$; $C_S(R_Q) \leftarrow \emptyset$; $Y \leftarrow R_Q$ // Y: uncovered results
2 $P_{SR} \leftarrow P_{SR}(R_Q, C(R_Q))$
3 while $Y \neq \emptyset$ do
4 foreach $c \in C(R_Q)$
5 $n \leftarrow |Y|/|Y \cap R_{Q' \wedge c}|$
6 $P_{SR} \leftarrow P_{SR}(R_Q)$
7 $u \leftarrow |Y|$
8 compute $cost(c, u)$ using Equation 4.4
9 endFor
10 Let $cmin$ be the suggestion with min $estCost(c, |Y|)$
11 $C_S(R_Q) \leftarrow C_S(R_Q) \cup cmin$
12 $Q' \leftarrow Q \wedge cmin$
13 $Y \leftarrow Y \backslash R_{Q' \wedge cmin}$
14 $C(R_Q) \leftarrow C(R_Q) \backslash cmin$
15 endWhile
16 return $C_S(R_Q)$

Figure 4.5. UniformSuggestions Heuristic

Therefore, the cost equation (4.3) can now be rewritten as:

$$cost(c, |R_Q|) = \begin{cases} |R_Q| & , |R_Q| < T \\ P_{SR}(R_Q) \cdot |R_Q| + \left(1 - P_{SR}(R_Q)\right) \cdot & \\ \left[\begin{array}{c} B + n + (1 - P_E(c)) \cdot cost(c, |R_Q|/n) + \\ P_E(c) \cdot \left(|C(A_C)| + cost(c, |R_Q|/|C(A_c)|)\right) \end{array} \right] & , |R_Q| > T \end{cases} \quad (4.4)$$

The recursion terminates when the size of the result $|R_Q|$ drops below a threshold T. Since a navigation should be able to narrow down the result to a single tuple, we set T to 1.

The algorithm, based on the uniform-condition heuristic assumption is presented in Figure 4.5. The algorithm computes the estimated $cost$ of each facet condition using the simplified cost formula in Equation 4.4 (lines 4-9), and selects the condition with the minimum $cost$ ($cmin$) to be added to the set of selected conditions (lines 10-11). Next, we remove from the set V of uncovered results the results covered by $cmin$. The algorithm terminates when all the results in R_Q are covered.

The result of applying the UniformSuggestions heuristic algorithm to the result set R_Q in Figure 4.3 is shown in Figure 4.3(c). Recall from the discussion in Section 4.5.1 that the cost model selects conditions with moderate selectivity and high $P(c)$. Under our heuristic assumption, a facet condition c is evaluated under the assumption that all conditions in $C(R_Q)$ have the same characteristics as c. Therefore, a condition with moderate selectivity and a high $P(c)$ has a lower cost when evaluated using the simplified cost formula in Equation 4.4 and these are just the conditions selected by the algorithm.

4.6 Experimental Evaluation

In this section, we present a thorough evaluation of the algorithms and heuristics described in Section 4.5 and show that FACeTOR achieves a significant decrease in navigation cost compared to current approaches The experiments are based on a large-scale simulation of user navigations presented in Section 4.6.2. The metric used is the average navigation cost as defined by the cost formula in Equation 4.1. Section 4.6.3 measures the time requirements of our heuristics and shows that they can be used for real-time interaction.

Table 4.2. FACeTOR Evaluation Query Workload.

Query	#Results	# of Facet Conditions	Query	#Results	# of Facet Conditions
***UsedCars* Dataset**			***IMDB* Dataset**		
honda	789	234	baldwin	112	1545
toyota	1470	366	oscar	189	2141
dallas	2932	990	love	415	2989
miami	211	230	American	111	1096
coupe	599	334	history	272	2716
sedan	1693	524	white	284	3058
2000	896	641	black	221	2327
2004	3711	1124	time	145	907
black	2391	972	john 2007	391	4545
gold	709	508	action 2007	272	2601

4.6.1 Setup

The primary goal of these experiments is to evaluate the effectiveness of the system in decreasing the user navigation cost for a set of query results. To this end, we compare the two heuristics presented in Section 4.5 to each other and to the current state of the art algorithm, which is the single-facet-based-search [33], henceforth called INDG. All experiments were conducted on a Dell Optiplex machine with 3GHz CPU and 3GB of RAM. We use MySQL as our database and Java for algorithms.

Datasets: We evaluate FACeTOR on two datasets, *UsedCars* and IMDB. We assume that the numeric attributes have been appropriately discretized. The *UsedCars* database was downloaded from Yahoo! Autos site and contains 15,191 car tuples with 41 attributes/facets. From the IMDB dataset, we extracted a total of 37,324 movies. For our experiments we only leveraged the movie, actors, directors, ratings and genre data. Note that actors, directors and genres are set-valued attributes, that is, each movie can have multiple actors and/or directors.

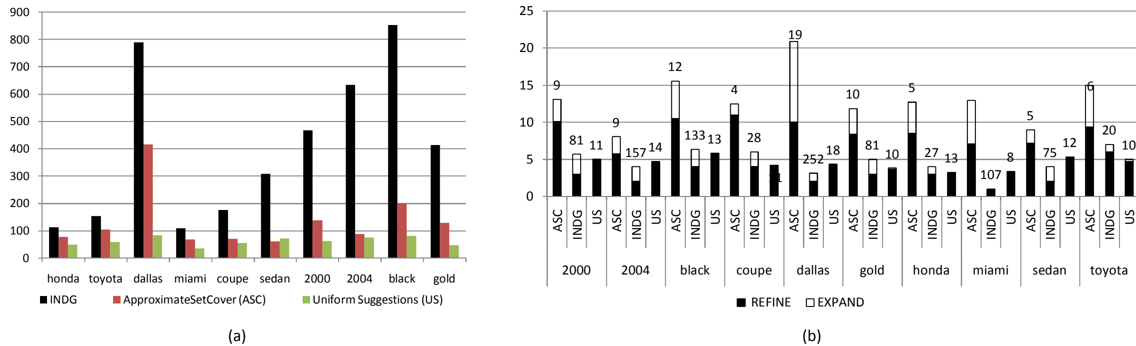


Figure 4.6. For the UsedCars Dataset: (a) Average Navigation Cost, and (b) Average Number of REFINE and EXPAND Actions, and Average Number of Suggested Conditions per Navigation Step (numbers on top of the bars), for $B = 1$.

Experimental Methodology: For each dataset, *IMDB* and *UsedCars*, we select a number of keyword queries (see Table 4.2) whose results from the initial result set R, and a random result tuple as the target for navigation for each query. Next, we measure the number of navigation actions (REFINE/EXPAND actions, facet conditions displayed and results viewed) incurred before reaching the target tuple as the navigation cost for the query. In our system, the target tuple can be reached by multiple navigations. For example, tuple t_4 in the result set of Figure 4.3 can be reached by REFINEing by any one of the two conditions in Figure 4.3(b).

Since, the user's navigation cannot be known in advance, we consider an evaluation approach that considers both these navigation paths. To account for uncertainty in user navigation,

we use a *guided randomized simulation* of user navigation. In this simulation, we randomly select one of the facet conditions $c \in C_S(R_Q)$ for navigation. The probability that the agent selects a condition c is proportional to $P(c)$, the probability that the user would know or likes the facet condition c. The simulation is guided in the sense that it only follows the paths that lead to the target result. For example, if the agent encounters the two suggestions in Figure 4.3(b) and the target is tuple t_3, the simulation would choose either $Make = Honda$ or EXPAND. The probability of choosing EXPAND is $\prod_{c \in C_S(R_Q)}(1 - P(c))$, where $C_S(R_Q)$ are the suggested conditions. We execute the navigation for each query 1000 times using this simulation technique and average the cost over the individual navigations. We also report the average number of times each navigation action is executed during the simulation.

The navigation cost is sensitive to the constant B according to the cost function in Equation 4.1. Varying these constants changes the set $C_S(R_Q)$ for UniformSuggestions, but not for ApproximateSetCover, since it does not consider B. Intuitively B denotes the patience of the user towards suggestions generated by the system. If the user sees a small number of conditions she would have to execute more REFINE actions to reach the result. Thus by setting B to a large value the user should typically see more suggestions per REFINE and vice versa.

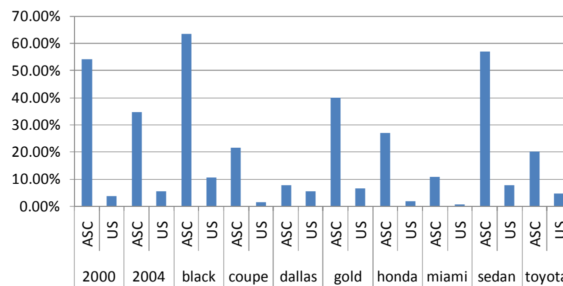


Figure 4.7. Average Overlap per Navigation Step for the UsedCars Dataset, for $B = 1$.

We experiment with different values of B and observe the effect on the overall navigation cost for the *UsedCars* query workload in Table 4.2. We also compare the number of suggested conditions generated (on average) and the number of REFINE actions. We compare our approach with the current state of the art INDG algorithm [33]. This algorithm constructs a decision tree that partitions the result set R_Q by a facet (attribute) at each level.

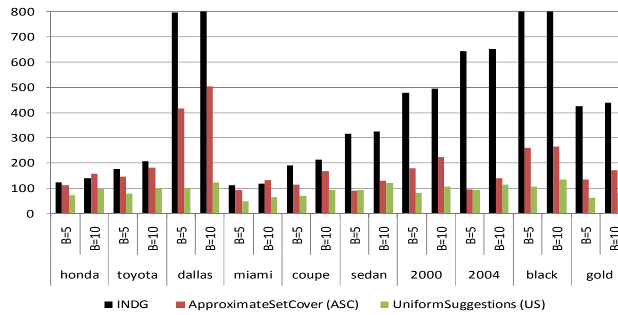


Figure 4.8. Average Navigation Cost for $B = 5$ and $B = 10$ (UsedCars Dataset).

The aim is to minimize the average depth of the tree in reaching the results. The user is presented with all the facet conditions on the attribute that forms the root of the decision tree. Since INDG generates suggestions from a single attribute, the simulation for this algorithm differs from above as follows: at each step, the agent chooses to EXPAND or REFINE by one of the suggestions of an attribute A_i with probability $P_A(A_i)$ and EXPAND action reveals all the facet conditions for a different attribute.

4.6.2 Experiments with Navigation Cost

The average navigation costs for the INDG, ApproximateSetCover and UniformSuggestions algorithms for the *UsedCars* queries in Table 4.2 are shown in Figure 4.6(a). As seen in the graph, by following our approach leads to significant savings in navigating cost. Figure 4.6(b) shows some of the individual components of the total cost for Figure 4.6(a), that is, the average number of REFINE actions, average number of EXPAND actions. Also shown (top of

the bars) are the average numbers of suggestions per navigation step. As expected, the INDG algorithm has very few REFINE and EXPAND actions, but reveals a large number of facet conditions, resulting in high overall cost. The INDG algorithm ignores the cost of inspecting suggested and therefore produces a large number of suggestions at each navigation step.

The average cost incurred by UniformSuggestions algorithm is less compared to ApproximateSetCover. ApproximateSetCover has a higher number of REFINE and EXPAND actions as compared to UniformSuggestions, even though the average number of suggestions at each navigation step is comparable. In each iteration, the greedy ApproximateSetCover algorithm selects a small set of facet conditions with a high value of $P(c)$ that also cover a large number of results. These suggested conditions therefore, have a low selectivity and therefore tend to have a high degree of overlap among the suggested conditions, as shown in Figure 4.7, thereby reducing the effectiveness of REFINE actions. Thus, the user has to perform many REFINE actions in order to reach the target result. Figure 4.8 shows the effect of increasing B, the cost of executing a REFINE. As expected, the average overall cost increases. The UniformSuggestions heuristic adapts to a changing value of B, whereas the ApproximateSetCover heuristic and INDG do not. Therefore the cost of UniformSuggestions increases at a slower rate than the other two algorithms. This is primarily because, for a higher B, UniformSuggestions generates more suggestions per REFINE/EXPAND.

The results of *IMDB* workload queries in Table 4.2 are shown in Figure 4.9. As in the *UsedCars* workload, the UniformSuggestions heuristic outperforms ApproximateSetCover. Also, the observations for the number of EXPAND and REFINE actions and the number of suggested conditions generated is also similar to those for the *UsedCars* dataset. However, the navigation cost with the UniformSuggestions algorithm is much lower than ApproximateSetCover. A *movie* in the

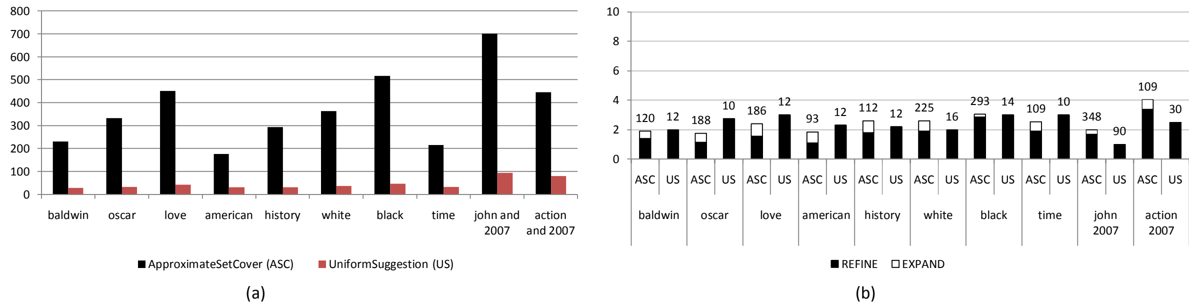


Figure 4.9. For the IMDB Dataset: (a) Average Navigation Cost, and (b) Average Number of REFINE and EXPAND Actions, and Average Number of Suggested Conditions per Navigation Step (numbers on top of the bars), for $B = 1$.

IMDB dataset can be classified into a large number of facet conditions. For example, each movie can have multiple actors or directors or genres. Therefore executing an EXPAND action reveals a very large number of facet conditions (the number on top of bars in Figure 4.9(b)), thereby significantly increasing the navigation cost.

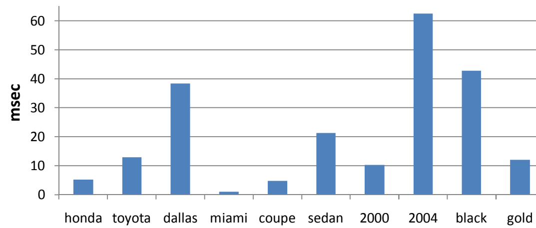


Figure 4.10. Average Execution Time of UniformSuggestions Heuristic (UsedCars dataset & $B = 1$).

4.6.3 Execution Time Evaluation

This experiment aims to show that UniformSuggestions is fast enough to be used in real-time. The average execution time of UniformSuggestions per REFINE action for the queries in Table 4.2 (*UsedCars* dataset) is shown in Figure 4.10. The execution time for this heuristic depends primarily on the number of facet conditions in the result set R_Q. As the number of facet conditions decreases, as is the case towards the end of navigation, the performance of

UniformSuggestions improves dramatically. In the interest of space, we omit reporting these values, as well as the results for ApproximateSetCover which, given its simplicity, is much faster.

4.7 User Evaluation

In this section, we present the results of a large scale user study we conducted to compare the user experience with FACeTOR and other state of the art interfaces. We measure the following: (a) the actual time it took users to navigate using different interfaces, (b) how realistic is our cost model, by studying the relationship of the actual time (actual cost) with the *estimated cost* and *(c)* the *user's perception* of the faceted interfaces through a questionnaire. By comparing the actual navigation time to the users' perception, we study if lower actual time corresponds to more intuitive (cognitively easier) interfaces.

We constructed 8 randomly created result sets of 1000 tuples from the *UsedCars* dataset and for each one we created a task that involves locating a set of *target tuples* (cars), which satisfy a set of attribute/value conditions. For each one of the 8 result sets, we showed the requested conditions to the users and asked them to locate the target tuples using three interfaces: (a) *FACeTOR*, (b) *Amazon-Style*, which suggests at most 5 facet conditions with the highest cardinality for each attribute, and (c) One-attribute-at-a-time *INDG* [33], where an attribute is selected at each step and all its conditions are displayed. We deployed our system on Amazon Mechanical Turk [24] task and collected a total of 37 responses.

Actual Time Figure 4.11 shows the actual time as well the average time taken by users to navigate each of the eight result sets using the three interfaces. As shown, FACeTOR speeds up the navigation by 18% and 37% over Amazon-Style and INDG respectively, even for relatively small result sets of 1000 tuples and short navigations consisting of only 4 REFINE actions (Figure 4.12).

This is primarily because users spend less time in reading suggested conditions and deciding which one to follow next, as evidenced by Figure 4.12. FACeTOR shows 36% fewer suggestions than Amazon-style and 57% fewer suggestions than INDG, while it requires the same number of REFINE and EXPAND actions (on average) to reach the target tuples. This is an indication of *high quality* suggestions provided by FACeTOR.

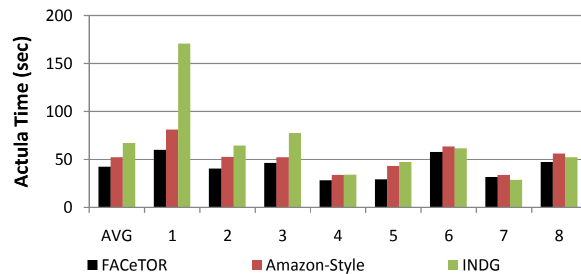


Figure 4.11. Actual User Navigation Time for 8 Result Sets.

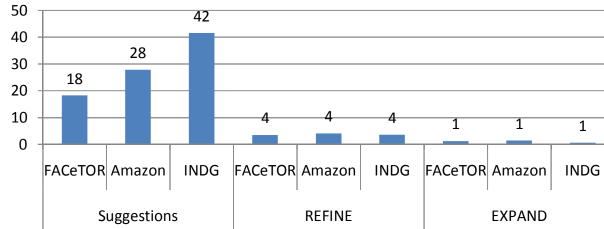


Figure 4.12. Average number of Suggestions and Actions.

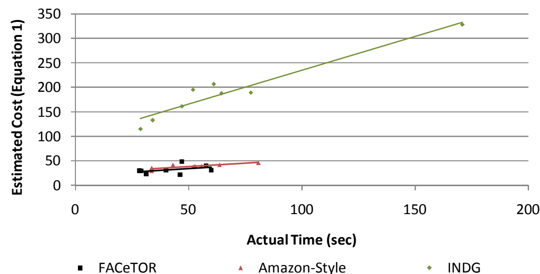


Figure 4.13. Actual Time vs. Estimated Navigation Cost.

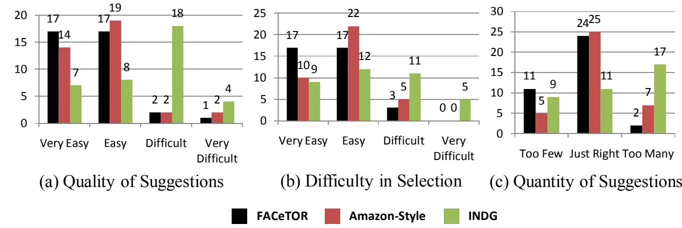


Figure 4.14. Users Perception (Questionnaire).

Estimated Cost Figure 4.13 displays the data points of actual time vs. estimated cost, as computed by Equation 4.1, for the eight result sets for the three interfaces. Based on these data points, Figure 4.13 also shows the trend line between actual time and estimated navigation cost for each interface. We observe that the actual time is linearly proportional to the estimated navigation cost for all three interfaces, which shows that our cost model is realistic.

Users Perception The study also included a questionnaire where we elicited the users' opinion on various aspects of the three interfaces, including the ease of use, size and intuitiveness of suggested conditions and preferred choice of interface. The results of this survey are shown in Figure 4.14. 92% of users said that they thought the suggestions presented by FACeTOR at each step made the task of locating the target tuples easier (Figure 4.14(a)), compared to 89% for Amazon-style and 40% for INDG. A large majority of users (92%) also said that the suggestions provided by FACeTOR had a low "cognitive dissonance" (Figure 4.14(b)) in the sense that it was very easy (45%) or easy (46%) to decide which suggestion to follow. The corresponding cumulative percentages for Amazon and INDG were 81% and 54% respectively. We also asked the users if the number of suggestions provided by the interfaces were adequate (Figure 4.14(c)). A significant percentage (30%) said that FACeTOR provided too few suggestions at each navigation step, indicating that users prefer more choices even if it means an increase in absolute navigation cost – a situation that could easily be remedied by increasing the value of constant B.

4.8 Related Work

Ranking Ranking could be applied in conjunction with a faceted interface. Chaudhuri et al. [20] use the unspecified attributes and apply Probabilistic Information Retrieval principles to rank the results of a database selection query. Various ranking techniques have also been proposed for keyword search on structured databases [13, 80] based on the size and relevance of the results.

Faceted Search on Structured Data Faceted search is employed by major e-Commerce websites (Amazon, eBay) that typically display all the facet conditions applicable to the current set of query results. If too many values are available for a facet, then the most popular are displayed, and a "more" button reveals the rest. In contrast, our approach displays only a subset of applicable facet conditions chosen to minimize the overall navigation cost. English et al. [81] was one of the first to introduce faceted search and discusses facets from a user interface perspective.

Our work is closest to the works of Chakrabarti et al. [23] and Roy et al. [33], which also use a navigation cost based approach for faceted navigation. In particular, we adopt ideas from both works and addresses their key shortcomings. In both these works, the navigation algorithm selects one attribute (or possibly multiple attributes [33]) and displays *all* the values of these attributes to the user. Alternatively, a text box could be displayed [33], but we believe that this is impractical, given all known values would have been in the original query. Our approach differs from these works, because at each navigation step, we display a mix of facet conditions from several attributes, that is, our algorithm operates at the attribute value level and not the attribute level.

Keyword-Based Faceted Search and Query Refinement: The GrowBag project [43] and Sarkas et. al [44] suggest additional search terms based on the co-occurrence patterns of these terms in the query result. The GrowBag algorithm [43] computes higher order *co-occurrences* of

terms in the document collection and suggests terms appearing in the *neighborhood* of each search term as refinement suggestions whereas [44] suggests terms that co-occur with search terms and narrow down the result-set to *interesting* subsets using the *surprise* metric. Our work is also related to query refinement systems [82, 83]. [82] recommends new terms for refinement such that the *recall* of the resulting query is maximized, whereas [83] uses *relevance judgment feedback* on the results to refine the query. Our approach also suggests facet conditions to refine the query, but we use the navigation cost as metric. Our navigation model is similar to BioNav [25, 27], which uses the ontological annotations of PubMed publications to create a navigation tree. A key difference is that in BioNav, there is a given concept hierarchy [58], which prunes the search space. In contrast, there is not such tree in FACeTOR, which makes the selection of a set of faceted conditions harder.

OLAP: A faceted interface can be viewed as an OLAP-style cube over the results. Wu et al.[84] generate hierarchical partitions over the query results based on a cost model for user navigation and display this hierarchy to the users. The interestingness of group-by aggregations is used to rank candidate aggregations to display.

4.9 Summary

Faceted navigation is employed to reduce the *information-overload.* The effectiveness of these interfaces is limited as they often show too many or irrelevant facet conditions. Our system addresses these problems by selectively showing a subset of the available facet conditions that are selected based on an intuitive cost-based navigation model that attempts to minimize the navigation cost by hiding uninteresting or ineffective conditions. We provide feasible solutions for this problem and demonstrate their effectiveness by a thorough experimental evaluation and a user study.

Chapter 5

Comprehension-Based Result Snippets

5.1 Introduction

A large number of databases are heterogeneous in nature. Examples of such databases include product catalogs (Amazon, eBay etc.) and medical data (Stanford diabetes study, patient records, Human Genome project), among many others. Such heterogeneous data is characterized by a high structural variance amongst the objects in the database, where objects have different and usually overlapping sets of attributes. As an example, consider the Amazon product catalog which consists of objects of different types including *Laptop, Desktop* and *Camera etc*.

Each object type is associated with different and possibly overlapping schemas. For instance, *Laptop* has attributes *Price, Display Size*, *Fingerprint Reader* etc., whereas *Camera* has attributes such as *Price, Shutter Speed, Zoom* etc. As another example, a patient record stores various types of data, e.g., a diabetic patient's record includes *Blood Pressure*, *Blood Sugar* level and *Insulin Dosage,* whereas a patient's record with *Osteoporosis* includes *Bone Density*, *Calcium Level,* etc.

Keyword search is the dominant query interface in most such systems. Hence, naturally the query answer typically consists of a list of heterogeneous objects, due to the ambiguous nature of keyword queries. Query interfaces use a variety of methods to help users find the results that they

are most interested in, like ranking[15, 16, 41] , categorization (facets) [23, 27, 28], and result snippets.

A snippet is a summary of the information contained in a result and its purpose is to help the user make a decision about the relevance of the given result. As an example, consider snippets for the results of the query *'acer laptop 3gb'* on an e-commerce Web site, as shown in Figure 5.1. Amazon.com, as most other similar systems, uses a fixed hardcoded snippet schema for each type of object, as in Figure 5.1(b). In particular, Amazon.com always displays attributes *Brand, Model, Display, Model Name* and *Color* for *Laptop* results (only *Brand* and *Display* are in our result-set). This is clearly suboptimal, since such snippets do not always allow differentiating among the displayed results ([21] makes this point for XML results). For example, the snippets of two *Laptop* results (#1 and #3) in Figure 5.1(b) show the same attributes and values. Instead, it is beneficial to include a discriminating attribute for Acer laptops (e.g. *Cover Material*) in the snippet, which may not be important for other results in the query. Further, this fixed-schema approach is inefficient for diverse result-sets [22, 85, 86] in which the results have few attributes in common.

The only work we are aware of on snippets construction for structured data, studies snippets of XML results, has focused on the *informativeness* of the snippets, which describes how useful the information on the snippets is to help user select a result, e.g., how representative or distinguishable the snippets are [21, 22]. For instance, if many results have *Display Size=11.3"*, this information should be displayed on the snippets. Figure 5.1(c) shows a list of snippets generated with informativeness in mind. Comparing Figures 5.1(b) and 5.1(c), we observe that the snippets in Figure 5.1(b) appear to have a somewhat uniform schema (at least for items of the same type), while the ones in Figure 5.1(c) look very jagged and disorganized. The tradeoff is that the more uniform snippets are easier to read, while the disorganized ones may offer more useful

information about the returned results. The goal of snippets is to minimize the user effort (time) in finding the results of interest. Hence, we argue that *the user time spent reading the snippets is important as is the information on the snippets*. Figure 5.1(d) shows an example of a snippet that is both highly *informative* (contains a number of attributes) and is easy to *comprehend* since the attributes in snippets of results are aligned and are therefore easy to read.

ID	Type	Brand	Memory	Capacity	Processor	Display	Cover Material	Price
1	Laptop	Acer	4Gb		Intel i5	13.3"	Aluminum	$1299
2	Tablet	Acer		16Gb	Nvidea Ion	7"	Gorilla Glass	$380
3	Laptop	Acer	3Gb		Intel i5	13.3"	Carbon Fiber	$1250
4	HDD	Seagate		500Gb			Plastic	$200
5	Memory	Kingston	2Gb					$100
6	Memory	Corsair	4Gb					$150

(a) Result-set of query '*acer laptop 3gb*'

1	*Brand*: Acer		*Display:*13.3"
2	*Brand*: Acer	*Capacity*: 16Gb	*Display*: 7"
3	*Brand*: Acer		*Display:*13.3"
4	*Brand*: Seagate	*Capacity*: 500Gb	
5	*Brand*: Kingston		
6	*Brand*: Corsair		

(b) Fix Schema Snippet, with high information loss

1	*Memory:*4Gb	*Processor:* Intel i5	*Price:* $1299
2	*Capacity:*16Gb	*Display:* 7"	*Cover:* Gorilla Glass
3	*Brand*: Acer	*Processor:* Intel i5	*Display:* 13.3"
4	*Capacity:* 500Gb	*Cover*: Plastic	*Price*: $200
5	*Brand*: Kingston	*Memory*: 2Gb	*Price*: $100
6	*Brand*: Corsair	*Memory*: 4Gb	*Price*: $150

(c) A very informative, but hard to read snippet

1	*Cover:* Aluminum	*Memory:*4Gb	*Processor:* Intel i5
2	*Brand*: Acer	*Display:* 7"	*Processor:* Nvidea Ion
3	*Cover:* Carbon Fiber	*Display:* 13.3"	
4	*Brand*: Seagate	*Capacity:* 500Gb	*Price*: $200
5	*Brand*: Kingston	*Memory:*2Gb	*Price*: $100
6	*Brand*: Corsair	*Memory:*4Gb	*Price*: $150

(d) An informative and easily comprehensible snippet

Figure 5.1. A heterogeneous result-set for query 'acer laptop 3gb' and three snippets with different characteristics.

No work has studied the *comprehension cost* of structured snippets, which is the user effort required to read and digest the information displayed by the snippets. We argue that the comprehension cost should be taken into consideration during the snippets generation process. In this chapter, we propose a methodology to construct snippets that simultaneously minimize the comprehension effort and information loss (i.e. maximize informativeness). Our work achieves this goal as follows:

First, we propose the first model for the user comprehension cost of reading structured snippets. We perform user surveys that confirm our intuition that by indenting the snippet attributes in a way that common attributes have the same position across the snippets we dramatically reduce the user comprehension cost. In particular, we show the surprising result that *the comprehension cost for an attribute does not depend on the number of snippets that contain it, but only on the number of different positions where it appears in the snippets*.

Next, we define a quantitative model for the information content (termed *informativeness)* of snippets with respect to the complete results. We leverage previous work on snippet informativeness and adapt it to structured objects.

We analyze the problem of constructing optimal snippets, i.e., minimizing both the comprehension cost and the information loss, for a list of results and show that this problem is NP-hard. We present efficient algorithms for snippet construction and evaluate their performance and efficiency.

5.2 Framework and Definitions

Definition 5.1 (Database): The database is a single relation D with m attributes $A = \{A_1, \ldots, A_m\}$. Each attribute A_i has an associated active domain $ADom(A_i)$ of un-interpreted constants, which includes the *null* value. The database D is sparse and heterogeneous, i.e. tuples $r \in D$ have values for different subsets of A, and have the ϵ value for the rest of the attributes. We use $A^r \subseteq A$ (typically $|A^r| \ll |A|$) to denote the set of attributes of a tuple r .

Definition 5.2 (Result-set): A user exploring D typically submits a query and the system returns a ranked result-set $R = \{r_1, \ldots, r_n\} \subseteq D$ of objects (we use the terms object, tuple and result interchangeably depending on the context).

Example 5.1: Figure 5.1(a) shows a subset of the results of query *'acer laptop 3gb'* on Amazon.com. Only seven (we do not count *Type* as an attribute) of nearly hundred attributes are shown. The query returns not only *Laptops* (indicated by the *Type* attribute), but also results of type *Memory*, *Tablets* and *Hard Disk Drives*. The schema of each object depends on its *type* and can have common attributes with other types (e.g. *Price*), but also different attributes (e.g. *Laptops* and *Tablets* have a value for attributes *Display* and *CPU*, whereas *Memory* does not).

A snippet with many attributes can be difficult to present and can overwhelm the user with details. Therefore we typically require that the size of each snippet be bounded to k attributes. (The snippets in Figures 5.1(b-d) have size $k \leq 3$). Note that the size of a snippet could also be defined in other ways like the number of characters. However, we have found that the number of attributes offers a reasonable bound that also allows a structurally uniform presentation (e.g., in tabular form).

Definition 5.3 (Result Snippet): A snippet $s(r)$ for a result r is a k-tuple $< A_1 = a_1, \dots, A_k = a_k >$, where $A_i \in A^r$ is an attribute of r or is empty, and $a_i \epsilon Dom(A_i)$ is a value. To simplify the presentation, we often denote $s(r)$ as $< A_1, \dots, A_k >$ and use $|s(r)|$ to denote the number of attributes in $s(r)$. E.g., the first snippet *r* in Figure 5.1(b) is <*Brand, null, Display*>, and |*s(r)*|=2.

Definition 5.4 (Result-set Snippet): A result-set snippet $S(R, k)$ of a result-set $R = \{r_1, \dots, r_n\}$ is $S(R, k) = \{s(r_1), \dots, s(r_n)\}$ where $s(r_i)$ is the snippet of result r_i, and $|s(r_i)| \leq k$.

Figures 5.1(b-d) show example result-set snippets of the result-set in Figure 5.1(a). It is possible that results of same type have different attributes in their snippets, e.g. the first and the third *Laptop* snippets in Figure 5.1(c). To construct $S(R, k)$, a subset $A^s \subseteq A^r$ of size at-most k has

to be selected for each result r, and the attributes in A^s have to be ordered as a k-tuple. The order (position) of the attributes is important factor for the comprehension cost, as we explain in Section 5.3.

Result-set Snippet Construction Problem: Before formally defining the problem, we must define what a good result-set snippet is. Let function $\mathcal{F}(S, R, k)$, which will be defined below, be the goodness of $S(R, k)$, given R and k. By goodness, we mean that $S(R, k)$ simultaneously minimizes the *comprehension cost* and maximizes the *informativeness*.

Given a result-set R and snippet size bound k, construct a result-set snippet $S(R, k)$ such that:

$$S(R,k) = argmax_{S'(R,k)}\big(\mathcal{F}(S'(R,k), R, k)\big) \qquad (5.1)$$

To capture the comprehension effort, we introduce the following function predicate: $Compreh(S, R, k)$ that quantifies the user effort in reading and understanding the result-set snippet $S(R, k)$. Analogously, $Inform(S, R, k)$ captures the informativeness of the result-set snippet.

The goodness of a result-set snippet decreases with increasing comprehension effort, i.e.

$$\mathcal{F}(S, R, k) \propto 1/Compreh(S, R, k)$$

and increases with the informativeness, i.e.

$$\mathcal{F}(S, R, k) \propto Inform(S, R, k)$$

To combine the two competing factors, we formulate the snippet construction problem as a bi-criteria optimization problem and introduce a trade-off parameter $\lambda \in [0,1]$. The range (and units) of $Compreh$ is different from that of $Inform$. Therefore, to avoid having the goodness be

dominated by a single factor, we choose to define the optimization function as a product, instead of the more common linear combination, as follows:

$$\mathcal{F}(R,S,k) = \frac{Inform(S,R,k)^{\lambda}}{Compreh(S,R,k)^{1-\lambda}} \quad (5.2)$$

Intuitively, smaller values of λ lead to result-set snippets with smaller comprehension cost, which translates to fewer unique attributes and stricter alignment of same attributes (Section 5.3), whereas a large value of λ favors more informative snippets.

5.3 Comprehension Model

In this section we model how users read snippets and present a comprehension cost model. We start by describing the process by which a user reads and understands a list of results and identify the factors that affect comprehension. Next, we describe the details of a user study specifically designed to study the effect of the aforementioned factors on comprehension effort. Finally, we present the results of this study and use it to formulate a comprehension model that quantifies this effort.

5.3.1 Comprehension Model and Factors

Users can read tables horizontally −one snippet at a time− or vertically –one column at a time. When users view a result-set snippet, they generally look for attributes of interest and for each such attribute scan all snippets to see its value in other results, in order to get a picture of the result-set. E.g., in Figure 5.1(d), the user may scan the result-set snippet and get interested in attribute *Memory*, and then scan vertically to examine the value of *Memory* in other snippets. Then, the user may pick *Brand* and repeat the process. Eventually, the user will have *comprehended* the result-set snippet, that is, examine all attributes of interest to her, in order to make a decision (e.g.

select a result or refine the query). The comprehension cost $Compreh(S, R, k)$ is the total user cost during this process.

A key *assumption* is that the effort required to locate the attributes of interest and their values, depends on how they are arranged in the result-set snippet. More specifically, we hypothesize that if all the snippets have the same schema (as in Figure 5.1(b)), then the user first determines the position of a given attribute that she is interested in and then reads its values for all snippets. The associated comprehension cost then mainly consists of the cost to locate the attribute position, since reading (comparing) a list of aligned values entails an almost fixed effort, as we show below. However, fixing the schema is not possible for heterogeneous result-sets when *informativeness* (e.g., diversity) must be taken into consideration. This leads to increased user effort and thereby increased comprehension cost.

Another factor that possibly affects comprehension is the number of times an attribute $(say\ A_i)$ appears in the snippets. For example, if an attribute appears multiple times in the snippets, then the user would have to locate each instance of the attribute to satisfy her information need, thereby increasing the comprehension effort. For example, if the user is interested in the *Brand* attribute in Figure 5.1(c), then she would have to locate its three occurrences. Based on the above discussion, we identify two factors that *may* play a role in the comprehension effort:

- $nOccur(A_i)$: number of times an attribute appears in the result-set snippet, e.g., 3 for *Memory* in Figure 5.1(c).
- $nPos(A_i)$: number of unique positions of an attribute in the result-set snippet, e.g., 2 (first and second) for *Memory* in Figure 5.1(c).

Hence, the comprehension cost for an attribute A_i in the result-set snippet $S(R,k)$ is a function:

$$Compreh(S,R,k,A_i) = f\big(nPos(A_i), nOccur(A_i)\big) \tag{5.3}$$

Note that we overload $Compreh(.)$ from Section 5.2.

Given the above discussion on vertical scanning of the snippets, the overall comprehension cost for $S(R,k)$ is approximated by the sum of the costs of its attributes, that is:

$$Compreh(S,R,k) = \sum_{occursIn(S,A_i)} Compreh(S,R,k,A_i) \tag{5.4}$$

where $occursIn(S,A_i)$ returns true is attribute A_i is in at least one of the snippets in $S(R,k)$.

Equation 5.4 shows that only two factors affect the comprehension cost. Although this is supported by our user surveys below, we acknowledge that there can be several other factors that affect comprehension cost such as *comprehension difficulty* of attribute names and values. For example, understanding a *Legal Disclaimer* attribute of results requires more effort than understanding the *Color* attribute. Yet another factor could be the format in which attributes are displayed – e.g. highlighting attributes. However, such factors cannot be quantitatively modeled in any straightforward way, e.g. comprehension difficulty is data-dependent, and highlighting depends on the presentation design. We leave the study of such additional factors as future work.

What is left is to study the properties of function $f(.)$ in Equation 5.3. As mentioned above, comprehension of snippets is a complex activity involving a number of factors such as locating, reading and understanding the data present in the snippets. The effort or cost of these actions is subjective and is difficult to measure. Instead, we propose to measure the overall effort by measuring the time taken by a user to complete a comprehension task. Next, we describe the

user experiment we conducted to measure the effort in comprehending an attribute A_i, namely, $\left(f\left(nPos(A_i), nOccur(A_i)\right)\right)$ in a result-set snippet.

Table 5.1. Combinations of $nOccur(A_i)$ and $nPos(A_i)$. Each cell is a task.

$nOccur(A_i)$	*1*	*4*	*8*	*12*	*15*
$nPos(A_i)$	1	1	1	1	1
$nPos(A_i)$		2	2	2	2
$nPos(A_i)$		4	4	4	4

5.3.2 User Study Setup

To determine the effect of the parameters of our comprehension model, namely number of positions $nPos(A_i)$ and its occurrences $nOccur(A_i)$ in snippet $S(R,k)$, and the relationship between them, it is necessary to determine the time it takes for users to comprehend the given attribute for different configurations of these parameters. More concretely, for a snippet size k and a result-set size of n ($> k$), an attribute can be present in snippets of all or some (between 1 and n) of the n results and can be placed in any number of positions between 1 and k. Measuring the time taken by users to *comprehend* the attribute in these multiple configurations gives the estimated *relative* effort.

For this study, we manually constructed snippets for results of queries on a popular e-commerce website. The snippets were constructed for first 15 results of each query and the snippet size k was fixed to *k*=6. An attribute that appears in 5 or fewer snippets can appear in at-most as many positions, whereas an attribute that appears in 6 or more (up to 15) result snippets can appear in between one and 6 positions, giving a total of 75 $\left(\sum_{i=[1,5]} i + 10 \times 6\right)$ possible configurations of an attribute in a result-set snippet. Instead of checking for all 75 configurations, we test on a subset consisting of 16 configurations, as shown in Table 5.1.

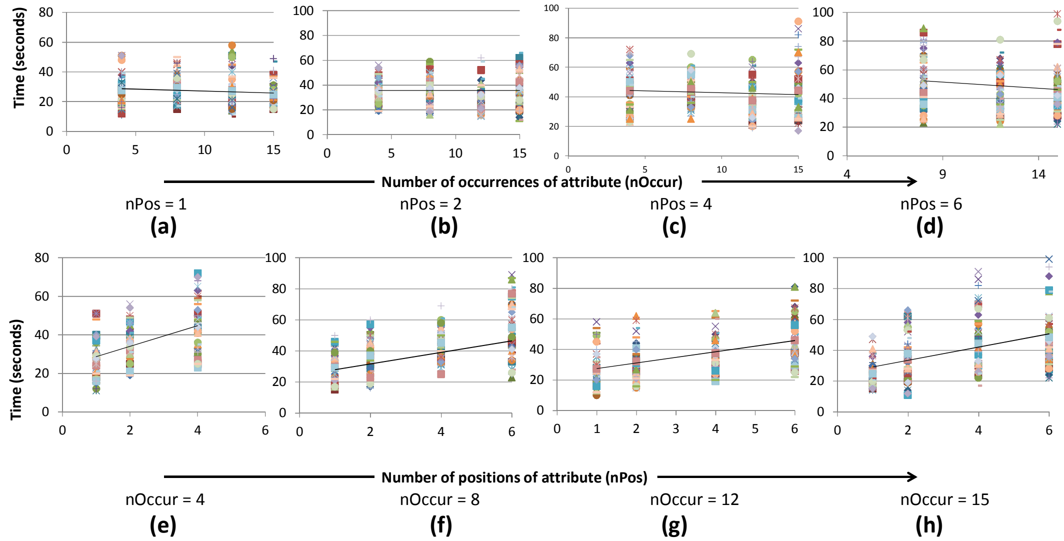


Figure 5.2. Comprehension Cost User Study Results.

In particular, we chose five different values for the number of occurrences $nOccur(A_i)$ of attribute A_i, and for each value, we consider a number of positions $nPos(A_i)$ of A_i according to Table 5.1. For example, for Table 5.1 entry $(nOccur, nPos) = (8,2)$ corresponds to a snippet in which a particular attribute (e.g. *Price*) appears in 8 results and in 2 (vertically aligned) positions. The user is asked a single question about a particular attribute in the result set. These questions are designed to gauge the overall comprehension of the attribute. A sample question for the task of query '*acer laptop 3gb*' (Figure 5.1), might be '*Which product has the maximum Price?*' For each task, we measure the time taken to answer the question correctly, which estimates the value of $f(.)$.

5.3.3 User Study Results

We deployed the user study on Amazon Mechanical Turk [24]. Figure 5.2 shows the response times of users to answer the question for different configurations of attribute arrangement (except $nPos = nOccur = 1$, which cannot be easily compared). This experiment was repeated to test for repeatability and the results shown in Figure 5.2. The first row of Figure 5.2 shows the plot of response times for differing number of occurrences of an attribute, while keeping the number of attribute positions in snippets ($nPos$), fixed. Here, we observe that for a given number of positions, the mean response time does not vary significantly based on the number of occurrences ($nOccur$).

For example, Figure 5.2(a) shows that when the number of positions of an attribute is fixed to 1 (second line, in Table 5.1), the mean response times for 4, 8, 12 and 15 occurrences of the attribute were 27, 29, 27 and 24 seconds, respectively, indicating that the response times, and therefore *effort, does not depend much on the number of occurrences of the attribute*. This observation is supported by Figures 5.2(a-d) and also by statistical hypothesis tests for equivalence, the two one-sided testing (TOST) procedure, developed by Schuirmann et al.[87].

Intuitively, the reason is that once the user located the position, it is fast to make a vertical scan to check the values of this attribute in the other snippets. Of course, the above observation assumes that the number of snippets is reasonably small (15 in our experiment), which is the case in practice, given that the result-set snippet must fit in the screen. On the other-hand, as shown in Figures 5.2(e-h), the number of positions of an attribute does affect navigation cost. This is because, the user has to expend more effort in navigating the result and check for each position of the snippet and look for a particular attribute. In particular, we see that the user time increases linearly with the number of positions. We summarize our finding as follows. For a result-set snippet:

Observation 1: The comprehension cost does not depend on the number of occurrences of an attribute.

Observation 2: The comprehension cost increases linearly with the number of different positions of the attribute.

Therefore, the per-attribute cost function $f(.)$ can now be expressed as follows

$$f\big(nPos(A_i)\big) = a \cdot nPos(A_i) + b \quad (5.5)$$

To compute a and b we fit the response time against the number of positions into a linear function and obtained the following function, where the unit is seconds.

$$f\big(nPos(A_i)\big) = 5.17 \cdot nPos(A_i) + 22 \quad (5.6)$$

We also experimented with higher order functions, but observed that they did not fit well with the data, which confirms our initial linearity observation. Note that the particular values for a and b depend on the nature of the result-set snippet, and particularly on factors like the number of snippets (15 in our experiment) and the comprehension difficulty of the attributes and values (see discussion in Section 5.3.1).

5.4 Informativeness

In this section we present a set of factors that make a result-set snippet informative. In the example of Figure 5.1, it is useful to show the *Price* attribute since a user at an e-commerce website is typically very interested in the price of the product. The importance of an attribute in a result-set is subjective and depends on factors such as user preferences, global (result-set-independent) attribute importance or the distribution of attribute values in the result-set. Furthermore, informativeness could be defined at the attribute value level, instead of the attribute

level. This is particularly desirable when different values have different importance for the user. For example, $Special\ Offer = true$ is more important than $Special\ Offer = false$ since it is advantageous to display a special offer to the user, if it is available for the given product.

There is no previous work defining the informativeness (or usefulness) of tabular snippets. For that, we borrow ideas from works on faceted navigation [28, 33], results diversity[85, 86, 88, 89], text snippets[90, 91] and XML snippets [21, 92]. These works define desirable principles for useful attributes or snippets, but do not provide a quantitative measure to compare the usefulness of two snippets. Note that this section should not be viewed as a key contribution of our work, but is included for completeness. We introduce the function $I(.) \rightarrow \mathbb{R}^+$ to quantify the informativeness, and specifically two variants:

- $I(A_i, R)$: Informativeness of attribute A_i in result-set R
- $I(A_i, R, v)$: Informativeness of value v of A_i in R.

Without loss of generality, we assume that $I(.)$ assigns higher values to more informative attributes, i.e., if $I(A_i, R) > I(A_j, R)$, then it is preferable to include A_i in the result-set snippet instead of A_j.Some of the attribute usefulness factors that have been proposed in previous work are:

- *Distinguishability*: snippets should show the differences between results [22, 33, 93].
- *Diversity*: showing a variety of attributes gives to the user a broader view of the results [85, 88, 94, 95].
- *Importance*: show attributes that are more important in the result-set than in the whole database [21, 22, 96].

Next we present concrete ways to quantify $I(A_i, R)$ and $I(A_i, R, v)$, which use some of the proposed factors, and we use in our experiments. We define $I(A_i, R)$ to be equal to *inverse* of the *indistinguishability* (INDG) [33] of the values of A_i in *R*:

$$I(A_i, R) = \frac{N(A_i, R)(N(A_i, R) - 1)}{2} - INDG(A_i, R) \tag{5.7}$$

where $N(A_i, R)$ is the number of occurrences of attribute A_i in R and $INDG(A_i, R)$ is the indistinguisability score defined as

$$INDG(A_i, R) = \sum_{a \in ADom(A_i, R)} \frac{|D(a, R)|(|D(a, R)| - 1)}{2}$$

where, $ADom(A_i, R)$is the active domain of A_i in the resultset R, which also includes the null value, and $|D(a, R)|$ is the number of times a value $a \in ADom(A_i, R)$ appears in R. Alternatively, we could use the entropy of these values or a user-specified global importance of the attributes (e.g., *Price* is more important than *Processor*). For example, in Figure 5.1(a), *I(Brand,R)=12*, *I(Capacity,R)=2* (due to the 3 null values). Next, we define the total informativeness of result-set snippet $S(R, k)$ as the sum of the informativeness of its attributes:

$$Inform(S, R) = \sum_{A_i \in S} I(A_i, R) \tag{5.8}$$

The above formula assumes that the scores of attributes are independent of each other, which is a reasonable simplifying assumption, but clearly not true for all informative definitions. A number of value-specific informativeness functions ($I(A_i, R, v)$) can be defined, such as in[21]. The choice of such function is orthogonal to our problem, and for the rest of the chapter we focus only on attribute specific informativeness. The algorithm discussed in Section 5.6 can be adapted for value-specific informativeness functions.

5.5 Complexity Results

In this section we study the complexity of the Snippet Construction Problem. We consider a simplified version of the Snippet Construction Problem, termed Fixed Snippet Construction (FSC), where the comprehension cost is the number of attribute positions. The key simplification in FSC is that it does not try to maximize informativeness. By showing that FSC is NP-hard, we also show that Snippet Construction Problem is NP-hard.

FSC Problem: Given a result-set R, construct a result-set snippet $S(R)$ (FSC has no snippet size constraint *k*), such that the comprehension cost is up to L and each snippet in $S(R)$ is non-empty. The comprehension cost for an attribute is the number of positions it appears in, and the comprehension cost of $S(R)$ is the sum over all attributes in $S(R)$. The informativeness is constant.

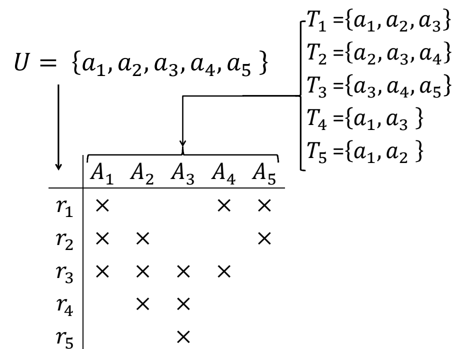


Figure 5.3. Reduction of Set Cover to the Fixed Snippet Construction (FSC) Problem.

Theorem 5.1: FSC is NP-Complete.

Proof: The problem is obviously in NP. Given a result-set snippet $S(R)$, it is easy to verify that *S(R)* has comprehension cost L. To prove that the problem is NP-Complete, we reduce the Set Cover Problem (SCP) to FSC.

SCP: *Given a set of elements* $U = (a_1, \dots, a_n)$ *and a set of subsets* $T = \{T_1, \dots, T_m\}$ *of* U *find a subset* $T' \subseteq T$ *of size at most Z, such that* $\bigcup_{T_i \in T'} T_i = U$.

Given an arbitrary instance (U, T) of SCP, we construct an instance of FSC as follows. For each $a \in U$, create a result r_a. For each $T_i \in T$, create an attribute A_i, and add this attribute to all results r_a whose corresponding element a is in T_i. Figure 5.3 shows this reduction. Recall that results are heterogeneous, that is, they have different attributes. Finally, we map *Z*=*L*. We now show that this mapping is indeed a reduction. A solution $S(R)$ to FSC is mapped to a solution T' to SCP by including to T' the subsets that correspond to the attributes in $S(R)$. Note that when we add an attribute A_i to $S(R)$, we simply add it to the snippets of all results that contain A_i since it does not increase the comprehension cost and there is no limit *k* on the size of a snippet. A solution to FSC is a solution to SCP because every result is non-empty, which means that every element in the universe in SCP is selected at least once. The comprehension cost of $S(R)$, which is the number of attributes in $S(R)$, is $|T'|$. Similarly, we can show the other direction of the solutions mapping.

5.6 Snippet Construction Algorithm

Challenges: Due to the intractability result of Theorem 5.1, in this section we propose efficient approximate algorithms for the Snippet Construction problem. Intuitively, there are two sources of intractability regarding the comprehension cost:

- how to select which attributes to display in each snippet of the result-set snippet, and
- how to arrange them, i.e., assign positions.

To minimize the comprehension cost, we want to select common attributes across the snippets and assign them the same position, as in Figure 5.1(d). However, we must also consider

informativeness, which further complicates computation. Note that the informativeness contribution of an attribute or value in the result-set snippet S does not depend on the other attributes or values in S, according to the formulas presented in Section 5.4, which is clearly not true for the comprehension cost. Hence, the latter is the main complexity source that our algorithms must tackle.

Algorithm Overview: To create an efficient approximate algorithm, we carefully relax both intractability sources listed above. The result-set snippet construction algorithm, presented in Figure 5.4, iteratively constructs a result-set snippet $S(R,k)$ by greedily evaluating and adding one attribute at a time. The selected attribute is placed in the minimum number of positions possible in the partially constructed result-set snippet. We can view the result-set snippet as an initially empty $n \times k$ matrix. At each iteration, we select an attribute A_i and add it to the row of each result r that contains A_i, i.e., $A_i \subseteq A^r$. We continue until either the matrix is full or adding an attribute decreases the goodness of the result-set snippet, as computed by Equation 5.2. The algorithm works by maintaining a $pool$ of candidate attributes that can be added to the snippet $S(R,k)$ along with auxiliary information about which results and the number of positions the attribute can be placed in the snippet matrix and a *heuristic* goodness score based on number of positions an attribute can occupy in $S(R,k)$. The attributes in the $pool$ are processed in the decreasing order of score and the remaining entries in the pool are updated to reflect this addition.

Algorithm Details: As a first step, the algorithm initializes a $pool$ of candidates (line 2). Each attribute A_i in the result-set R is represented in the pool by an entry e_i of the form $e_i\!: < A_i, nOccur, nPos, score >$ which includes the number of times ($nOccur$) and the number of positions ($nPos$) the attribute will appear in the final result-set snippet S. The entry also stores the $score$ as defined by Equation 5.2, which is computed by assuming that the snippet consists solely

of the given attribute placed in the given number of positions ($nPos$). Of course, the algorithm also has knowledge of which results in R contain which attributes.

The $pool$ is implemented as a priority queue arranged by decreasing $score$. The $pool$ is initialized (lines15-20) with entries for all attributes in R in their most optimal arrangement, i.e., assuming that all values of an attribute are added to the snippet in perfect alignment ($nPos = 1$). Next, in lines 3-14 the snippet is built iteratively by processing attributes in decreasing order of $score$. At each step, the attribute with the maximum score is chosen (line 7) and added (line 13) to the result-set snippet with the configuration (places and positions) dictated by the entry.

Given that an attribute is added independently of others, it is possible that the entry being processed cannot be added to the snippet in the configuration dictated by the entry – in $nPos$ positions occupying $nOccur$ in the snippets. This situation arises when potential spots are filled up by other attributes in previous iterations.

For instance, attribute A_5 may be part of the 2nd and 4th result of the result-set, but the result-set snippet matrix does not have any position (column) for which both the cell of the 2nd and 4th rows are free. This situation is handled in lines 5-6, where this incompatibility is checked and the $pool$ is recomputed (lines 21-26) by adjusting the positions and places the remaining attributes can occupy, given the snippet $S(R, k)$ computed this far.

Algorithm: *SnippetConstructionAlgorithm*

Input: Result-set R, snippet size bound k and trade-off parameter λ.

Output: Snippet $S(R,k)$ of R with size bound k

1.$prevScore = 0,\ inform = 0,\ compCost = 0$

2.$pool < A_i, nOccur, nPos, score > \leftarrow$ initPool()

3.**while** $(pool.size > 0\ and\ S\ is\ not\ full)$

4. $e \leftarrow pool.peekMax()$

5. **if** $(incompatible(e,S))$

6. recomputePool();

7. $e \leftarrow pool.removeMax()$;

8. $prevScore \leftarrow \frac{inform^{\lambda}}{compCost^{1-\lambda}}$

9. $inform \leftarrow inform + Inform(e.nOccur)$

10. $compCost \leftarrow compCost + Compreh(e.nPos)$

11. **if** $\left(prevScore < \frac{inform^{\lambda}}{compCost^{1-\lambda}}\right)$

12. **stop** and **return** S.

13. $add(e,S)$ // add $e.nOccur$ instances of $e.A_i$ to S at $e.nPos$ positions

14. **end while**

15. **return** S.

Procedure: *initPool*

Input: Result-set R and trade-off parameter λ.

Output: The $pool$ of candidate attributes to add to snippets

16. **foreach** $A_i \in attributes(R)$

17. $nOccur \leftarrow numResults(A_i, R)$

18. $e = < A_i, nOccur, 1, \frac{Inform(A_i)^{\lambda}}{Compreh(A_i)^{1-\lambda}} >$

19. $pool.add(e)$

20. **endfor**

Procedure: *recomputePool*

Input: A $pool$ of candidates, the partial result-set snippet S.

Output: The $pool$ of candidate attributes to add to snippets

21. **foreach** $e \in pool$
22. **if** $\left(incompatible(e)\right)$
23. **while** $\left(incompatible(e)\right)$
24. **Alternately** $e.nPos$ ++ or $e.nOccur$--
25. $e = < A_i, nOccur, nPos, \frac{Inform(A)^{\lambda}}{Compreh(A)^{1-\lambda}} >$
26. **endfor**

Figure 5.4. Snippet Construction Algorithm.

The algorithm also maintains the *global* informativeness and comprehension cost of the partially constructed result-set snippet S . It is possible that adding an attribute would decrease the overall goodness of a snippet. For example, if the attribute being added has a very low informativeness and it is being added to a many different positions, then the overall informativeness of the snippet can potentially decrease. To avoid this, the algorithm checks (lines 11 & 12) to see if adding the attribute would decrease the overall score. If the global score decreases, not only is the attribute not added, but the computation stops since any attribute that is added in future would not increase the score.

The adjustment (line 23) works by increasing the number of positions $(e.nPos)$ or decreasing number of results that it can placed in $(e.nOccur)$. For example, in the case of A_5 above, the algorithm might decide to place A_5 in the snippet of only one of the result (2nd or 4th), depending on availability or it might choose to place them in two different positions (columns).The

algorithm prioritizes informativeness over comprehension cost, therefore attempts to place the attribute in multiple positions (by increasing $nPos$) before decreasing informativeness. The score of the attribute is recomputed (line 25) which might result in a change of position in the score ordered $pool$.

To check if a given configuration (entry) of an attribute is compatible, the $incompatible(e, S)$ method (not shown in Figure 5.4) scans the positions (columns) of the partially constructed snippet $\big(S(R, k)\big)$ in the decreasing order of number of cells available for coverage of the attribute in results. For example, if an attribute A6 appears in 10 of (say) 15 results $(\text{i.e.}\, e_6.nOccur = 10)$, and the configuration dictates that the attribute should be placed in 2 positions $(e_6.nPos = 2)$, then the $incompatible(e_6, S)$ method looks for two positions (columns) with empty cells that would cover 10 occurrences by scanning positions (columns) in the snippet matrix in decreasing order of number of empty cells. The rationale for doing so is to fit the given attribute in the minimum number of possible positions.

Complexity: Let N_a be the number of attributes in R. The worst case complexity of the *SnippetsConstructionAlgorithm* is $O(kN_a^2)$ since the algorithm (lines 3-14) considers a $pool$ of N_a attributes to be placed in k positions, giving a complexity of kN_a. Furthermore, in each iteration the $pool$ can be recomputed, to give the overall (worst-case) running time. However, this worst-case running time is misleading the pool is rarely recomputed. We further employ efficient bit-vector manipulation to check for incompatibility making the cost of $incompatible$ negligible.

Table 5.2 Snippet Construction Evaluation Queries

	Query	# Categories	# Attributes
Q1	toshiba laptop	6	39
Q2	dell laptop 3GB	7	43
Q3	dell printers	4	40
Q4	Asus laptops	7	40
Q5	hard drive	7	42
Q6	dell Intel	6	55
Q7	seagate drive	7	40
Q8	dell 13.3	6	51
Q9	hp printer	7	42
Q10	seagate 1TB	4	52

5.7 Experimental Evaluation

In Section 5.7.1, we describe the experiment setup including the datasets used, query workload and the comparison baselines. We present the results of the experiments in Section 5.7.2 and show that snippets constructed using our methods effectively balance comprehension cost and informativeness. We also present the time requirements of our heuristics and shows that they add an insignificant overhead to overall query processing.

5.7.1 Experimental Setup

Dataset and Query Workload: We evaluate our approach on a subset of products data from a popular e-commerce website. The dataset consists of diverse products *types* such as *Computers (Desktops, Laptops etc.)*, *Printers (InkJet, LaserJet etc.)* among many others. In total we extracted 63,126 products from 52 categories (types) and 150 unique attributes distributed

amongst the different types. The queries used in the evaluation are shown in Table 5.2. Table 5.2 also shows some characteristics of the result-set of each query, namely, the number of categories (types) and the total number of unique attributes in query results.

Baselines: We compare our approach with the following two commonly used methods to construct snippets:

Baseline 1 (Fixed-Schema): In this snippet construction method, the schema of the result-set is fixed by choosing the most commonly occurring attributes in results. In a heterogeneous result-set, a result-set snippet would have many empty ($null$) values, since a result may not have a value one or more selected attributes and therefore be less informative. However, these snippets have a low comprehension cost, since all displayed attributes are aligned.

Baseline 2 (Popular-attributes): This method chooses the most informative attributes *for each result*, as its snippet. The informativeness of an attribute is computed based on the result-set using Equation 5.8. The k selected attributes from each result are displayed in a tabular format and are ordered by decreasing informativeness scores. These snippets have are highly informative, since they always select the maximum possible k informative attributes, for each snippet. However, the comprehension cost would be high, due to partial non-alignment of attributes across result snippets and due to inclusion of *superfluous* or unnecessary attributes.

Evaluation Methodology: For each query, we construct snippets of one page of the results, i.e. 15 results, using the algorithm described in Section 5.6 and for the aforementioned baselines. The snippets size k was set to 6 in all our experiments. For snippets constructed using each method, we compute the total comprehension cost and attribute informativeness using Equations 5.4 and 5.8 respectively and report the absolute numbers. We also report the combined

cost $\mathcal{F}$, using Equation 5.2. All experiments were performed by setting $\lambda = 0.5$, i.e. by giving equal weights to informativeness and comprehension factors.

5.7.2 Results

The resulting goodness scores for the snippets of the 10 queries used in the evaluation are shown in Figure 5.5. The goodness scores for our snippet construction algorithm are much better on average as compared to the other two baselines, with an improvement of nearly 27% over Fixed-Schema and 32% over Popular-Attributes approach. Both the Fixed-Schema and Popular-Attributes baseline have similar scores for these queries, since Fixed-Schema approach minimizes comprehension cost while Popular-Attributes maximizes informativeness. Our approach balances both these factors, thereby achieving higher scores.

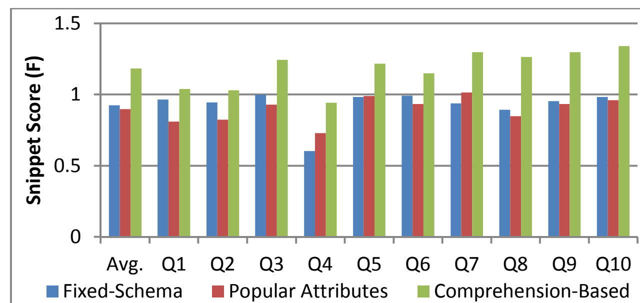


Figure 5.5. Goodness Score ($\mathcal{F}$) of Result-set Snippets

Figure 5.6 shows the overall informativeness of the snippets constructed using the two baselines and our approach. As expected, the informativeness of Fixed-Schema approach is uniformly lower than other two approaches. This is because, by fixing the schema, a number of places in the snippets remain empty (as in Figure 5.1(b)) or less informative attributes get selected. In contrast, the informativeness of snippets constructed by the Popular-Attributes baseline is better than other two approaches since each position in the snippet is occupied by highly informative attributes. For our approach, the informativeness is lower than Popular-Attributes, by about 22%.

This is expected, since we sacrifice informativeness in an effort to make a snippet more comprehensible. By selecting attributes based on comprehension cost, in addition to informativeness, results in selection of some attributes that have low informativeness. Also, some positions in the snippets might remain unoccupied since adding additional attributes might result in degrading of the overall *goodness* score of the snippet.

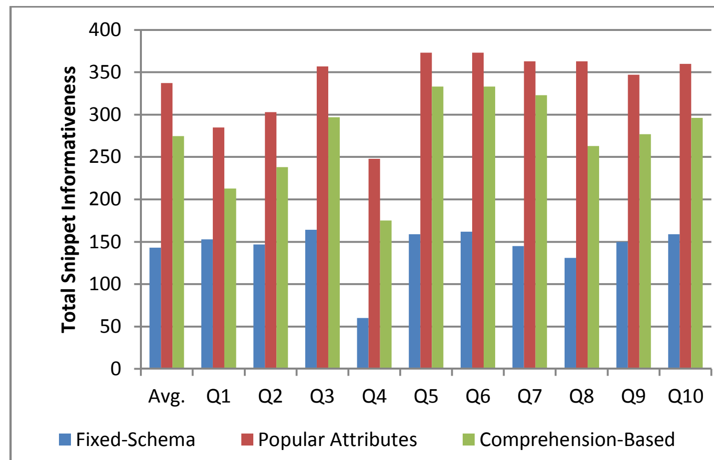


Figure 5.6. Total Informativeness of Attributes in Snippets.

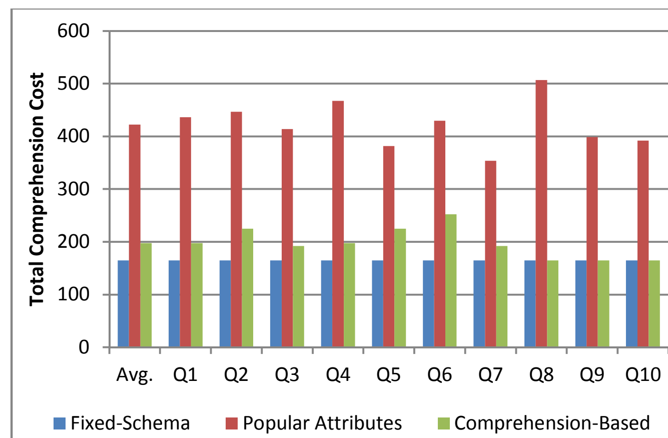


Figure 5.7. Total Result-set Snippet Comprehension Cost.

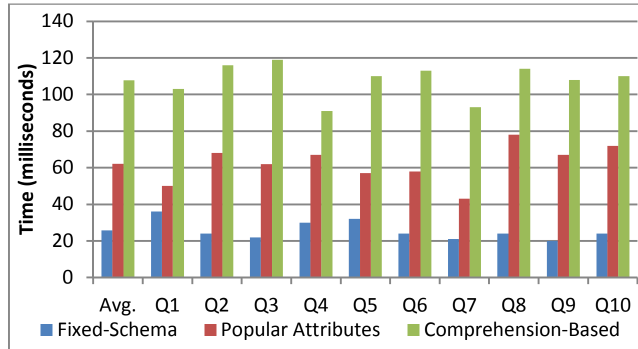


Figure 5.8. Snippet Construction Algorithm Performance.

The loss in informativeness is more than made up (113% improvement, on average) when we consider the comprehension cost, in Figure 5.7, of our snippets to those constructed by the Popular-Attributes approach. The Popular-Attributes baseline does not consider comprehension cost and therefore arranges attributes in *jagged* or non-aligned order across results resulting in a high comprehension cost. Additionally, by selecting attributes independently for each result, the number of attributes that are selected across snippets is high, thereby adding a huge comprehension cost overhead. The snippets constructed using the Fixed-Schema approach have a fixed comprehension cost (Figure 5.7). This is the lowest possible cost, given that all positions in the snippet are occupied. Our approach, which balances comprehension and informativeness, constructs snippets with higher comprehension cost (by 20%) but the resulting snippets are also highly informative (Figure 5.6).

Figure 5.8 compares the execution times of our approach with those of the two baselines. The Fixed-Schema approach, whose schema is fixed beforehand, takes almost negligible amount of time (26ms) on average. Our algorithm takes much longer, 107ms on average, which is higher than Popular-Attributes (avg. 63ms), but is still fast and adds a small overhead to the search and retrieval process since the time to execute the query and retrieve results is considerably higher.

5.8 Related Work

Snippets of text documents: There has been much work on snippets construction for Web documents. Earlier work relied on query-independent document summarization techniques [90, 91]. More recently, query-specific snippet generation techniques have been proposed [97]. However, none of these works consider the problem of comprehension cost across snippets, since each snippet is just a list of sentences.

Faceted Search on Structured Data: Faceted search employed by most e-commerce websites (Amazon, eBay) typically supports predefined top-down navigation on the concept hierarchy, where all the attributes of the currently selected concept are also displayed. Recent works[28, 33] have studied the problem of what attributes to display to minimize the user effort, but operate on flat relations of products without any classification. BioNav[27] presents the bibliographic results of queries on PubMed on an ontological hierarchy and allows users to effectively navigate on that hierarchy. However, attributes are not considered and each publication is viewed as a result hung on a leaf of the hierarchy.

Search Results Comprehension: *Comprehension*, as applied to designing data-driven user interfaces is a subjective measure that falls into the realm of cognitive psychology [98] and Computer Human Interaction (CHI) and focuses on identifying design features of user interfaces and results presentation that maximize the efficiency of user in understanding the set of results or the interface in question. We use the methods and techniques from cognitive psychology and computer human interaction to design user-studies to develop our comprehension cost model and here we discuss works that are related to the goals of our user-study.

Dalal et al. [99] propose a design of website's home pages along several *"theoretical"* guidelines and conduct user studies to measure the comprehension of web-site home pages along three dimensions – accuracy, speed and perceived comprehension and conclude that pages designed with their theorized guidelines achieve better cognition. We also theorize a model of comprehension (Section 5.3) and we use the user study to parameterize the model (define the function $Compreh(.)$). Several recent works[100, 101] study the effect of layout of data elements on their cognition. Our study focuses on measuring the effect of placement of snippet constituents near each other. [100, 101] conduct eye-movement studies of pieces of text to conclude with a direct correlation between placement of target text and comprehension, measured as answers to questions. We follow a similar mechanism, but instead use time spent in reading and answering questions as measure of comprehension.

5.9 Summary

In this chapter, we introduced the problem of incorporating the user comprehension cost into the construction of snippets for structured data. In particular, we defined the framework and identified the criteria for constructing snippets that minimize the effort required by users in comprehending the set of results. We presented a complexity analysis of the problem and efficient algorithms to construct snippets that minimize the comprehension cost, while maximizing the informativeness of the snippets. Our results are supported by user studies and quantitative experiments.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This dissertation focuses on improving user efficiency and experience in data exploration tasks over structured databases. Even though data exploration is a very common activity on the part of everyday users, it has not been adequately addressed in the research community. The current solutions are therefore rife with the use of ad-hoc techniques and methods which often adds to the burden of users in data exploration activities. We carefully scrutinize the existing solutions for data exploration tasks such as query formulation, results navigation and presentation and identify several problems and limitations. We provided holistic solutions to these problems and evaluated them to show that by using our approaches, the burden or effort incurred by users in data exploration tasks is significantly reduced.

In Chapters 3 and 4, we focused on the results navigation aspect of data exploration and presented a *navigation cost* based approach to minimize the user effort in navigating the results. Chapter 3 focuses on the problem commonly encountered by users in searching databases in which results can be classified into a classification or a concept hierarchy. Instead of the static *level-by-level* approach commonly used to navigate the concept hierarchy, we propose a dynamic approach that exposes concept nodes selected from descendants. The nodes are selected based on a navigation cost model of the user and selects nodes based on several factors including the current

state on navigation and distribution of results amongst the concepts. The resulting navigation cost or effort is significantly less compared to standard approaches, as evident from the experimental evaluation. We analyze the complexity of the problem of minimizing the navigation effort and show that the problem is NP-Hard and propose heuristics to compute the *optimal* (in terms of navigation cost) set of nodes in each expansion.

Largely inspired by the performance of our navigation cost based model for concept hierarchies, we extended this approach to faceted navigation, as described in Chapter 4. Here a result (or a database tuple), instead of being classified into one or more concepts, is classified into multiple facets and the problem is to choose the *best* subset of facets and facet conditions such that the user effort in navigating faceted result-sets is minimized. The additional complexity due to multiple classifications warrants a different and more intricate navigation and cost models. The analogous problem of minimizing effort is again NP-Hard and we propose efficient heuristics to select a *near-optimal* set of facets. We experimentally evaluate our approach and show a significant reduction in user-effort using our approach as compared to state-of-the-art. We augment our evaluation with user studies which in addition to showing improvement in navigation effort, also show that our model is realistic and is highly correlated to actual user time.

In Chapter 5, we shift focus on the presentation of search results of structured databases. Typically, a result contains a lot of information (attribute-values) to be displayed entirely on the query interface and only a *snippet* or a subset of fields can be displayed. Whereas existing works have focused on selecting the *best* (defined by an objective score) attribute-values to select to display, in Chapter 5 we consider the auxiliary problem of presenting these attributes. By means of a comprehensive user study, we identify factors that influence the difficulty in reading or

comprehending a snippet and device a simple model to quantify this effort. Further, we presented and evaluated an algorithm to select result-set snippets that balance the score and comprehension.

In Chapter 2, we revisit the keyword-based query formulation paradigm which, due to its inherent simplicity and ease of use, is commonly used in structured databases. However, keyword queries are typically ambiguous and return a large set of results, many of which are irrelevant to the user. We proposed a novel approach in which a keyword query is augmented by adding structured conditions to focus the keyword query to results that the user might be interested in. We presented the design, implementation and evaluation of the framework to support such rich queries.

6.2 Future Work

The query formulation framework suggested in Chapter 2 is based solely on the data-distribution of the underlying dataset and could be augmented with user preferences gleaned from query logs. In this framework, we only consider a specific type of dataset (see Tree Data Model). However many datasets are organized as Directed Acyclic Graphs (DAGs). While the query formulation on such datasets would benefit from the ideas described in the chapter (HCDs), it is not straightforward to apply the algorithms and summarization techniques described to these scenarios and adapting the provided solutions is in itself a very interesting problem.

The problems addressed in this dissertation span a wide variety of components of a query interface. Most of our solutions are built around a model of user, which is inherently difficult to capture due to the existence of a large number of factors. The models we have proposed capture a subset of these factors and can be augmented with additional factors. For example, in Chapter 5 we considered two factors (#occurrences and #positions of attributes) effecting comprehension cost or effort and ignored such factors as complexity of values, the dependence between attributes etc.

These factors were also not considered in the navigation models of Chapters 3 and 4. However, it should be noted that additional factors add to the complexity of the model and can result in computationally in-efficient models whose utility in terms of reducing user effort is not guaranteed.

The techniques presented in Chapters 3 and 4 are quite general and can easily be applied to other navigation scenarios. For example, one could easily develop a model to navigate a list of paginated results where the ranked list of results is computed with additional factors such as clustering[74, 76] or diversity[56, 85, 86, 89, 102] and the corresponding cost-minimization problem being a way to compute the best page of results at each step that would minimize the number of pages a user has to navigate. Furthermore, the two techniques could be combined together in a single navigation and cost model for structured datasets that have both facets and a navigation hierarchy.

References

[1] *Amazon.com*. Online: http://www.amazon.com.

[2] *eBay.com*. Online: www.ebay.com.

[3] *Yellow Pages*. Online: http://www.yellowpages.com/.

[4] *Yelp*. Online: www.yelp.com.

[5] Maglott, D., Ostell, J., Pruitt, K. D. and Tatusova, T. Entrez Gene: Gene-Centered Information at NCBI. *Nucleic Acids Research*, Vol. 33. 2005,pp. D54-D58

[6] *OMIM—Online Mendelian Inheritance in Man*. Online: http://www.ncbi.nlm.nih.gov/Omim/, 2008.

[7] *DBLP*. Online: http://www.informatik.uni-trier.de/~ley/db/.

[8] *PubMed*. Online: http://www.ncbi.nlm.nih.gov/pubmed/.

[9] Jagadish, H. V., Chapman, A., Elkiss, A., Jayapandian, M., Li, Y., Nandi, A. and Yu, C. Making database systems usable. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (Beijing, China, 2007).

[10] Jayapandian, M. and Jagadish, H. V. Automating the Design and Construction of Query Forms. In *Proceedings of the 22nd International Conference on Data Engineering* (2006).

[11] Jayapandian, M. and Jagadish, H. V. Automated creation of a forms-based database query interface. *Proc. VLDB Endow.*, Vol. 1,1. 2008,pp. 695-709

[12] *Lucene*. Online: http://lucene.apache.org.

[13] Aditya, B., Bhalotia, G., Chakrabarti, S., Hulgeri, A., Nakhe, C., Parag, P. and Sudarshan, S. BANKS: browsing and keyword searching in relational databases. In *Proceedings of the 28th international conference on Very Large Data Bases* (Hong Kong, China, 2002).

[14] Hristidis, V. and Papakonstantinou, Y. Discover: keyword search in relational databases. In *Proceedings of the 28th international conference on Very Large Data Bases* (Hong Kong, China, 2002).

[15] Guo, L., Shao, F., Botev, C. and Shanmugasundaram, J. XRANK: ranked keyword search over XML documents. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (San Diego, California, 2003).

[16] He, H., Wang, H., Yang, J. and Yu, P. S. BLINKS: ranked keyword searches on graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (Beijing, China, 2007).

[17] Sadikov, E., Madhavan, J., Wang, L. and Halevy, A. Clustering query refinements by user intent. In *Proceedings of the 19th international conference on World wide web* (Raleigh, North Carolina, USA, 2010).

[18] Sanjay Agrawal, Surajit Chaudhuri, Gautam Das and Gionis, A. Automated Ranking of Database Query Results. In *Proceedings of the First Biennial Conference Innovative Data Systems Research* (2003).

[19] Käki, M. Findex: search result categories help users when document ranking fails. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Portland, Oregon, USA, 2005).

[20] Chaudhuri, S., Das, G., Hristidis, V. and Weikum, G. Probabilistic information retrieval approach for ranking of database query results. *ACM Trans. Database Syst.*, Vol. 31,3. 2006,pp. 1134-1168

[21] Huang, Y., Liu, Z. and Chen, Y. Query biased snippet generation in XML search. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (Vancouver, Canada, 2008).

[22] Liu, Z., Sun, P. and Chen, Y. Structured search result differentiation. *Proc. VLDB Endow.*, Vol. 2,1. 2009,pp. 313-324

[23] Chakrabarti, K., Chaudhuri, S. and Hwang, S.-w. Automatic categorization of query results. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (Paris, France, 2004).

[24] *Amazon Mechanical Turk*. Online: https://www.mturk.com.

[25] Kashyap, A., Hristidis, V., Petropoulos, M. and Tavoulari, S. BioNav: Effective Navigation on Query Results of Biomedical Databases. In *Proceedings of the 2009 IEEE International Conference on Data Engineering* (2009).

[26] Kashyap, A., Hristidis, V., Petropoulos, M. and Tavoulari, S. Exploring biomedical databases with BioNav. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data* (Providence, Rhode Island, USA, 2009).

[27] Kashyap, A., Hristidis, V., Petropoulos, M. and Tavoulari, S. Effective Navigation of Query Results Based on Concept Hierarchies. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 23,4. 2011,pp. 540-553.(c) 2011 IEEE.Reprinted, with permission, from [Abhijith Kashyap, Vagelis Hristidis,Michalis Petropoulos, Sotiria Tavoulari. Effective Navigation of Query Results Based on Concept Hierarchies,Transactions on Knowledge and Data Engineering, April/2011]

[28] Kashyap, A., Hristidis, V. and Petropoulos, M. FACeTOR: cost-driven exploration of faceted query results. In *Proceedings of the 19th ACM international conference on Information and knowledge management* (Toronto, ON, Canada, 2010).

[29] Kashyap, A. and Hristidis, V. Comprehension-based result snippets. In *Proceedings of the 21st ACM international conference on Information and knowledge management* (Maui, Hawaii, USA, 2012).

[30] Xu, Y. and Papakonstantinou, Y. Efficient keyword search for smallest LCAs in XML databases. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data* (Baltimore, Maryland, 2005).

[31] *Transinsight GmbH—GoPubMed,* . Online: http://gopubmed.org, 2008.

[32] Frasincar, F., Telea, A. and Houben, G.-J. *Adapting Graph Visualization Techniques for the Visualization of RDF Data*. In Visualizing the Semantic Web, Springer London, 2006.

[33] Roy, S. B., Wang, H., Das, G., Nambiar, U. and Mohania, M. Minimum-effort driven dynamic faceted search in structured databases. In *Proceedings of the 17th ACM conference on Information and knowledge management* (Napa Valley, California, USA, 2008).

[34] Dietz, P. F. Maintaining order in a linked list. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing* (San Francisco, California, USA, 1982).

[35] Fagin, R., Lotem, A. and Naor, M. Optimal aggregation algorithms for middleware. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (Santa Barbara, California, USA, 2001).

[36] Polyzotis, N. and Garofalakis, M. XCluster Synopses for Structured XML Content. In *Proceedings of the 22nd International Conference on Data Engineering* (2006).

[37] Spiegel, J. and Polyzotis, N. TuG synopses for approximate query answering. *ACM Transactions on Database Systems*, Vol. 34,1. 2009,pp. 1-56

[38] Fagin, R., Kumar, R. and Sivakumar, D. Comparing top k lists. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms* (Baltimore, Maryland, 2003).

[39] Sanjay, A., Chaudhuri, S. and Das, G. *DBXplorer: a system for keyword-based search over relational databases*. 2002.

[40] Hristidis, V., Gravano, L. and Papakonstantinou, Y. Efficient IR-style keyword search over relational databases. In *Proceedings of the 29th international conference on Very large data bases - Volume 29* (Berlin, Germany, 2003).

[41] Balmin, A., Hristidis, V. and Papakonstantinou, Y. Objectrank: authority-based keyword search in databases. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30* (Toronto, Canada, 2004).

[42] Cohen, S., Mamou, J., Kanza, Y. and Sagiv, Y. XSEarch: a semantic search engine for XML. In *Proceedings of the 29th international conference on Very large data bases - Volume 29* (Berlin, Germany, 2003).

[43] Diederich, J. and Balke, W.-T. *The Semantic GrowBag Algorithm: Automatically Deriving Categorization Systems*.Research and Advanced Technology for Digital Libraries. Springer Berlin Heidelberg. 2007. pp.1-13

[44] Sarkas, N., Bansal, N., Das, G. and Koudas, N. Measure-driven keyword-query expansion. *Proc. VLDB Endow.*, Vol. 2,1. 2009,pp. 121-132

[45] Li, X., Wang, Y.-Y. and Acero, A. Extracting structured information from user queries with semi-supervised conditional random fields. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval* (Boston, MA, USA, 2009).

[46] Sarkas, N., Paparizos, S. and Tsaparas, P. Structured annotations of web queries. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (Indianapolis, Indiana, USA, 2010).

[47] Li, Y., Yu, C. and Jagadish, H. V. Schema-free XQuery. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30* (Toronto, Canada, 2004).

[48] Bast, H. and Weber, I. Type less, find more: fast autocompletion search with a succinct index. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval* (Seattle, Washington, USA, 2006).

[49] Nandi, A. and Jagadish, H. V. Effective phrase prediction. In *Proceedings of the 33rd international conference on Very large data bases* (Vienna, Austria, 2007).

[50] Nandi, A. and Jagadish, H. V. Assisted querying using instant-response interfaces. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (Beijing, China, 2007).

[51] Ioannidis, Y. The history of histograms (abridged). In *Proceedings of the 29th international conference on Very large data bases - Volume 29* (Berlin, Germany, 2003).

[52] Aboulnaga, A., Alameldeen, A. R. and Naughton, J. F. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *Proceedings of the 27th International Conference on Very Large Data Bases* (2001).

[53] Polyzotis, N., Garofalakis, M. and Ioannidis, Y. Approximate XML query answers. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (Paris, France, 2004).

[54] Polyzotis, N. and Garofalakis, M. XSKETCH synopses for XML data graphs. *ACM Transactions on Database Systems*, Vol. 31,3. 2006,pp. 1014-1063

[55] Shatkay, H. and Feldman, R. Mining the Biomedical Literature in the Genomic Era: An Overview. *Journal of Computational Biology*, Vol. 10,6. 2003,pp. 821-855

[56] Chen, Z. and Li, T. Addressing diverse user preferences in SQL-query-result navigation. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (Beijing, China, 2007).

[57] Lin, J. and Wilbur, J. W. PubMed related articles: a probabilistic topic-based model for content similarity. *BMC Bioinformatics*, Vol. 8,1. 2007,pp. 1-14

[58] *Medical Subject Headings (MeSH)*. Online: http: //www.nlm.nih.gov/mesh, 2010.

[59] Delfs, R., Doms, A., Kozlenkov, E. and Schroeder, M. GoPubMed: ontology-based literature search applied to GeneOntology and PubMed. In *In Proceedings of German Bioinformatics Conference. LNBI* (2004).

[60] *Vivisimo, Inc.—Clusty*. Online: http://clusty.com/, 2008.

[61] *Entrez Programming Utilities*. Online: http://www.ncbi.nlm.nih.gov/books/NBK25500/, 2008.

[62] Feige, U., Kortsarz, G. and Peleg, D. The Dense k-Subgraph Problem. *Algorithmica*, Vol. 29. 1999,pp. 2001

[63] Kundu, S. and Misra, J. A Linear Tree Partitioning Algorithm. *SIAM Journal on Computing*, Vol. 6,1. 1977,pp. 151-154

[64] *PubMed PubReMiner: A Tool for PubMed Query Building and Literature Mining*. Online: http://bioinfo.amc.uva.nl/human-genetics/pubreminer, 2008.

[65] Perez-Iratxeta, C., Bork, P. and Andrade, M. A. XplorMed: a tool for exploring MEDLINE abstracts. *Trends Biochem Sci. 2001 Sep;26(9):573-5.*,- 0968-0004 (Print). 2001

[66] *XplorMed: eXploring Medline abstracts*. Online: http://www.ogic.ca/projects/xplormed/, 2008.

[67] Plake, C., Schiemann, T., Pankalla, M., Hakenberg, J. and Leser, U. AliBaba: PubMed as a graph. *Bioinformatics. 2006 Oct 1;22(19):2444-5. Epub 2006 Jul 26.*,- 1367-4811 (Electronic). 2006

[68] Hoffmann, R. and Valencia, A. A gene network for navigating the literature. *Nature genetics*, Vol. 36,7. 2004

[69] *iHOP—Information Hyperlinked over Protein*. Online: http://www.ihop-net.org/UniPub/iHOP, 2008.

[70] Lee, W.-J., Raschid, L., Sayyadi, H. and Srinivasan, P. Exploiting Ontology Structure and Patterns of Annotation to Mine Significant Associations between Pairs of Controlled Vocabulary Terms. In *Proceedings of the 5th international workshop on Data Integration in the Life Sciences* (Evry, France, 2008).

[71] Zhang, T., Ramakrishnan, R. and Livny, M. BIRCH: an efficient data clustering method for very large databases. *SIGMOD Rec.*, Vol. 25,2. 1996,pp. 103-114

[72] *Stanford Univ.—HighWire Press*. Online: http://highwire.stanford.edu, 2008.

[73] *PubMatrix: A Tool for Multiplex Literature Mining*. Online: http://pubmatrix.grc.nia.nih.gov/, 2003.

[74] Demner-Fushman, D. and Lin, J. Answer extraction, semantic clustering, and extractive summarization for clinical question answering. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics* (Sydney, Australia, 2006).

[75] Bodenreider, O. The Unified Medical Language System (UMLS): integrating biomedical terminology. *Nucleic Acids Research*, Vol. 32,suppl 1. 2004,pp. D267-D270

[76] Hearst, M. A. Clustering versus faceted categories for information exploration. *Communications of ACM*, Vol. 49,4. 2006,pp. 59-61

[77] Yee, K.-P., Swearingen, K., Li, K. and Hearst, M. Faceted metadata for image search and browsing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Ft. Lauderdale, Florida, USA, 2003).

[78] Hearst, M. A. *User Interfaces and Visualization. Modern Information Retrieval*.Ricardo Baeza-Yates and Berthier Ribeiro-Neto, Eds. ACM Press. 1999. pp.257–323

[79] Chvatal, V. A Greedy Heuristic for the Set Cover Problem. *Mathematics of Operations Research*, Vol. 4,3. 1979,pp. 233-235

[80] Hristidis, V., Hwang, H. and Papakonstantinou, Y. Authority-based keyword search in databases. *ACM Transactions on Database Systems*, Vol. 33,1. 2008,pp. 1-40

[81] English, J., Hearst, M., Sinha, R., Swearingen, K. and Yee, K.-P. Hierarchical faceted metadata in site search interfaces. In *CHI '02 Extended Abstracts on Human Factors in Computing Systems* (Minneapolis, Minnesota, USA, 2002).

[82] Vélez, B., Weiss, R., Sheldon, M. A. and Gifford, D. K. Fast and effective query refinement. *SIGIR Forum*, Vol. 31,SI. 1997,pp. 6-15

[83] Ortega-Binderberger, M., Chakrabarti, K. and Mehrotra, S. An Approach to Integrating Query Refinement in SQL. In *Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology* (2002).

[84] Wu, P., Sismanis, Y. and Reinwald, B. Towards keyword-driven analytical processing. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (Beijing, China, 2007).

[85] Smyth, B. and McClave, P. Similarity vs. Diversity. In *Proceedings of the 4th International Conference on Case-Based Reasoning: Case-Based Reasoning Research and Development* (2001).

[86] Vieira, M. R., Razente, H. L., Barioni, M. C. N., Hadjieleftheriou, M., Srivastava, D., Traina, C. and Tsotras, V. J. On query result diversification. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering* (2011).

[87] Schuirmann, D. A comparison of the Two One-Sided Tests Procedure and the Power Approach for assessing the equivalence of average bioavailability. *Journal of Pharmacokinetics and Biopharmaceutics*, Vol. 15,6. 1987,pp. 657-680

[88] Clarke, C. L. A., Kolla, M., Cormack, G. V., Vechtomova, O., Ashkan, A., Buttcher, S. and MacKinnon, I. Novelty and diversity in information retrieval evaluation. In *Proceedings of*

the 31st annual international ACM SIGIR conference on Research and development in information retrieval (Singapore, Singapore, 2008).

[89] Agrawal, R., Gollapudi, S., Halverson, A. and Ieong, S. Diversifying search results. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining* (Barcelona, Spain, 2009).

[90] White, R. W., Ruthven, I. and Jose, J. M. Finding relevant documents using top ranking sentences: an evaluation of two alternative schemes. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval* (Tampere, Finland, 2002).

[91] Turpin, A., Tsegay, Y., Hawking, D. and Williams, H. E. Fast generation of result snippets in web search. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval* (Amsterdam, The Netherlands, 2007).

[92] Chen, H. and Karger, D. R. Less is more: probabilistic models for retrieving fewer relevant documents. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval* (Seattle, Washington, USA, 2006).

[93] Roy, S. B., Amer-Yahia, S., Chawla, A., Das, G. and Yu, C. Constructing and exploring composite items. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (Indianapolis, Indiana, USA, 2010).

[94] Jain, A., Sarda, P. and Haritsa, J. *Providing Diversity in K-Nearest Neighbor Query Results*.Advances in Knowledge Discovery and Data Mining. Springer Berlin Heidelberg. 2004. pp.404-413

[95] Carterette, B. *An Analysis of NP-Completeness in Novelty and Diversity Ranking*.Advances in Information Retrieval Theory. Springer Berlin Heidelberg. 2009. pp.200-211

[96] Das, G., Hristidis, V., Kapoor, N. and Sudarshan, S. Ordering the attributes of query results. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (Chicago, IL, USA, 2006).

[97] Varadarajan, R. and Hristidis, V. A system for query-specific document summarization. In *Proceedings of the 15th ACM international conference on Information and knowledge management* (Arlington, Virginia, USA, 2006).

[98] Kitajima, M., Blackmon, M. and Polson, P. *A Comprehension-based Model of Web Navigation and Its Application to Web Usability Analysis*.People and Computers XIV — Usability or Else! Springer London. 2000. pp.357-373

[99] Dalal, N. P., Quible, Z. and Wyatt, K. Cognitive design of home pages: an experimental study of comprehension on the World Wide Web. *Information Processing and Management*, Vol. 36,4. 2000,pp. 607-621

[100] Bicknell, K. and Levy, R. Rational eye movements in reading combining uncertainty about previous words with contextual probability. In *Proceedings of the 32nd Annual Conference of the Cognitive Science Society* (2010).

[101] Dubey, A., Keller, F. and Sturt, P. The Effect of Phonological Parallelism in Coordination: Evidence from Eye-tracking. In *Proceedings of the 2nd European Cognitive Science Conference,* (2007).

[102] Radlinski, F. and Dumais, S. Improving personalized web search using result diversification. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval* (Seattle, Washington, USA, 2006).