Effective Faceted Browsing

by

Manish Singh

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2014

Doctoral Committee:

Professor Hosagrahar V. Jagadish, Chair
Assistant Professor Michael J. Cafarella
Professor Paul J. Resnick
Associate Professor Soo Young Rieh

© Manish Singh 2014

All Rights Reserved

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Prof. H. V. Jagadish, for his invaluable guidance, continuous encouragement and wholehearted support in every stage of my doctorate study. It was a privilege to work under his supervision. I always felt amazed to see his research expertise. Like an expert researcher, he could solve problems that I would be struggling for days and months in just a few minutes. I am very grateful to him for generously giving me so much time to teach various aspects of research and communication skills. He gave me ample opportunities to be creative and to pursue my thoughts with complete freedom. Although the PhD process is a long and arduous journey with many ups and downs, he made it one of my most joyful experiences. I sincerely appreciate his genuine care and generous help in all aspects of my life.

I am also extremely grateful to my dissertation committee members, Prof. Michael Cafarella, Prof. Soo Young Rieh and Prof. Paul Resnick, for spending their valuable time on my dissertation. Their insightful comments and guidance have greatly improved the quality of my dissertation.

I would like to express my gratitude to other professors at UM from whom I learned a lot about different styles of research and teaching: Prof. Barzan Mozafari, Prof. Atul Prakash, Prof. Kevin Compton, and Prof. Kristen LeFevre. It is a great honor to be a part of the Michigan database group. I am very thankful to all my past and current database group friends: Michael Anderson, Yongjoo Park, Dolan Antenucci, Rajesh Bejugam, Matt Burgess, Shirley Zhe Chen, Daniel Fabbri, Lujun

Fang, Fernando Farfan, Fei Li, Bin Liu, Arnab Nandi, Eric Li Qian, Anna Shaverdian, and Jing Zhang, for providing me lot of valuable support. And I would also like to thank my other friends at UM: Ramasubramanian Krishnamurthy, Krishna C. Garikipati, Beng Heng, Alex Crowell and Adam Patterson.

I am thankful to my many friends and teachers at Indian Institute of Technology, Delhi, for giving all the inspiration to pursue this research in computer science. It is by their support that I have got this unique opportunity to do my PhD at the University of Michigan. I am also very thankful to my parents, grandmother and other family members for their support and patience during this work.

Finally, and most importantly, I would like to thank Supreme Lord for giving me all the intelligence to do this work. I hope I can use this gift properly for the service of mankind.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii

LIST OF FIGURES vii

LIST OF TABLES xi

ABSTRACT xii

CHAPTER

I. Introduction 1

1.1 Benefits of Faceted Navigation 1
1.2 Limitations of Faceted Navigation 3
1.2.1 Limited Feedback 3
1.2.2 Low Adaptivity 4
1.2.3 Hidden Facets 5
1.3 Challenges 6
1.4 Approaches and Contributions 7
1.5 Dissertation Organization 9

II. Background and Interaction Model 10

III. Querying High-Dimensional Data Using Faceted Interface . . 15

3.1 Problem Overview 15
3.2 Limitations in the Query Panel 19
3.3 Implicit Choices Summary 20
3.3.1 The ICSummary 21
3.3.2 Algorithmic Challenges 26
3.4 Algorithms 29
3.4.1 Creating ICSummary 29
3.4.2 Selecting Top-K Queriable Facets 34

3.5 Efficient Implementation of ICSummary 36
3.5.1 Faceted Search Implementation 36
3.5.2 ICSummary Using Additional Queries 38
3.5.3 ICSummary Using Extended Summary Digest . . . 40
3.6 Evaluation 41
3.6.1 Experimental Setup 42
3.6.2 Performance 42
3.7 Conclusion 46

IV. TPFacet: Two-Phased Faceted Browsing 47

4.1 Problem Overview 47
4.2 Limitations in the Query Panel 50
4.3 Summarized Facet Interaction 52
4.3.1 The SFI View 54
4.3.2 Algorithmic Challenges 58
4.4 The Static SFI View 62
4.4.1 Generating Candidate IUnits 62
4.4.2 Finding Important Facet Interactions 64
4.4.3 Selecting Top-k IUnits 66
4.5 The Dynamic SFI View 67
4.5.1 Highlighting Similar IUnits 67
4.5.2 Finding Similar Facet Values 69
4.6 Evaluation 70
4.6.1 Experimental Setup 71
4.6.2 Quantitative Analysis of User Study 72
4.6.3 Qualitative Analysis of User Study 79
4.6.4 Performance 85
4.7 Conclusion 89

V. Skimming through Relational Query Result 90

5.1 Problem Overview 90
5.2 Paging Interface with Scrolling 94
5.2.1 Problem Definition 94
5.2.2 User Interface 95
5.2.3 Goodness Measure 96
5.3 Algorithms 99
5.3.1 Naïve Sampling 100
5.3.2 K-medoids Based Sampling 100
5.3.3 K-means Based Sampling 104
5.3.4 Truncating History for Fast Performance 109
5.4 Evaluation 110
5.4.1 Experimental Setup 110
5.4.2 Performance 111

5.4.3 User Study 118
5.5 Conclusion 125

VI. Related Work 126

6.1 Faceted Search 126
6.2 Lookup vs. Exploratory Search 127
6.3 Parallel Coordinates 128
6.4 Interaction Between Attributes 131
6.5 Query Formulation 131
6.6 Search Result Presentation 132
6.7 Information Visualization 134

VII. Conclusion and Future Work 136

7.1 Contributions 136
7.2 Future Work 138

BIBLIOGRAPHY 140

LIST OF FIGURES

Figure

1.1 This screen capture from `cars.com` represents a typical example of a faceted navigation interface. Like most such interfaces, it is divided between a small *query panel* on the left and a *results panel* on the right. The left-hand side also includes a small *summary digest* describing the overall result set. 2

2.1 This screen capture from `amazon.com` shows many add-on extensions that are seen in typical faceted interface. 11

3.1 ICSummary for `BodyType` = `Sedan`, where the user has selected cars with `Year` $\geq$ 2012. For each dependent facet we show its chi-square and p-value in parenthesis. With each facet value we show its observed count / expected count information in parenthesis. The font colors green, blue and red indicates increased, unaltered and decreased likelihood respectively. 23

3.2 ICSummary for `BodyType` = `SUV`, where the user has selected cars with `Year` $\geq$ 2012. For each dependent facet we show its chi-square and p-value in parenthesis. With each facet value we show its observed count / expected count information in parenthesis. The font colors green, blue and red indicates increased, unaltered and decreased likelihood respectively. 24

3.3 Conditional dependency between attributes of a used-car dataset represented in the form of Bayesian network. 35

3.4 Time breakdown for static computations in the YahooUsedCar dataset, which is a small dataset with 11 attributes. 43

3.5 Time breakdown for static computations in the Census dataset, which is a large dataset with 68 attributes. Effect of optimizations on computing the extended summary digest. 44

4.1 This screen capture shows a sample SFI View shown during query revision phase. It shows the top-4 IUnits for each of the five body types the user has selected in the Pivot Facet BodyType. The user has also selected Year $\geq$ 2012 and Transmission = Automatic. . . . 53

4.2 In this task users had to build a binary class classifier. Statistical analysis shows that TPFacet affects the quality of classfier by ($\chi^2(1) = 5.572,\ p = 0.018$), increasing the F1 score by about 0.078 ± 0.0285. 74

4.3 In this task users had to build a binary class classifier. Statistical analysis shows that TPFacet affects the time taken by ($\chi^2(1) = 8.54,\ p = 0.003$), lowering it by about 5.44 ± 1.56 minutes. 74

4.4 In this task users had to find the most similar facet value pair from a given list of four categorical facet values. Statistical analysis shows that there is no significant difference in users response quality by using the two types of interfaces. 76

4.5 In this task users had to find the most similar facet value pair from a given list of four categorical facet values. Statistical analysis shows that TPFacet affects the time taken by ($\chi^2(1) = 12.04,\ p = 0.0005$), lowering it by about 6.00 ± 1.23 minutes. 76

4.6 In this task users had to find an alternate search condition that would lead to the same result set as the given search condition. Statistical analysis shows that TPFacet affects the users alternative search condition by ($\chi^2(1) = 3.28,\ p = 0.07$), lowering the retrieval error by about 0.329 ± 0.172. 78

4.7 In this task users had to find an alternate search condition that would lead to the same result set as the given search condition. Statistical analysis shows that TPFacet affects the time taken by ($\chi^2(1) = 2.58,\ p = 0.108$), lowering it by about 2.00 ± 1.14 minutes. 79

4.8 This screen capture shows a sample SFI View that a user had seen during the observational study. It shows the top-4 IUnits for each of the four price ranges the user had selected in the Pivot Facet Price. The user had also selected Make = Chevrolet, BodyType = Sedan and Transmission = Automatic. 80

4.9 This screen capture shows a sample SFI View that a user had seen during the observational study. It shows the top-4 IUnits for each of the four manufacturers the user had selected in the Pivot Facet Make. The user had also selected BodyType = Hatchback and Transmission = Automatic. 80

4.10 Time breakdown to compute SFI View for different result set sizes. Here the system parameters are chosen to demonstrate the worst-case performance of our system. 86

4.11 System performance with varying number of generated candidate IUnits. 88

4.12 System performance with varying number of informative facets. . . 88

5.1 *User interface:* Results are displayed in a paginated interface, browsable using the scrollbar. The scroll position determines the currently displayed page. Instead of overwhelming the user with a full display where all tuples are shown, the user is presented with a condensed display featuring a set of representative tuples. These tuples are selected from the current page based on contents of the page itself, the history of pages seen so far, the scroll position and the speed of scrolling. 95

5.2 Comparison between computing representative tuples without considering user's browsing history vs. considering browsing history. . . 99

5.3 Effect of initial cluster centers on the two history based K-means sampling algorithm. 108

5.4 Computational time of all seven sampling algorithms with varying system parameters such as page size, number of dimension and sampling rate. 112

5.5 Information quality of all seven sampling algorithms with varying system parameters such as page size, number of dimension and sampling rate. 115

5.6 Effect of the user's browsing history size on computation time. . . . 116

5.7 Effect of the user's browsing history size on information quality. . . 117

5.8 In this task users had to find the two pages with minimum and maximum information variation in stock prices. Statistical analysis shows that users response quality is almost similar for both the interfaces. But condensed display affects the time taken by ($\chi^2(1) = 8.78,\ p = 0.003$), lowering it by about 272.5 ± 68.2 seconds. 120

5.9 In this task users had to predict a missing attribute value for a given tuple using manual regression. Statistical analysis shows that users response quality is almost similar for both the interfaces. But condensed display affects the time taken by ($\chi^2(1) = 6.18,\ p = 0.013$), lowering it by about 125 ± 40.9 seconds. 121

5.10 In this task users had to find the maximum discriminating column for data classification. Statistical analysis shows that time taken is almost similar for both the interfaces. But condensed display affects the users' ability to find the most discriminating column by ($\chi^2(1) = 5.31,\ p = 0.021$), increasing the quality by about 1.13 ± 0.45 rank. . 123

5.11 Relationship between users average scrolling speed and suitable sampling rate 124

6.1 Parallel coordinates with matching. 128

6.2 Parallel coordinates with aggregation. 130

LIST OF TABLES

Table

3.1 Sample tuples from a used-car dataset 17

3.2 Sample contingency table for attributes `BodyType` and `Drivetrain` . 38

3.3 Time breakdown for dynamic computations (or creating an ICSummary). 45

4.1 Sample informative facets for used-car dataset. 63

ABSTRACT

Effective Faceted Browsing

by

Manish Singh

Chair: Hosagrahar V. Jagadish

Faceted browsing is a popular paradigm for end-user data access. It is, at present, the de-facto search interface for almost all e-commerce. A typical faceted interface has two main component panels: a query panel and a result panel. Faceted browsing is primarily designed to help users quickly get to a specific item if they know the characteristics they are looking for. However, limitations in the query and the result panel deter effective faceted browsing, especially for users unfamiliar with the data.

In this dissertation, we highlight two such limitations, one each in the query and the result panel. We propose add-on extensions to address each of these limitations. In a faceted interface, users progressively select a sequence of facet values to get to their desired result set, which is called an exploration path. If the dataset is high-dimensional, the query panel can only show a few of those dimensions as queriable facets. Users cannot see in the query panel the overall space of available exploration paths, and thus end up choosing an inferior exploration path. Many users have difficulty in selecting or understanding an exploration path when there are many non-queriable facets and the query panel has very limited information of interaction between facets. We address this limitation by showing users an integrated summary

of facet interaction that summarizes their chosen exploration path, and by presenting a two-phased faceted interface that provides users a facetwise way to compare the available exploration paths. The result panel that is normally used for presenting relational tuples, including faceted interface, cannot support fast browsing. When a user scrolls fast through data having alphanumeric values, then everything seems like a fast changing blur. To help the user get a quick sense of data, we propose a novel variable-speed scrolling interface, which provides the user a good impression of the data through selected representative tuples that are chosen based on the user's scrolling speed and browsing history.

CHAPTER I

Introduction

1.1 Benefits of Faceted Navigation

Many e-commerce sites struggle to present their data to users in an easily accessible manner, especially when the users have limited knowledge of what is contained in their database or lack technical expertise to form proper queries. *Faceted navigation* is a central tool that these e-commerce sites use to address this challenge. Faceted navigation is currently the *de-facto* search interface for almost all e-commerce sites.

Figure 1.1 from `cars.com` [2] is a screenshot of a typical faceted search interface for browsing a database of car models. Users select interface elements in the left-hand *query panel* to control what database items are shown in the central *result panel*. The query panel user interface elements generally consist of links, simple check boxes or sliders, which are used to place selection criteria on the underlying database. The result panel is scrollable, so that users can examine more results than can be displayed on the screen at once. The query panel may also include a small *summary digest* that captures overall information about the result set, such as the current query and the number of tuples in the set.

Faceted navigation offers a number of advantages over, say, raw SQL queries. For example, it is easy for users to recover when an operation in the query panel inadvertently yields an empty result set (users can simply unclick a check box). The

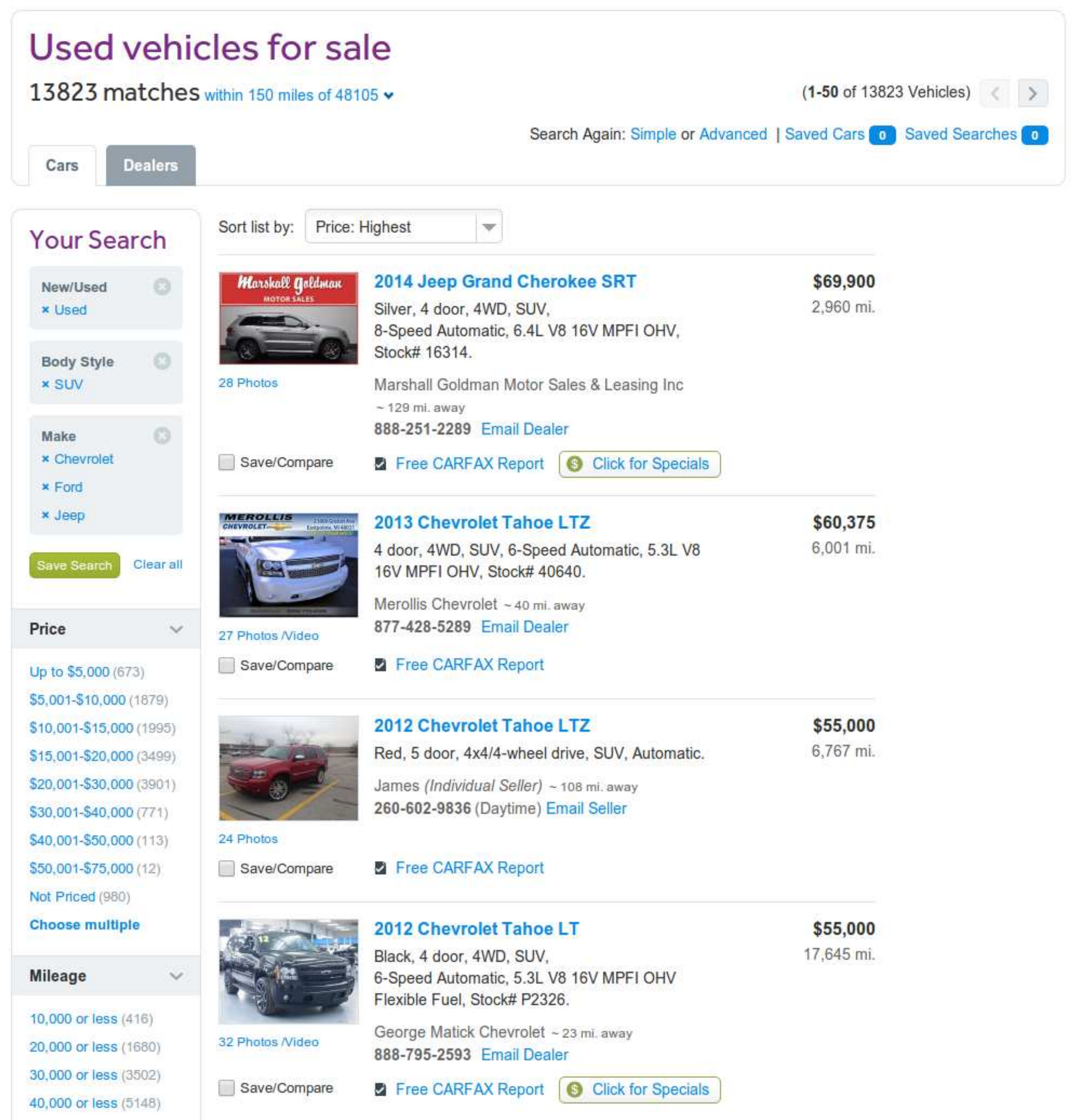


Figure 1.1: This screen capture from `cars.com` represents a typical example of a faceted navigation interface. Like most such interfaces, it is divided between a small *query panel* on the left and a *results panel* on the right. The left-hand side also includes a small *summary digest* describing the overall result set.

converse is also true: it is easy to pose additional selection for drilling down when a search result is too large, and it is easy for domain-knowledgeable users to encode a desired query. Finally, such tools protect users from themselves by always giving results that are meaningfully organized: users are unable to, say, perform a projection that renders the result set incomprehensible. In addition to the above qualities of the interface itself, most back-end databases that support faceted navigation are configured to give extremely fast query results, thereby enabling rapid query iteration.

1.2 Limitations of Faceted Navigation

1.2.1 Limited Feedback

One of the main reasons that faceted navigation has almost universally replaced other forms of information organization is its *self-explorable* quality. It does not restrict users to follow a predetermined navigation order. Users can browse a faceted interface through independent facets based on their specific preference, or domain knowledge. However, giving too much independence, without suitable feedback, also leads to users feeling lost, especially when they are unfamiliar with the data.

While creating a faceted interface, designers want to choose a subset of attributes as facets that are important for most of the underlying data and are also independent. However, this independence assumption rarely holds true in real-life applications. When users make certain choices, it narrows their search space by automatically eliminating many other choices. For example, cars with eight-cylinder engines tend to be large and expensive; someone who chooses the eight-cylinder facet value will not select facet values inexpensive and compact. Instead of providing users useful feedback information, such as information about their exploration direction, or relationship across facets and conditional distributions on facet values, the faceted interface leaves it to the users to manually figure out such information beyond a minimal summary

digest that provides some counts.

A faceted interface devotes the bulk of its display space to show the result panel. Since it is not possible for users to manually look at thousands of result tuples that are shown in the result panel, most users do not really care about the result panel until they near the end of their exploration. Users spend most of their time exploring the summary digest, which is shown within the small left-hand query panel, because that is their only means to understand the space of available options and get to their desired result set.

The tuple counts given by the summary digest can be helpful guidance for the users, but are not nearly enough — they only provide information on the number of matches without characterizing them in any meaningful way.

1.2.2 Low Adaptivity

The traditional faceted interface has a very simple presentation for the query and the result panel. Typically, the query panel shows a subset of attributes from the dataset as facets, which are shown in some fixed order, with each facet showing a list of facet values that are ordered by count or alphabetical order. Similarly, the result panel shows an unorganized list of result tuples, which users can often reorder by sorting along some provided sorting attribute, such as `Price, Year,` etc.

Some recent research work has addressed this low adaptivity aspect. For example, [67, 15] have looked at improving the adaptivity of query panel by ranking facets and facet values to minimize users' navigation cost. Similarly, the result panel can be made more user-adaptive by clustering the results [81, 111] and then allowing users to more easily find similar result tuples by using features such as more-like-this [1].

However, the result panel is not adaptive to allow the users to browse through large result sets at different browsing speeds and still be able to absorb the displayed information. The result panel that is normally used for presenting relational tuples,

including faceted interface, cannot support variable-speed browsing. Faceted search often yields a large set of tuples as query result. If the result comprises tuples of alphanumeric values with few visual markers, it is hard to quickly browse such data, even if the result is sorted. When users try to scroll fast through such alphanumeric data, then everything seems like a fast-changing blur. The result panel can be more effective if it can become aware of users' browsing pattern, and dynamically adjust itself to the users' need. For example, instead of showing users a large number of tuples that they cannot assimilate during fast scrolling, it would be useful to show users fewer tuples that they can assimilate even while fast scrolling. The trick is to choose carefully the tuples to show so as to minimize information loss.

1.2.3 Hidden Facets

Even though faceted interfaces are designed to permit self-exploration by users, limited screen real estate (and user fatigue) often force interface designers to restrict the number of queriable facets that are shown to users in the query panel. For example, `cars.com` does not offer the number of cylinders in the engine as a queriable facet, even though this information is available as a hidden facet (additional feature) for each result tuple.

Designers often choose attributes that are globally important for all the tuples as queriable facets. The remaining attributes (hidden facets) may also be very important, but they may not be applicable for all the tuples or used frequently by all the users.

Following are some important limitations due to the existence of hidden facets:

Limitation 1. Expressing Selection — If users want to perform selection using hidden facets, then existing faceted interface do not provide any support to assist such selection. In this dissertation, we show that if users are shown the relationship between hidden facets and queriable (shown) facets, then users can easily express

their selection in hidden facet(s) in terms of the queriable facets. We help users learn such insightful mappings between facets by providing appropriate feedback in the query panel.

Limitation 2. No Summary Feedback — Faceted interfaces almost ignores all the information present in the hidden facets. The only way users can see the information in hidden facets is by looking at the additional features for each individual tuple. Since many datasets are high-dimensional with number of attributes between 25 to a hundred or more [32] and the faceted interface shows only a few (typically less than 10) of those attributes as queriable facets, it means that users lose almost all the important information that is present in such high-dimensional data. Instead of ignoring all the hidden facets, it would be more useful if the appropriate information from hidden facets can be automatically combined with the queriable facets to provide better insight to users.

1.3 Challenges

Addressing the limitations described above requires overcoming multiple challenges:

Challenge 1. Retaining Basic Faceted Interface — One common major challenge in designing any new add-on extension for faceted interface is to not alter the basic fundamental structure of faceted interface. A faceted interface provides one of the best means to query a database, and a concise and an easily understandable summary of the query result. Extensions that address specific limitations should be designed as add-ons that do not destroy the fundamental structure of faceted interface. For example, there are many limitations of the existing query panel due to limited feedback, hidden facets, etc. Ideally, one would like to address these limitations without introducing much change in the basic query panel structure. Similarly,

the result panel should be improved in such a way that it looks very similar to the existing result panel, but it provides better quality information to users.

Challenge 2. Quality Evaluation — Most of the limitations that are addressed in this dissertation are related to users' difficulty in understanding the information that is presented in the traditional faceted interface. For example, we highlight the difficulties that users face in understanding: the query panel, the result panel and the hidden facets. We claim that although users somehow manage to do things in the existing faceted interface, without even properly understanding what they are doing; they can do their navigation more effectively if we can provide them more informative query and result panel. In all the user studies presented in this dissertation, we had the difficult task to objectively measure users better understanding through the use of our proposed add-on extensions.

Challenge 3. Performance Constraint — Faceted interface is a user-facing application. All user interface applications have strict computational constraint that requires small computation time (< 500 ms) for providing real-time performance. Although many algorithms that are presented in this dissertation are based on computationally intensive data mining algorithms, we have to design systems in such a way that we can provide real-time performance to users.

1.4 Approaches and Contributions

In order to make faceted navigation more effective, this dissertation makes three main contributions. The first two contributions improve the query formulation, and the last contribution improves the result presentation.

Contribution 1. Querying High-Dimensional Data — Due to limited screen real estate and user fatigue, designers can choose only a limited number of attributes from a high-dimensional dataset as queriable facets. The faceted interface does not

allow users to query or get summary digest for the remaining non-queriable (or hidden) facets. Users have difficulty in selecting or understanding an exploration path when there are many non-queriable facets and the query panel has very limited information of interaction between facets. We address this limitation by showing users an integrated summary of facet interaction that summarizes their chosen exploration path. In datasets with dependent attributes, when users explicitly select a value in one facet, they cannot understand the implicit selections that happen in other facets. To help users understand these implicit selections, we present a novel data summarization technique, known as *Implicit Choices Summary* (ICSummary), that can summarize the pairwise interaction of a given attribute value with values in other top-l other related attributes. At present, designers select the queriable facets using ad-hoc strategies or with help of domain experts, and thus have no principled way to measure the information loss that happens due to hidden facets. We use ICSummaries to minimize this information loss. In our proposed faceted interface, when users hover over a queriable facet value, we show them an ICSummary in the form pop-window that provides a summary preview of implicit selections in other facets. Since we can capture important facet interactions through ICSummaries, it gives us additional flexibility to choose the most informative subset of queriable facets.

Contribution 2. Two-Phased Faceted Browsing — A faceted interface allows users to compose a combinatorial number of possible queries, without providing systematic means to compare them, other than through exhaustive examination of large result sets. In consequence, users frequently end up choosing an inferior query specification, and hence miss the best results. We present a two-phased faceted interface, known as TPFacet, which provides a facetwise comparison of the top-k next choices as users incrementally build their query. Our interface relies on a novel Summarized Facet Interaction (SFI) View that provides users a context dependent summary of the data, subject to the choices the users have already made. Through the SFI View

users can see in facetwise manner many interesting exploration options and how those options compare with each other.

Contribution 3. Skimming through Relational Query Result — During faceted navigation users often start their exploration with a large result set (30K-40K tuples) and then gradually reduce it to a final result set of few tuples. Even during the interim stages of faceted navigation, users often look at sample results in the result panel to figure out the next facet value that they would like to select. A faceted interface presents the search result in a pagewise manner, where each page is scrollable. Since the search result often consists of alpha-numeric information, it is very hard for users to scroll fast through such result pages and get a quick sense of the data; rather everything seems like a fast changing blur. To help users get a "good impression" of the data even during rapid scrolling, we present a novel variable-speed scrolling interface, called *Skimmer*, which can provide users a good impression of the data through selected representative tuples.

1.5 Dissertation Organization

The remainder of the dissertation is organized as follows: Chapter II gives a background of faceted navigation with its interaction model. Chapter III discusses a novel faceted interface that is integrated with ICSummaries to allow users query high-dimensional data in a more informed manner. Chapter IV presents the TPFacet system that can assist users gain a better understanding of various exploration paths that are available to them in a facetwise manner. Chapter V presents a variable-speed scrolling interface to support fast browsing over relational result panel. Finally, we present our related work in Chapter VI and conclusion in Chapter VII.

CHAPTER II

Background and Interaction Model

Faceted navigation is a type categorization technique that is used to access a collection of items by their multiple classifications, or facets. It can be used to access both documents and structured databases, though we will primarily focus on its application to relational databases.

As shown in Figure 1.1, a basic faceted interface has two main component panels: a query panel and a result panel. The latter typically occupies the majority of the screen real estate and shows the set of currently selected items. The former is usually on the left (or top) side, and offers both user interface controls as well as a summary digest of the current query and the result set.

Due to the many benefits of faceted interface, many researchers in IR and Databases have worked on improving the usability of basic faceted interface by providing numerous add-on extensions to improve both the query and the result panel. The faceted interface shown in Figure 1.1 is from `cars.com`, which is a major online car shopping company for buying and selling cars. Unlike `cars.com` that only deals with cars, things become much more complicated at sites, such as `amazon.com, ebay.com,` etc., where they manage huge collection of highly heterogeneous items. Although `amazon.com` uses the basic faceted interface, but it is integrated with many useful add-on extensions, as shown in Figure 2.1. Although the basic faceted interface

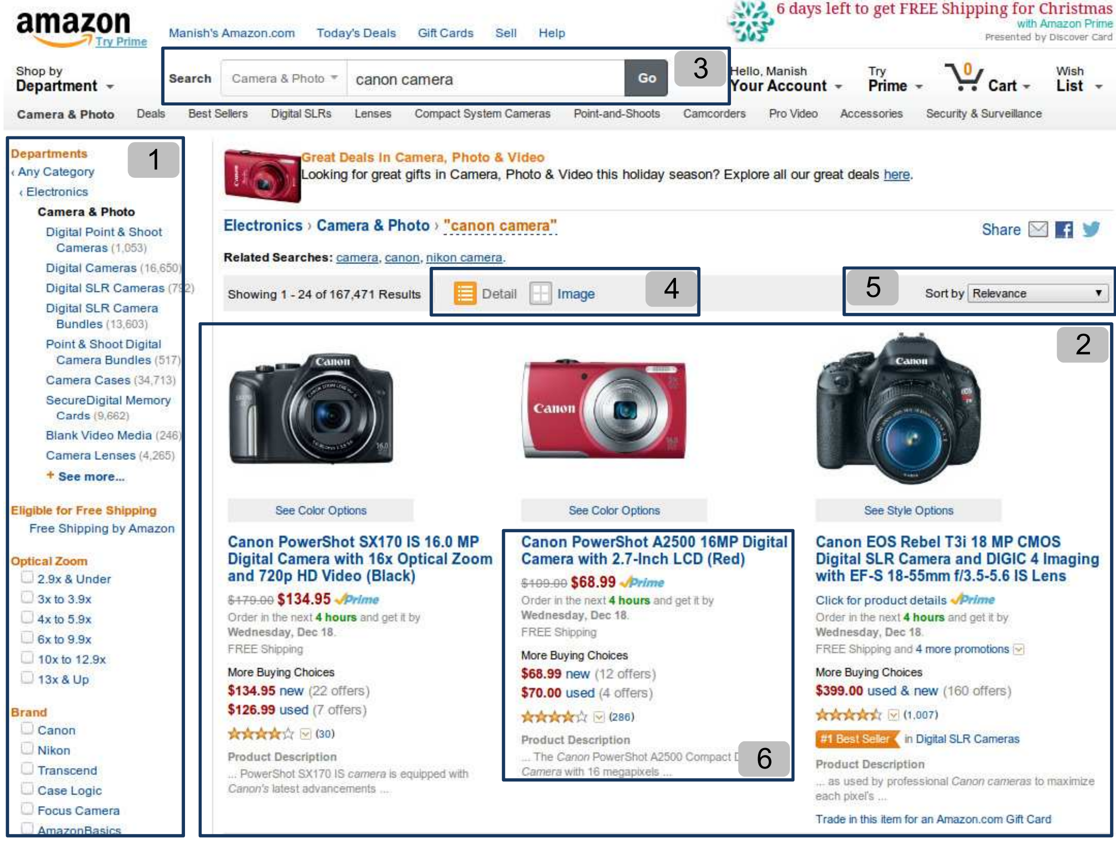


Figure 2.1: This screen capture from `amazon.com` shows many add-on extensions that are seen in typical faceted interface.

is quite usable for exploring small databases having homogeneous (same set of attributes) collections of items, it is not very effective for exploring large databases having heterogeneous collection of items.

In this chapter, we use the two screen captures shown in Figure 1.1 and Figure 2.1 to explain the important components and add-on extensions that are found in typical faceted interfaces.

1. Facet: Facets refer to categories used to characterize information items in a collection. Each facet has a name, such as `Departments, Optical Zoom, Brand, Price,` etc. A facet can be *flat* or *hierarchical*. For hierarchical categories, such as `Department, Location,` etc., one can use hierarchical facets so that users can roll-up or drill-down in a hierarchy of categories. Since our primary focus is on relational databases with flat columns, we treat each relational attribute as a flat facet. In this dissertation, we use the terms facet, dimension, feature and attribute interchangeably.

Based on whether a facet can be queried, one can divide the facets into two groups: queriable and non-queriable facets. Since the queriable facets are typically shown in the main query panel, we can call them as *shown* facets. In a traditional faceted interface, the non-queriable facets are like additional features that users can possibly see for individual tuples, but they cannot query or get summary digest for such facets. We call these non-queriable facets as *hidden* facets or additional features.

2. Facet Value: For each facet, we enumerate a discrete set of facet values, with a preference to keep this set small. For facets with categorical domains, or with a small discrete numerical domain, the facet values are obtained directly from the domain. Where the number of values is very large, such as for most numerical domains, ranges of values are binned together to create a small number of discrete facet values. For example, an `Optical Zoom` facet might be shown with a few different discrete zoom ranges (*e.g.*, 2.9x & under, 3x to 3.9x, 4x to 5.9x) as facet values. In a faceted interface, users select facet values using links or check boxes. Links usually indicate straight-

forward equals condition (single selection). For example, "I want to narrow down my search result to Department equals Digital Camera". Check boxes typically indicate an additive "or" condition (multiple selection). For example, "I want to narrow down my search result to Brand equals Canon, Nikon or Sony"

3. The Query Panel: The query panel is usually on the left (or top) side, and offers both user interface controls as well as a summary digest of the current query and the result set. In Figure 1.1, the query panel has two separate parts: the users' current query at the top and summary digest at the bottom. The summary digest typically comprises all the facet values that appear in the current query result, grouped by their corresponding facet. The summary digest may also include a tuple count for each facet value. As users select facet values from the summary digest, the facet values become part of the users' current query. Users can select single or multiple facet values from any subset of the displayed facets.

In Figure 1.1, the top part of the query panel shows the user's current query, where the user has selected single value for New/Used and Body Style, and several values for Make. The bottom part of query panel shows the summary digest of current query result in the form of all facet values that appear in the query result grouped by their facets. The summary digest has some amount of statistical information in the form of tuple counts next to each unchosen facet value (*e.g.*, there are 87 cars with Price of Up to 5000). In many faceted interfaces, such as Figure 2.1, the user's current query and the summary digest is shown in interleaved manner.

4. The Keyword Search Bar: The keyword search bar, labeled as box 3 in Figure 2.1, is one of the most important extensions of the query panel. It is crucial in applications that require search over heterogeneous collection of items. Based on the keywords that users enter in the keyword search bar, the system determines the appropriate facets that should be shown on the left-side query panel. In this dissertation, we do not deal with the keyword search bar.

5. Result Panel: The result panel, labeled as box 2 in Figure 2.1, typically occupies the majority of the screen real estate and shows the set of currently selected items. The results panel is scrollable, so the user can examine more results than can be displayed on the screen at once.

The basic faceted interface shows the query result in the result panel in some random order, and thus is not very usable for browsing large result sets. There are various existing add-on extensions to improve the usability of the result panel. For example, the designer can provide an appropriate ranking function to rank the result tuples. Even users can reorder their result set using provided sorting options, such as sort by `Relevance, Price,` etc., as shown in box 5 of Figure 2.1. Users can also see the result in zoomed-in or zoomed-out view by using options—`Detail, Image` view—as shown in box 4.

6. Result Snippet: Since most datasets are high-dimensional, it is not possible to show all the attributes of each result tuple in the main result panel. Rather each result tuple is shown in the form of a snippet, where each snippet is a small subset of attribute values, as shown in box 6 of Figure 2.1.

For a database with homogeneous data, one can show the same subset of attributes for all the tuples as snippet. This is the easiest form of snippet. However, most e-commerce applications have to handle heterogeneous data and they show different subset of attributes for each result tuple as snippet, as can be seen in both Figure 1.1 and Figure 2.1. To create such a variable format snippet, they create an additional attribute where they manually enter the snippet that should be shown whenever the particular tuple appears in the query result. Both the above types of snippets are static in the sense that they are not dependent on the users' query.

CHAPTER III

Querying High-Dimensional Data Using Faceted Interface

3.1 Problem Overview

In this chapter and Chapter IV, we address limitations in the query panel of faceted interface in relation to information exploration. We use the following running example to illustrate the key concepts:

Example III.1. Consider a used-car database of an auto dealer, which contains a single table $\mathcal{D}$ with n attributes where each tuple represents a car for sale. The table has numerous attributes that describe details of the car, such as `Price`, `Make`, `Model`, `BodyType`, `Drivetrain`, `Mileage`, `EngineSize`, `NumCylinders`, `Color`, `FuelEconomy`, `Power`, `Year` etc. Consider a user Mary who is unfamiliar to car domain and wants to buy a used-car. She has constraints in her price budget. She is only interested in cars whose Year $\geq$ 2012. She starts her exploration with an intial result set $\mathcal{R}$ = SELECT * FROM $\mathcal{D}$ WHERE `Year` $\geq$ 2012. This query will lead to a large result set with thousands of tuples. Mary has to specify more constraints to get to a small result set that also satisfies her other preferences.

Let's consider Mary's difficulties in selecting a suitable `BodyType`. The dataset has 12 different body types, such as Sedan, SUV, Hatchback, Coupe and so on, to choose

from. If she is not very informed of how different body types compare, then showing a list of all the available body types is of very limited help. Let's assume that of all the body types, Mary's highest preference is for SUV. If she constrains her result set $\mathcal{R}$ by additionally selecting `BodyStyle` = SUV, then in her chosen year range, she will mostly get SUVs that are quite costly. Moreover, SUVs typically have poor fuel efficiency. On the other hand, if she selects `BodyStyle` = Hatchback, she will get many Hatchbacks with lower price and good fuel efficiency. But these cars have very limited cargo capacity. Since each body type has very complicated relationship with many other attributes, such as `Model`, `Make`, `Drivetrain`, `Price`, `FuelEconomy`, `GroundClearance`, `EngineSize`, etc. It is difficult for Mary to choose the body type such that she can get to a next smaller result set where most of the cars simultaneously satisfy her higher preferences in other attributes.

When Mary selects `BodyType` = SUV, then the next result set matches her preference in many attributes, such as `Model`, `Make`, `CargoCapacity`, and so on. However, she does not like the values in few attributes such as `Price` and `FuelEconomy`. If she has to come to this conclusion by actually looking at tuples, then she has to go through many tuples before she can come to the conclusion that in this year range most of the SUVs would be quite expensive. Moreover, SUVs in general have low fuel efficiency. Similarly, when she selects Hatchback, she has to again go through many tuples to realize that none of them have good cargo capacity.

Users often have an unspecified, complicated preference function that spans across multiple attributes. When they try to select optimal choice(s) along one attribute, it often leads to non-optimal choices in other attributes. In most real datasets, each attribute is dependent on many other attributes. When one selects a value from one attribute, it leads to implicit selection of values in other attributes. One cannot understand these implicit selections by observing one or few tuples. However, by looking at many tuples, one can statistically understand these implicit selections.

For example, most SUVs in Mary's chosen year range will be costly compared to cheaper body types such as Hatchbacks, Wagons and Sedans. When users are about to select an attribute value, if we can show all the important implicit selections in other attributes, then users will not have to take this approach of trying to take an exploration path, and then later on retract because it does not satisfy the preference in some other attribute. If users see that the implicit selection in even one attribute is not according to their preference, they can save the effort of exploring further in that direction.

There are two ways common ways of browsing structured data: as individual tuples and as aggregated summaries. When there are only few tuples one can easily compare them in a pairwise manner. However, if there are many tuples, then one prefers to look at aggregated summaries and then compare a given tuple with respect to those aggregated summaries. As compared to looking at individual tuples, one can evaluate the relevance of a search result set more effectively by looking at the aggregated summaries. We next explain the challenges in browsing individual tuples of complex datasets, such as used-cars, and also the challenges in creating meaningful statistical summaries for such datasets.

MPG	**DriveTrain**	**Cyl**	**Price**	**Make**	**BodyType**
31	2WD	V6	24K	Ford	SUV
32	2WD	V4	12K	Chevrolet	Wagon
25	2WD	V6	17K	Dodge	Minivan
21	4WD	V8	38K	Chevrolet	SUV
30	2WD	V4	11K	Kia	Wagon

Table 3.1: Sample tuples from a used-car dataset

In Table 3.1, we show five cars from the auto-dealer's database. It is very difficult for humans to easily comprehend large amount of alpha-numeric information when presented in this type of tabular and unordered form. In this type of data presentation, users can clearly see the information in a given tuple, but they cannot easily

compare the given tuple with respect to other tuples in the database. Because of not being able to see the relative comparison between various available choices, they often end up choosing an inferior choice. For example, in Table 3.1, there are two SUVs that have very different prices. Just by looking at all the attributes of these two tuples, users cannot reliably say which attribute(s) are the main cause for this price difference. But if they have looked at many tuples and observed how different attributes affect price, then they can more precisely point out the exact cause(s) of this price difference.

For complex datasets, as in Table 3.1, a simple columnwise summary, such as min, max, average, etc., will provide very limited insight to users. It is very unlikely that just by computing a single data summary, we can satisfy all the needs of all the users. For example, when Mary wants to compare different body types in a given year range, we should show her a statistical summary comparison of how cars of different body types compare at that year range. In the form of summary digest, faceted interface shows some amount of statistical summary of the query result. However, in the faceted interface, if users select multiple values from the same facet, then the statistical summaries of all the values gets commingled, and as a result even the simple count based summary digest is no longer usable.

In complex datasets, users cannot see the interaction between attributes, which is required for them to effectively visualize and explore such high-dimensional datasets. In this chapter, we propose a new data summarization technique, known as *Implicit Choices Summary* (ICSummary), which can be used as an add-extension to faceted interface. For a given result set and a facet value, an ICSummary summarizes the interaction of the given facet value with values in other top-l related facets. In the query panel of faceted interface, when users hover over a facet value, we show them an ICSummary of the facet value in the form of a pop-window. Since faceted interface is typically used by normal end-users, we make sure that the information shown in

ICSummary is simple, informative and not overwhelming to users.

The main contributions of this chapter are as follows:

- We present limitations in the query panel of faceted navigation, especially for datasets having many facets or having facets with many values. (Section 3.2)
- We present an interaction and query technique called Implicit Choices Summary (ICSummary) that addresses the limitations by highlighting the important relationships between facet values within and across facets. (Section 3.3)
- We present the algorithms and techniques necessary to create and present the relevant ICSummaries. (Section 3.4 and Section 3.5)
- We perform detailed evaluation using two real datasets. (Section 3.6).

Finally, we conclude in Section 3.7.

3.2 Limitations in the Query Panel

In this chapter, we address the following three limitations in the query panel:

Limitation 1. Size of Query Panel — Due to limited screen real estate (and user fatigue) a faceted interface can typically show only 5-10 attributes as queriable facets in the query panel. This is a small subset of available attributes in most e-commerce applications. Users can neither select facet values nor see summary digest for the non-queriable facets. If there are categorical attributes that have many facet values, then to save screen space even such attributes are not typically shown as queriable facets. Instead of hiding all the information in the non-queriable facets, it would be more effective if the query panel can show to users the relevant information in the non-queriable facets through the queriable facets.

Limitation 2. Back and Forth Browsing — In the query panel of faceted interface while selecting a value from one facet, users cannot see what values are getting

implicitly selected in the other facets. After they select the value, they have to browse to other facets to see what choices are available in other facets. If they do not like the values in other facets, then they have to modify their selected value and it causes lots of back-and-forth browsing across the query panel. Instead of making users go repeatedly back-and-forth across the query panel, we can automatically compute the information that users are trying to manually obtain by looking at the summary digest, and provide it to them as a preview information, in the form of pop-window, even before they explicitly select the facet value.

Limitation 3. Support for Multiple Selection — In faceted interface with single-selection per facet the facet values are often shown along with their tuple counts. Although very inconvenient, users can understand the dependencies between facet values across facets by observing changes in the tuple counts after each selection. Most e-commerce sites allow users to perform multiple selections per facet. Tuple counts are often not shown in multi-selection interface because it breaks the query semantics. When users select one facet value, the count of all the remaining values in that facet becomes zero. However, the query panel of multi-selection faceted interface has to show all the facet values that are present in the database, and not just the users' current result. Showing a summary digest with many facet values having zero counts (or the total counts in the database) has very limited utility to users. Moreover, when users select multiple values from the same facet, then the count information for values in other facets gets commingled. The limited summary digest shown in multi-selection interface further limits users' ability to understand facet interaction.

3.3 Implicit Choices Summary

In this section, we discuss our solution to the limitations of faceted navigation for information exploration in complex databases. Our solution relies on a novel

summary feedback information, known as *Implicit Choices Summary* (ICSummary). One can view ICSummary as a more detailed feedback compared to the tuple count information that is shown in the existing summary digest. We discuss the design and algorithmic challenges in creating and presenting the ICSummaries.

3.3.1 The ICSummary

3.3.1.1 Overview

The ICSummary is a new component that we propose adding to the standard faceted navigation (in addition to the existing query, result, and summary digest components). It can be seen as an extension of the query panel whose role is to show the interaction of an attribute value with values in other attributes in a pairwise manner.

ICSummary: Given two result set $\mathcal{R}_1$ and $\mathcal{R}_2$, such that $\mathcal{R}_2 \subseteq \mathcal{R}_1$, an ICSummary shows the statistical difference between $\mathcal{R}_1$ and $\mathcal{R}_2$. The result set $\mathcal{R}_2$ is obtained from the result set $\mathcal{R}_1$ using a more constrained query.

The ICSummary as defined above is for any two given result sets. However, in this chapter, we will use an ICSummary in the context of a given facet value v from a given facet f. We assume that $\mathcal{R}_1$ is the result set when a set of facet values V is selected from the given facet f. $\mathcal{R}_2$ is the result set when only one facet value $v \in V$ is selected, with conditions in all other facets being the same as in result set $\mathcal{R}_1$. We use ICSummary to show the implicit selections due to facet value v as compared to the implicit selections due to all the facet values in V.

When users are about to constrain their result to a specific facet value, our goal is to show them a preview of how the particular facet value changes their current search result. The tuple counts that are shown in the traditional faceted interface are a simple form of such a summary. The tuple count is the preview of what would

be size of the next result set. But the tuple count does not provide a preview of what would be the distribution of facet values in other facets in the next result set. In order to show the effect of selecting a particular facet value, we show to users an ordered list of facets that would be most affected by the selection of the given facet value, and also the changes in facet value distribution of those most affected facets.

In Figure 3.1 and 3.2, we show two sample ICSummaries obtained from a real dataset for the example query discussed in Section 3.1. We use a tabular structure to represent ICSummary. For a given facet value v, the first column in the ICSummary table shows a ranked list of dependent facets whose facet value distribution is most affected by selection of the facet value v, which includes both queriable and hidden facets. For each dependent facet we show its chi-square and p-value in parenthesis. In the remaining columns we show representative facet values from each of the dependent facets categorized according to some given distribution change categorization function $\mathcal{C}$ (explained in the following paragraph). For each facet value we show its count information in brackets as (observed count/expected count). We use the concept of categorization function based on motivation from layouts that are used in text documents. A text document is easy to follow when it follows a standard layout, with all the information being arranged in appropriate sections. Similarly, to summarize the distribution change in each of the dependent facets in a uniform manner, we assume a given distribution change categorization function $\mathcal{C}$.

One can come up with a categorization function $\mathcal{C}$ that is customized for specific application. However, we propose a function $\mathcal{C}$ that has generic importance. We categorize the implicitly selected facet values in other facets using two dimensions: *frequency* and *likelihood*. Tuple count is an important information that is typically shown in the query panel of faceted interface, except multi-selection interface. Based on the observed tuple counts we categorize the facet values into following three categories: *high frequency*, *medium frequency* and *low frequency*. We lay the columns in

ICSummaryPScore – Mozilla Firefox

localhost:8080/solr/collection1/browse?&q=&fq={!tag=dt}Year:2011-2012"OR"{!tag=dt}\

ICSummary for BodyType = Sedan

	Frequency		
Facets	**High**	**Medium**	**Low**
Model (96141, 0.0)	'Impala LT'(370/182) 'Altima '(234/115) 'Malibu LT'(204/100) 'Sentra '(203/99) 'Camry LE'(192/94)	'Sonata GLS'(168/82) 'Corolla LE'(166/81) 'Camry '(164/80) 'Fusion SE'(163/80) 'Cruze LT'(142/69) 43 More Values	'Mazda3 '(36/22) 'Mazda3 i Sport'(36/17) 'Aveo LS'(36/22) 'Cruze LTZ'(35/17) 'S60 T5'(34/16) 266 More Values
Doors (20999, 0.0)	4(7023/6365)		2(0/584) 3(0/72)
Make (14504, 0.0)	Chevrolet(1228/1205) Toyota(889/606) Nissan(823/682) Ford(733/1000)	Hyundai(495/325) Dodge(412/412) Volkswagen(360/271) Chrysler(323/243) Honda(254/258) 5 More Values	Buick(105/71) Mitsubishi(88/104) Acura(81/77) Cadillac(70/78) Lincoln(64/51) 15 More Values
Engine (8046, 0.0)	V4(4574/3549)	V6(1986/2492)	V5(281/193) V8(182/779)
Drivetrain (7424, 0.0)	2WD(6640/5414)		AWD(383/807) 4WD(0/801)
Price (5332, 0.0)	15-20K(3106/2409) 10-15K(1808/1237)	20-25K(946/1341) 25-30K(507/846)	30-35K(259/459) 35-40K(124/278) 50K+(97/182) 40-45K(87/140) 45-50K(59/108) 1 More Values
Color (1429, 0.0)	Silver(1602/1506) White(1342/1405) Black(1283/1302) Gray(1002/972)	Red(766/788) Blue(610/568)	Gold(140/113) Beige(98/87) Brown(78/90) Green(38/80) Bronze(38/36) 3 More Values
Mileage (933, 0.0)	30-40K(1838/1614) 20-30K(1688/1749) 10-20K(1582/1660) 0-10K(1066/1210)	40-50K(673/574)	50-60K(132/153) 60-70K(22/32) 80-90K(12/9) 70-80K(8/13)
Transmission (739, 0.0)	Automatic(6787/6725)		Manual(236/297)
Year (44, 4.008686667056338E-7)	2011-2012(6814/6816)		2013-2014(209/206)

Figure 3.1: ICSummary for `BodyType` = `Sedan`, where the user has selected cars with `Year` $\geq$ 2012. For each dependent facet we show its chi-square and p-value in parenthesis. With each facet value we show its observed count / expected count information in parenthesis. The font colors green, blue and red indicates increased, unaltered and decreased likelihood respectively.

ICSummaryPScore – Mozilla Firefox

localhost:8080/solr/collection1/browse?&q=&fq={!tag=dt}Year:2011-2012"OR"{!tag=dt}

ICSummary for BodyType = SUV

	Frequency		
Facets	**High**	**Medium**	**Low**
Model (96141, 0.0)	'Escape XLT'(138/33) 'Traverse LT'(112/27) 'Equinox LT'(98/24) 'Sorento LX'(98/24) 'Liberty Sport'(93/22) 4 More Values	'RAV4 '(61/14) 'Edge Limited'(59/14) 'Santa Fe GLS'(58/14) 'Compass Sport'(54/13) 'Patriot Sport'(52/12) 64 More Values	'Nitro Heat'(12/2) 'Juke SL'(12/2) 'CX-7 i Sport'(12/2) 'Armada SV'(12/2) 'Flex SE'(12/2) 183 More Values
Doors (20999, 0.0)	4(3444/3177)		2(61/291) 3(0/36)
Make (14504, 0.0)	Ford(597/499) Chevrolet(592/601) Jeep(394/96)	Nissan(224/340) GMC(184/60) Kia(164/115) Honda(157/128) Dodge(143/205) 8 More Values	Cadillac(57/39) Acura(54/38) BMW(46/60) Lincoln(41/25) Buick(40/35) 12 More Values
Engine (8046, 0.0)	V6(1999/1244) V4(1065/1771)	V8(441/389)	V5(0/96)
Drivetrain (7424, 0.0)	2WD(1344/2702) AWD(1112/403) 4WD(1049/399)		
Price (5332, 0.0)	20-25K(935/669) 15-20K(769/1202) 25-30K(662/422)	30-35K(408/229) 35-40K(303/139) 50K+(150/90) 40-45K(143/70) 45-50K(100/54)	10-15K(35/617) 5-10K(0/9)
Color (1429, 0.0)	Black(764/650) Silver(736/751) White(659/701) Gray(487/485)	Red(349/393) Blue(243/283)	Brown(68/44) Gold(66/56) Beige(50/43) Green(49/40) Orange(14/18) 3 More Values
Mileage (933, 0.0)	20-30K(1021/872) 10-20K(858/828) 30-40K(703/805) 0-10K(553/604)	40-50K(213/286) 50-60K(114/76)	60-70K(25/16) 70-80K(12/6) 80-90K(5/4)
Transmission (739, 0.0)	Automatic(3469/3356)		Manual(36/148)
Year (44, 4.008686667056338E-7)	2011-2012(3367/3402)		2013-2014(138/102)

Figure 3.2: ICSummary for `BodyType` = `SUV`, where the user has selected cars with `Year` $\geq$ 2012. For each dependent facet we show its chi-square and p-value in parenthesis. With each facet value we show its observed count / expected count information in parenthesis. The font colors green, blue and red indicates increased, unaltered and decreased likelihood respectively.

ICSummary table based on observed tuple counts. Tuple counts alone is not a sufficient indication of how the result set changes with each selection. If we assume that facets are all independent, then we can use probablity measures to get the expected counts of facet values in other facets. We explain expected count computation in Section 3.4.1.2. We can highlight to users how interesting a facet value is based on the difference between expected and observed count. We categorize the facet values into following three categories based on the difference between their observed and expected counts: *increased likelihood*, *unaltered likelihood* and *decreased likelihood*. The term *increased likelihood* means that difference between observed and expected counts is statistically significant and the observed count is greater than the expected count. Similarly, one can define the other likelihood based categories. We later on explain how one can use p-values to mathematically define these categories.

3.3.1.2 User Interaction

Since an ICSummary contains a lot more information compared to tuple count, we cannot show ICSummary of all the queriable facet values simultaneously because it will overwhelm users with information overload and will also clutter their interface. Rather we need to keep the ICSummaries as hidden information. We assume the following two modes of user interaction: *Hide ICSummary* and *Show ICSummary*, where users need to explicitly toggle between the two modes. In the *Hide ICSummary* mode, users won't see any ICSummaries, which is same as the traditional faceted interface. In the *Show ICSummary* mode, the interface uses some client-side scripting language, such as Javascript, to show the ICSummary of a facet value as a pop-up window as users hover over the particular facet value.

3.3.2 Algorithmic Challenges

Faceted navigation is a user-facing application with strict computational constraints. The ICSummary of all facet values changes with the change in query result. We defer the discussion on computational constraints to Section 3.5. In this section, we present algorithmic challenges in creating the ICSummaries.

The ICSummary is a complex structure. Our goals are (i) to populate this structure effectively, making the most of limited screen real estate available, and (ii) to arrange and present the information populated in this structure to maximize its value to users.

For the first goal, we have to efficiently find the set of most dependent facets, and then summarize the change in facet value distribution of those dependent facets. The ICSummary table lays out each dependent facet in rows.

For the second goal, the system must further decide how to choose and summarize each dependent facet, so that users get the maximum amount of dependency information in a concise, uniform and comparable manner.

3.3.2.1 Creating ICSummary

ICSummaries are created in the context of an ongoing navigation session, so there is always a set of selected facet values (this set could be empty) and an associated result set $\mathcal{R}$. With every change in users' query, we need to recompute ICSummary for all queriable facet values. Therefore, our overall task can be written as:

Problem 1 (Compute ICSummary): *Given a queriable facet $f \in \mathcal{F}$, a set of facet values $\mathcal{V}$ selected in f, a facet value $v \in \mathcal{V}$. Let $\mathcal{R}_{\mathcal{V}}$ be the result set in which all the $\mathcal{V}$ values are selected and $\mathcal{R}_v$ be the subset of $\mathcal{R}_{\mathcal{V}}$ in which only the facet value v is selected. Compute an ICSummary for facet value v by measuring the statistical difference between $\mathcal{R}_{\mathcal{V}}$ and $\mathcal{R}_v$.*

In Problem 1, our goal is to compute an ICSummary that highlights how the implicit selections due to facet value v compares with the implicit selections due to all the selected facet values $\mathcal{V}$. For example, a user might have selected four body types: Sedan, SUV, Wagon and Minivan, which we denote as the set $\mathcal{V}$. The user then wants to know how the implicit selections of Sedan compare with combined implicit selections of all the four body types.

Creating an ICSummary for a facet value entails the following two sub-tasks: to obtain the facets that are most affected by the selection of the given facet value, and then summarize the change in facet value distribution of those most affected facets. We present these two sub-tasks more formally in Problem 1.1 and Problem 1.2 respectively.

There are two contradictory user needs while browsing the ICSummaries. In terms of information quality, one would like the ICSummary of each facet value to show the best set of dependent facets for that facet value. However, users also have another very important requirement, namely *comparability*. ICSummaries would be usable if they have uniform structure and are comparable with each other. The tuple counts that are shown in the existing query panel is a uniform summary that is similar for all the queriable facet values. But we cannot expect the ICSummaries of all the facet values to look similar because all facet values do not have the same set of dependent facets. However, for comparability reasons we can make the ICSummaries of all the facet values within each queriable facet to look similar by selecting the same set of dependent facets, and showing the difference in facet value distribution of the common set of dependent facets. Thus, to create uniform and comparable ICSummaries, we need to solve the following two sub-problems:

Problem 1.1 (Find Dependent Facets): *Given a queriable facet f, a set of facet values $\mathcal{V}$ currently selected from f, find the top-l facets $\mathcal{I} \subseteq \mathcal{A}$ s.t. the facet values in $\mathcal{I}$ have maximum dependency on the given set of queriable facet values $\mathcal{V}$.*

In Problem 1.1, we compute the set of dependent facets $\mathcal{I}$ for a given queriable facet f by taking into account the facet values $\mathcal{V}$ from facet f that are present in the users' current query result. By looking at the distribution of facet values in the given queriable facet f, one can use information theoretic measures to compute dependent attributes that would be of maximum interest to the users.

To create comparable ICSummaries, we not only need to find the same set of dependent facets for the facet values in each queriable facet, but we also need to summarize the facet values in those dependent facets in a uniform manner. One way to create such a summary, for each dependent facet, is to categorize the facet values into different groups according to some common given categorization function. We use the categorization function $\mathcal{C}$ discussed in Section 3.3.1.1.

Problem 1.2 (Summarize Dependent Facets): *Given two result sets $\mathcal{R}_{\mathcal{V}}$ and $\mathcal{R}_v$, a queriable facet value v, the informative dependent facets $\mathcal{I}$, and a categorization function $\mathcal{C}$, create ICSummary for v by categorizing the facet values of each facet in $\mathcal{I}$ using categorization function $\mathcal{C}$ over the two given result sets $\mathcal{R}_{\mathcal{V}}$ and $\mathcal{R}_v$. The number of representative facet values that are shown from each dependent facet is determined by the user's display constraint.*

3.3.2.2 Top-K Queriable Facets

Using ICSummary it is possible to show information about hidden facets through the queriable facets. We can use ICSummary to avoid showing facets that have too many facet values or have redundant information. Our next problem is to help the designer select the top-k queriable facets.

Problem 2 (Find Informative Queriable Facets): *Given a database $\mathcal{D}$ with attributes $\mathcal{A}$, a subset of preselected queriable facets $\mathcal{S} \subseteq \mathcal{A}$, find the set of top-$k$ queriable facets $\mathcal{F}$, such that $|\mathcal{F}| = k$, each facet in $\mathcal{F}$ has less than $\mathcal{C}$ facet values,*

and $\mathcal{F}$ minimizes the information loss due to hidden facets $\mathcal{H}$, and $\mathcal{S} \subseteq \mathcal{F}$.

In Problem 2, our goal is to find the top-k queriable facets $\mathcal{F}$, where we assume that the interface designer has already selected some subset of attributes $\mathcal{S}$ as queriable facets, and so we can select only $(k - |\mathcal{S}|)$ additional queriable facets. There are many possible ways to define what should be the top-k queriable facets. For example, one can use actual workload from a given query log to find the most frequently queried attributes [23], or one can use information theoretic measures to find attributes that have interesting properties. In this chapter, we use an information theoretic approach where our goal is to minimize the information loss that users have to incur due to hidden facets. We want to select those attributes as queriable facets that require less than a given threshold amount of screen space, and can simultaneously provide information about the hidden facets through strong dependency relationships. If the queriable facets have strong dependency with hidden facets, then in the query panel users can see not only the information about the queriable facets, but can also see the relevant information about the hidden facets through the ICSummaries.

3.4 Algorithms

3.4.1 Creating ICSummary

In this section, we describe our solution to Problem 1. We use chi-square based statistical tests to select dependent facets and also to determine the "interesting" changes in likelihoods of facet values within each dependent facet. We use chi-square combined with p-values to ensure that the test results are applicable for all categorical attributes with any domain size. These chi-square tests are only applicable to categorical attributes. Although our datasets can have both numerical and categorical attributes. But after the numerical attributes are discretized, we can treat them as categorical attributes.

3.4.1.1 Find Dependent Facets

In this section, we present our solution to finding the dependent facets for a given set of facet values $\mathcal{V}$ that are currently selected from facet f. We compute the dependent facets by computing the dependency of facet values in all other facets with the values in facet f. We use chi-square test for independence to determine whether two attributes or variables are related. This test does not indicate causality. Using chi-square test we can measure the pairwise interaction between facets values in facet f with facet values in other facets. We use the standard significance level of p-value equals 0.05. We select the set of dependent facets $\mathcal{I}$ by taking the top-l attributes with highest chi-square statistic and p-value less than significance level of 0.05.

This statistical test leads to interesting results. For example, in Figure 3.1 and 3.2, we can observe that all the dependent facets, except facet `Doors`, shows great contrast between Sedan and SUV through the difference in their implicit selections. Although facet `Doors` don't show contrast between these two body types, still it is selected as the second most important dependent facet because the number of doors plays an important attribute for contrasting from other body types that has 2 or 3 doors. Since Mary had only specified a price range so these facets have been selected for comparing all the body types, and not just Sedans and SUVs.

ICSummary is much more flexible compared to parallel coordinates (see Chapter VI) in terms of showing the interaction between attributes. Parallel coordinates can show interaction between a fixed subset of attributes, and that too in a particular sequence, where one can see the interaction between only the adjacent attributes. In ICSummary, we dynamically select the top-l attributes that has most interesting interaction with the selected facet values $\mathcal{V}$, and show the interactions with all the l dimensions in an easily understandable manner.

3.4.1.2 Summarize Dependent Facets

In this section, we explain the details of our categorization function $\mathcal{C}$. As discussed in Section 3.3.1.1, we use two dimensions to categorize facet values: *frequency* and *likelihood*. For each facet value we show its observed count and expected count. When a user hovers over a facet value v, we extract the subset of tuples $\mathcal{R}_v$ from $\mathcal{R}_\mathcal{V}$ and compute the observed counts of all the facet values present in the dependent facets $\mathcal{I}$. Based on the counts of facet values in $\mathcal{R}_\mathcal{V}$, we also compute the expected counts when the user constrains to result set $\mathcal{R}_v$ from $\mathcal{R}_\mathcal{V}$. These expected counts are computed assuming all facets are independent.

We use $C_\mathcal{R}(u)$ and $E[C_\mathcal{R}(u)]$ to denote observed and expected counts of facet value u in result set $\mathcal{R}$. We assume that user has hovered over facet value v from facet f in which currently the selected set of facet value is $\mathcal{V}$. To compute the expected count, we use the following expression:

$$E[C_{\mathcal{R}_v}(u)] = \frac{C_{\mathcal{R}_\mathcal{V}(v)}}{\sum_{\forall w \in \mathcal{V}} C_{\mathcal{R}_\mathcal{V}(w)}} \times C_{\mathcal{R}_\mathcal{V}(u)} \tag{3.1}$$

We next explain the algorithm to categorize values in a dependent facet $d \in \mathcal{I}$. There are many techniques that one can use to categorize facet values based on their observed frequency count, such as quantiles. Quantiles are mathematically quite well-defined. However, they have some limitations. For example, if we have three facets values with counts 40, 5 and 3 respectively, and we consider 3-quantile, then it will assign each value facet to a separate category. In this case the counts 5 and 3 are so similar compared to count 40 that it is preferred that we show them in the same category. But 3-quantile will show each facet value in a different category. We want to create the ICSummary in such a way that just by looking at the cell where a facet value is placed, the user should get an impression of what would be the observed count of that facet value. For example, if we consider the `Drivetrain` facet in Figure 3.1,

then we can see that observed counts of AWD and 4WD are quite small compared to 2WD. We place AWD and 4WD in the same low-frequency category. Moreover, one can see that there is no `Drivetrain` facet value in the medium-frequency category. The categories in which facet values are placed give an indication to users of how many results they will find if they explore in that direction. Quantiles will not lead to this type of categorization.

In any given query result, for each facet the facet value that has the maximum count is considered important in most applications. We want to show how all the facet values in a facet compare with the facet value that has maximum count. To do this we propose the following observed count based categorization function: Consider facet value u in d whose count $C_{\mathcal{R}_v}(u)$ is maximum. Based on observed counts categorize all the facet values in d into following three groups: high-frequency $(C_{\mathcal{R}_v}(u)/2 - C_{\mathcal{R}_v}(u)]$, medium-frequency $(C_{\mathcal{R}_v}(u)/10 - C_{\mathcal{R}_v}(u)/2]$, and low-frequency $[0 - C_{\mathcal{R}_v}(u)/10]$. In order to find better break points, one can also use standard discretization techniques used in statistics and machine learning, such as [74, 34]. Since the screen space limited, we can only show limited number of facet values from each of the dependent facets. Most facets have less than 15 facet values, thus by showing five to six facet values from each of the three categories, we can most likely show information about all the facet values in an ICSummary. If some facets have many facet values (e.g. `Model` or `Make`), then we show few sample representative values. If for some categories we cannot show all the values, then we show in the ICSummary the number of values that are not shown as "X More Values". From the high-frequency category we show the five facet values with highest observed count. For the other two categories we show five randomly selected facet values.

To categorize facet values according to likelihoods, we need a measure that can determine whether for a given facet value the difference between the observed and the expected count is significant or not. If the difference is due to chance, then we call it

unaltered likelihood. If the difference is significant, and the observed count is greater than expected count, we call it *increased likelihood*. Otherwise we call it *decreased likelihood*. In order to determine whether the difference in count is significant, we need to take the domain size in consideration. For example, in facet `Model` there are large number of facet values compared to facet `Drivetrain` that has only three values. With each change in query result, the change in count would be more for values in `Drivetrain` compared to values in `Model`. To determine whether the change in count is significant in a domain independent manner, we propose a new measure that uses an approach similar to standard chi-square statistical test.

In the standard chi-square test the goal is to determine whether an observed data is according to a specific hypothesis. The value of test-statistic is given by:

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} \tag{3.2}$$

That is, chi-square is the sum of the squared difference between observed (O_i) and the expected (E_i) value (or the deviation), divided by the expected value in all possible categories. For this test the degree freedom is given by number of categories in the problem minus 1, which is $n - 1$. Using the appropriate degrees of freedom, one can use the chi-square distribution table to locate the p-value closest to the calculated chi-square value. If the p-value is less than 0.05, which is the commonly used significance level, then the difference is considered statistically significant.

In our problem, we want to determine whether the difference between the observed and the expected count is significant for each individual facet value, rather than all the facet values as a set. As a result, we cannot directly use normal test-statistic. Instead we use a reversed approach. We count the number of facet values in the dependent facet in the given result set, and from there we compute the degrees of freedom. For the obtained degrees of freedom, we obtain from the chi-square distribution table the required chi-square value for the significance level to be 0.05. We then divide

that chi-square value by number of facet values and use this value as a threshold to determine whether the count difference for the single facet value is due to chance or is statistically significant. From various experiments, we observed that the chi-square value we get for 0.05 significance level is so small that most facet values fall within either *increased likelihood* or *decreased likelihood* category. When we scaled the chi-square value by a factor of five, we found that we get values in all the three categories, and they seem more useful from practical utility.

3.4.2 Selecting Top-K Queriable Facets

In this section, we describe our solution to Problem 2. Inorder to keep the query panel size small, we want to select facets that have small domain size, are important and informative about hidden facets. Using ICSummaries we want to provide information about hidden facets through the queriable facets.

To the best of our knowledge, this problem cannot be directly solved by standard feature selection algorithms. There are two types of feature selection algorithms: supervised and unsupervised. In supervised feature selection, the goal is to select a subset of features that are relevant to the class attribute, and preferably non-redundant. In unsupervised feature selection the goal is to select a subset of features that can lead to good quality clusters or improve similar other unsupervised data mining task. One can possibly use an existing unsupervised feature selection algorithm to select a subset of queriable facets that are important and also informative about hidden facets. But such algorithms cannot take into consideration the fact that we want queriable facets with small domain size. If we use a naive approach of simply removing all facets that have large domain size before running the feature selection algorithm, it will not lead to good quality results.

We present a novel way of selecting top-k queriable facets that is based on graphical models, such Bayesian network or Markov network. Probabilistic graphical models

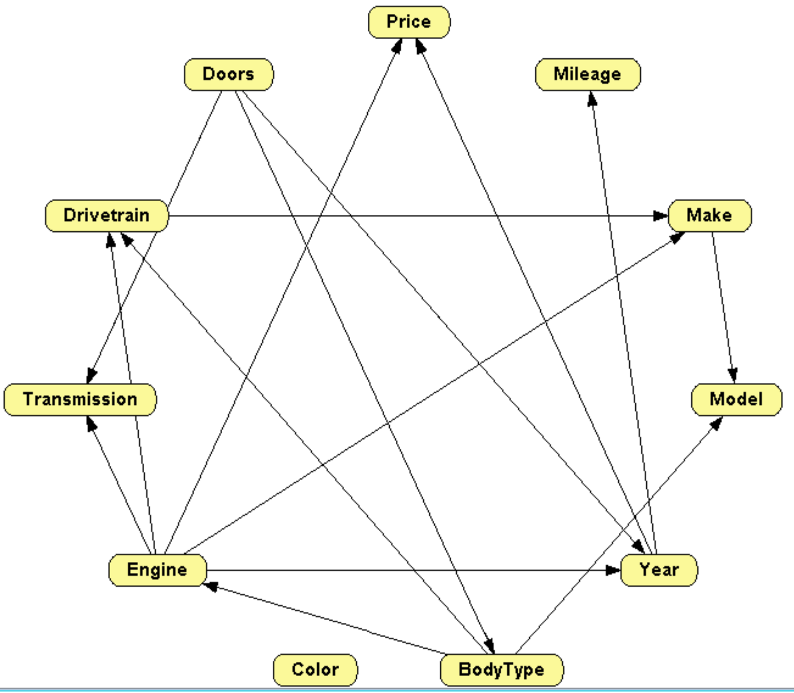


Figure 3.3: Conditional dependency between attributes of a used-car dataset represented in the form of Bayesian network.

use a graph-based representation as the foundation for encoding a complete distribution over a multi-dimensional space. In Figure 3.3, we show dependencies between the attributes of our used-car dataset using Bayesian networks. Graphical models can more precisely show the interaction between attributes as compared to feature selection algorithm. Once we have the complete dependency graph between attributes, we can select the right subset of attributes according to our specific optimization function. To solve Problem 2, we first of all compute the eigen vector centrality of all the nodes (or attributes) in the dependency graph. There are many eigen vector based centrality finding algorithms, such as Katz centrality, PageRank, etc. We then sort all the attributes by their centrality score. To select the top-k queriable facets we start by including the preselected attributes $\mathcal{F} = \mathcal{S}$. Until we get k queriable facets in $\mathcal{F}$, we go through the sorted list (in the descending order) of attributes based on centrality score, and include any attribute that has less than $\mathcal{C}$ facet values and is not yet included in $\mathcal{F}$. The central attributes are important and can present information

about related hidden facets through ICSummaries.

3.5 Efficient Implementation of ICSummary

In this section, we explain the challenges in integrating ICSummaries with faceted search. The way data is managed in faceted search, it is efficient for the existing user interaction model. But if we want to provide additional insightful data summaries to users, then there are various challenges in building such a system with real-time performance. Although faceted search engine can support efficient computation of information needed in Problem 1.2, it has limitations in relation to Problem 1.1. In this section, we present a novel data summary, known as *Extended Summary Digest* (ESD) to maintain various count information, that can lead to very efficient solutions for both Problems 1.1 and 1.2.

3.5.1 Faceted Search Implementation

In this section, first of all we present a brief review of how faceted search is implemented to provide real-time performance over even large datasets. From our experiments, we observed that the retrieval time of faceted search increases slightly with increase in dataset size. However, for a given dataset the retrieval time remains almost same irrespective of the selection condition. In order to provide this type of performance, all of today's faceted search engines are build on top of inverted (or text) indexes. They achieve the fast search response by efficiently searching the index, instead of the document set. A typical inverted index maintains an ordered list of *terms*. Each term points to a *posting list* that includes an ordered list of the IDs of documents or tuples containing that term. To perform a search, the index first locates the terms matching the list of keywords, and then merges the posting lists of those terms to compute the matching document set. By using bitset to represent these posting list, these search engines can perform any combination of union and

intersection operation over these posting lists very efficiently [84]. In faceted search the pair <facet, facet value> is used as index term.

Given a faceted query q, the system does the following: (1) identifies all matching documents $\mathcal{R}_q$, (2) computes the count of each facet value in $\mathcal{R}_q$, (3) displays a subset of facet values and their counts as specified by the interface designer. Of all the three steps, step 2 is considered computationally most costly because the system has to get the exact counts from even large result sets in real-time. This problem is addressed through optimized bitset operations. Step 1 is not the major constraint because the system does not have to actually return all the result documents immediately to users. It just has to return some 10-50 matching documents, where the number of retrieved documents is determined by how many results users wants to see per page. As users click to move to the next page, the system has additional time to fetch the next page results. However, the time required for step 1 would become extremely high if we want the search engine to return a large number of documents (say 5K or more) at once, because the cost of constructing documents back from the index is quite high.

To solve Problem 1.1, we would need the result set $\mathcal{R}_{\mathcal{V}}$ in a form that we can compute the dependencies between facets. In our solution, we use chi-square test for independence that requires as input the contingency table between the two pairs of attributes for which we want to measure the dependency. The count summary that is returned by faceted search cannot be used to compute this chi-square test. (Most likely it won't be sufficient for any other feature selection either). In Table 3.2, we show a sample contingency table. Faceted search returns only the information that is shown in the last row and last column of the table. But to compute the chi-square test we need the complete table information, which can only be obtained from the actual result $\mathcal{R}_{\mathcal{V}}$. It cannot be easily obtained by performing union and intersection of posting lists.

For our chi-square based solution to problem 1.2, we need the total count of each

	2WD	4WD	AWD	Row Total
Sedan	5000	0	200	5200
SUV	900	800	500	2200
Column Total	5900	800	700	7400

Table 3.2: Sample contingency table for attributes `BodyType` and `Drivetrain`

facet value in result set $\mathcal{R}_{\mathcal{V}}$ and $\mathcal{R}_v$ inorder to compute all the observed and expected counts. When users hover over facet value v, we can get the count information for the result set $\mathcal{R}_v$ by an additional internal query to the faceted search engine.

3.5.2 ICSummary Using Additional Queries

In this section, we present two simple algorithms to create ICSummaries, which are based on additional internal queries to the search engine. We assume that the user currently has the result set $\mathcal{R}_{\mathcal{V}}$, and as the user hovers over facet value v, we get the count information about result set $\mathcal{R}_v$ through an additional query to the search engine, and show the computed ICSummary for facet value v.

As discussed later in evaluation section (in Section 3.6), the computation time to retrieve large number of tuples from the search engine and the the time to compute the dependent facets is orders of magnitude more compared to querying the faceted search engine and retrieving few tuples.

Precomputed Dependent Facets based Algorithm — Since retrieving the tuples is required mainly to compute the dependent facets, we can avoid both the above costs by simply showing a precomputed set of dependent facets (based on the overall dataset) for each queriable facet. Using this algorithm, for each ICSummary we need to do an additional query to the search engine to summarize the values in each dependent facets, which would take an additional cost of 100–300 ms for most typical datasets.

Creating ICSummary with precomputed facets won't be very informative because

the list of dependent facets changes greatly with the user's explicitly selected facet values across all the facets. In other words, it depends on both $\mathcal{R}_\mathcal{V}$ and $\mathcal{V}$. We are not aware of any novel ways to reduce the time for retrieving the result set $\mathcal{R}_\mathcal{V}$. However, we reduce this cost by retrieving only a subset, say atmost 5K–10K tuples, from any result set. The slight delay in retrieving tuples won't effect user's interaction because the tuples are retrieved only when the user changes the query condition. When the user hovers over a facet value to see an ICSummary, we do not need to retrieve the result set $\mathcal{R}_v$. For creating ICSummary just the basic summary digest from $\mathcal{R}_v$ is sufficient.

Baseline Algorithm — Our second algorithm uses the result set $\mathcal{R}_\mathcal{V}$ (or its subset) to compute the dependent facets as required in Problem 1.1. We consider this as the baseline algorithm because it computes the exact ICSummary, as described in Section 3.4. For this one can use standard chi-square feature selection algorithm over the result set $\mathcal{R}_\mathcal{V}$. Since it is a supervised feature selection algorithm, we use the attribute corresponding to facet values $\mathcal{V}$ as the class attribute. In this algorithm, we can do various optimizations through caching and lazy evaluation. Computing dependent facets using chi-square feature selection is computationally costly, especially if the dataset is large or is high-dimensional. The set of dependent facets is the same for all the facet values $\mathcal{V}$ that are selected from the same facet. We can cache the dependent facets after we compute it once for any facet value within a facet. Moreover, we do not need to compute the dependent facets for all the queriable facets if the user is not interested to look at ICSummaries for those facets. Thus we compute the dependent facets using lazy evaluation. By using these two simple optimizations we can greatly reduce the overall cost to compute the dependent facets. We then use the same approach as in case of precomputed facets to summarize each dependent facets.

3.5.3 ICSummary Using Extended Summary Digest

In this section, we present a novel way of maintaining facet value counts in $\mathcal{R}_{\mathcal{V}}$ by which we can solve both Problems 1.1 and 1.2 by two orders of magnitude more efficienty compared to what we can achieve using the additional query based approach. We call this count summary as *Extended Summary Digest* (ESD) because it has more information compared to the traditional summary digest in faceted interface. Creating ESD takes additional 200ms–500ms, but it can reduce the time to solve problem 1.1 and 1.2 to just 2ms–10ms for any result size or dimensionality, as compared to 300ms–700ms that is required in the additional query based approaches. The gain is further more if the cost is amortized over many ICSummaries that users see per result set. All these are verified later in the evaluation section.

Extended Summary Digest — Let's assume that our current result set is $\mathcal{R}$ with n attributes $\mathcal{D} = \mathcal{F} \cup \mathcal{H}$. Our goal is to construct a count summary from $\mathcal{R}$ such that for any given attribute $d_i \in \mathcal{D}$ and a facet value $u \in d_i$, we should be able to efficiently compute the summary digest of the result set $\mathcal{R}_u$ in which only the facet value u is selected from attribute d_i, with all other selection conditions being the same as of result set $\mathcal{R}$. In multi-selection faceted interface the summary digest of different facet values get commingled and thus we cannot obtain the contingency table that is required in Problem 1.1. Instead of getting the summary digest by doing an additional query, we want to precompute in an efficient manner the summary digest for all the facet values for which the user might want to see the ICSummary. We use hash table with key as <facet, facet value> and the value as the corresponding summary digest. We process each tuple of the result set and update the counts in this hash table. We also maintain the summary digests in the form of hash tables to keep the insert and access time constant. One can see that the cost of creating this ESD is $\mathrm{O}(n^2|\mathcal{R}|)$, where n is the total number of attributes. In this algorithm we need to process each tuple exactly once and processing each tuple requires $\mathrm{O}(n^2)$ computations.

Optimized Extended Summary Digest — Computing the ESD as described above would be computionally costly if either $|\mathcal{R}|$ or n is large. As described in Section 3.5.2, we retrieve only at most 5K–10K tuples from any result set, for which the time to compute the ESD is less than 500ms for small n. However, if the dataset is high-dimensional, then the term n^2 can significantly increase the computation time. We can reduce this cost by using the following two heuristics: (1) Only compute the summary digest for attributes that are queriable facets, and (2) For each queriable facet compute the count of facet values for some preselected subset of attributes. Since users can select facet values from only queriable facets, we need to compute the summary digest for the facet values of only the m queriable facets. Again let's assume that we can show only l facets in any ICSummary, where $l << n$. If for each queriable facet we precompute the top-$2l$ dependent facets based on the overall dataset, we can assume that in most cases the dependent facets for any result set would be from these $2l$ facets. As a result, we can reduce the computation time from $O(n^2|\mathcal{R}|)$ to $O(ml|\mathcal{R}|)$. If we use this optimization, we might miss the valuable dependent facets in those cases where the result set $\mathcal{R}$ has significantly different distribution compared to the overall dataset. For example, if the selected facet values $\mathcal{V}$ have low relative count in the overall dataset, then the dependent facets based on the overall dataset may not be the best set of dependent facets for these facet values.

3.6 Evaluation

The benefits of ICSummary can be seen from the sample outputs. An ICSummary provides users an easy way to see the important interactions within and across facets, which is hard for them to observe in the traditional faceted interface. In this section, we mainly focus on evaluating the system performance. We need to compute too many ICSummaries for each result, where each ICSummary requires good amount of computational time. We have to make sure that they can be computed in reasonable

(interactive) time for the data set complexities and sizes that we expect. We discuss this issue in Section 3.6.2.

3.6.1 Experimental Setup

We have integrated ICSummaries with Apache Solr, using a backend server written in Java Servlet. We input users' query from faceted interface, compute the ICSummary in the backend server, and return it using HTML and Javascript. We use the chi-square algorithm available in Weka [46] to compute the dependent facets for the baseline algorithm described in Section 3.5.2. For computing the dependent facets in Section 3.5.3 we first of all construct the contingency table from the extended summary digest and then use the chi-square algorithm available in Apache Commons Math Library [4]. We also use other libraries from [4] to do the various statistical tests.

We used two real datasets—YahooUsedCar and Census [38]—to evaluate our system. We scraped Yahoo's used-car site [3] to create a table comprising 40,000 tuples with 11 attributes. The Census dataset has 2,458,285 tuples with 68 categorical attributes. These two datasets are at the lower and higher end of what one sees in a typical e-commerce dataset. In the Census dataset the number of attributes are towards the higher side. We show that optimizations that are proposed in Section 3.5.3 for computing the optimized ESD is important to get real time performance for datasets with large number of attributes.

3.6.2 Performance

To understand where the time goes in our system, we run our system on different datasets and on different sizes of query result. We categorize the computations in following two categories: *static computations* and *dynamic computations*. In static computations, we include all those computations that are done every time the user has

a new query result. For the same query result, as the user hovers over different facet values, we need to do some more amount of computation to create the ICSummaries. But this computation cost depends on the user's browsing pattern. We call these computations as dyanamic computations. We analyze these two computation costs in the following two subsections.

3.6.2.1 Static Computations

Every time the user has a new result set, we need to do the following three computations: retrieve tuples from the faceted search engine, compute the extended summary digest (ESD), and do some other computations such as transfer data over network. In our system, the first two steps, namely retrieving tuples and computing ESDs, are the two most computationally steps.

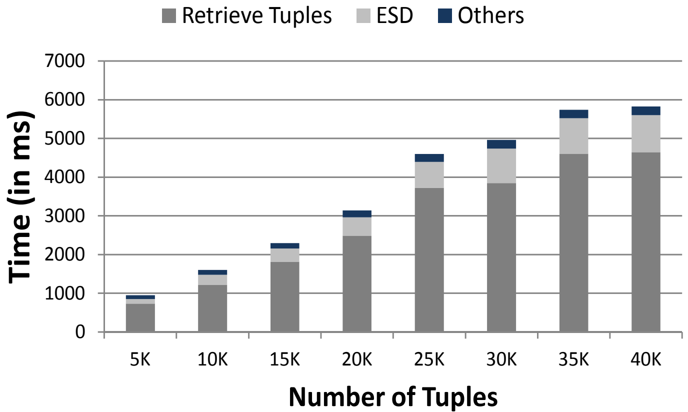


Figure 3.4: Time breakdown for static computations in the YahooUsedCar dataset, which is a small dataset with 11 attributes.

In Figure 3.4, we show the computation time for the three static computations in the YahooUsedCar dataset. One can see that in this figure the time to retrieve the tuples is computationally most expensive, followed by the time to compute extended summary digest. When we retrieve large number of tuples, the search engine has to do lot of computation to reconstruct the tuples from the index. But if we retrieve few

tuples, such as 10–50 tuples in a pagewise manner, then the time for these steps will be quite small (around 100–300 ms). As described in Section 3.5.2, we can reduce this total cost by retrieving only a small subset of say atmost 5K–10K tuples.

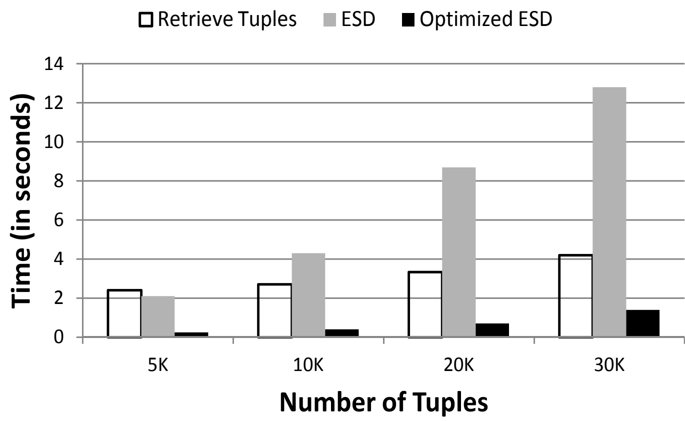


Figure 3.5: Time breakdown for static computations in the CENSUS dataset, which is a large dataset with 68 attributes. Effect of optimizations on computing the extended summary digest.

In Figure 3.5 we show the variation in computation time with different result size for the CENSUS dataset, which is very high-dimensional with 68 attributes. By comparing Figure 3.5 and Figure 3.4, we can see that the time to retrieve tuples and time to compute extended summary digest increases significantly with increase in number of attributes. In fact for this dataset as we retrieve larger number of tuples, the time to compute ESD becomes significantly greater compared to even the time to retrieve tuples. As discussed in Section 3.5.3, if we compute the extended summary digest in a naive manner then it will take $O(n^2|\mathcal{R}|)$ time. But if we do the optimizations as described in Section 3.5.3, we can reduce the computation time to $O(ml|\mathcal{R}|)$. In Figure 3.5, we use $m = 10$ and $l = 10$. This reduces the time to compute the extended summary digest by a factor of 20. Inorder to show top-l (here $l = 10$) dependents facets in the ICSummary, we need to count the facet values in the the top-$2l$ precomputed dependent facets. As a result, even theoretically one can see that computing the optimized ESD should be $(68^2/10 \times 20) \approx 23$ times more efficient.

3.6.2.2 Dynamic Computations

Algorithm	Find Dependent Facets (in ms)	Summarize Dependent Facets (in ms)
Precomputed Facets	0	130
Baseline	230	130
ESD	2	3

Table 3.3: Time breakdown for dynamic computations (or creating an ICSummary).

In Table 3.3, we show the comparison between the three ICSummary computation algorithms, discussed in Section 3.5, for creating a single ICSummary. These timings are the averages from YahooUsedCar dataset using 10K tuples size result set. We do not include the time for optimized ESD algorithm because this dataset only has 11 attributes. In the algorithm with precomputed facet order, there is no cost to compute the dependent facets. In this algorithm the main cost is the cost of additional query to the search engine. For this dataset, any query takes a time between 100–150ms if we just retrieve the summary digest and 20 result tuples. In the baseline algorithm, we compute the dependent facets using the standard chi-square algorithm [46] over the result tuples. For 10K tuples with 11 attributes, it takes an average of 250ms. Thus the total computation time for the baseline is $250 + 130 = 380$ms. If we use the extended summary digest to get the contingency table and then compute the dependent facets, then it takes just 2ms compared to 250ms using standard chi-square without contingency table. Using ESD the time to summarize all the dependent facets is around 1-3ms, which is two orders faster compared to doing an additional query.

The benefits of ESD would be further more if we look at the amortized cost over multiple ICSummaries that is seen per query result. For 10K tuples with 11 attributes computing the ESD takes around 250ms, which is a onetime cost per query result. If we assume that for a given query result, a user sees a total of N ICSummaries from M different queriable facets, then the total time computation time will be $130N$,

$(250M + 130N)$, $(2M + 3N)$ for the precomputed facet order, baseline and ESD based algorithms respectively. If we do not use the optimizations of caching and lazy evaluation while computing the dependent facets, then the computation time for baseline will become further worse, with comparatively much less impact on the ESD based algorithm.

Thus if we use ESD based algorithm, then users will see an additional increase of 250ms for each new query result, but this will make the display of ICSummaries seem much more real-time compared to additional query based approach that takes two orders of more computation time for each ICSummary. Maintaining ESDs will also lead to great reduction in server work load.

3.7 Conclusion

In this chapter, we introduced a novel add-on extension ICSummary to faceted search that can be used to see the pairwise interaction between attributes in complex datasets. ICSummary helps users to easily see the implicit selections that happen when they explicitly select a particular facet value. Understanding these implicit selections in other facets, including hidden facets, can help users to do selections in more informed manner. Although computing ICSummaries is computationally intensive, we present various algorithms and optimizations that enable it to be easily integrable with existing faceted interfaces, without compromising system performance.

CHAPTER IV

TPFacet: Two-Phased Faceted Browsing

4.1 Problem Overview

In Chapter III, we addressed some of the limitations in the query panel from users information loss perspective. Our main aim there was to assist users have a better understanding of their chosen exploration path through informative ICSummaries, as compared to simple tuple counts that are shown in the traditional faceted interface. Using ICSummaries we could show users how each facet value selection changes facet value distribution in other facets.

In this chapter, we address some more limitations of the query panel through another novel data summarization technique called the Summarized Facet Interaction (SFI) View. In the previous chapter, our goal was to give users a more detailed understanding of an exploration path that they are undertaking. Since in faceted navigation there are combinatorial number of exploration paths, users often take an inferior exploration path due to not understanding the space of available exploration paths. Users need help from the system in not only understanding a specific exploration path, but also to understand all the other available exploration paths. SFI View allows users to find and compare exploration paths systematically in facetwise manner.

There are limitations in the existing query panel from both the designers' and

the users' perspective. Faceted navigation is a type of data-categorization, with each facet value as a category label. To provide faceted navigation, designers have to create all the categories of interest in advance. Existing algorithms to automatically create categories are only partly successful [50]. As a result, users cannot find interesting and potentially unexpected or new trends in data. It is painful to manually create and maintain complex category structure. Furthermore, even if such a deep hierarchy of category labels were created, it cannot be shown in the query panel of faceted interface, since it will be too overwhelming to users.

When users are in an unfamiliar domain they have a limited understanding of what each facet value (category label) stands for. For example, in a car database, although a user might have vague understanding of what each facet value, such as SUV, Sedan, AWD, Ford, Chevrolet, etc., stands for, they may have limited understanding of the details within each facet value or what happens when they select multiple facet values. To do effective information exploration, a user must have a good understanding of dependencies between various facet values within and across facets. ICSummary can show pairwise interaction between facets. But as we will see in the next section that there are certain types of facet interaction that is not captured in ICSummaries. Since it is not necessary that a particular facet value will have a unique relationship with other facet values, a user would like to know what are all possible ways a given facet value is related with other facet values.

To address the above limitations, one can consider other data organization techniques that have complementary characteristics. Clustering is a widely used data organization technique for set level data exploration. It can show themes, and reveal unexpected or new trends in any data collection. A central property of clustering is that it depends on the data and is independent of the users' interest. The main disadvantage of clustering is that the result is unpredictable and not related to the users' specific exploratory goal.

When users are browsing in an e-commerce site, their final goal is to look at information at the instance (or tuple) level. Faceted browsing is very efficient for instance-level selection. But to do instance-level selection in an informed manner, users need to know the big picture (or set-level) information. Clustering provides set-level information, but it does so in a global manner and ignores selections already performed by the user. As such, standard clustering is not a useful complement to faceted search. Rather, we need some context-dependent notion of clustering, capable of providing set-level information to the user, conditioned on selections already applied, and helpful towards making additional selection choices.

In this chapter, we extend the traditional faceted interface by presenting a new two-phased faceted interface, known as TPFacet, that can provide users easy access to both instance and set level information. TPFacet shows the the set level information in the form of an SFI (Summarized Facet Interaction) View, which is created according to users' specific information need. We still retain the traditional faceted view so that users can explore information using two phases, first using one view and then the other.

The main contributions of this chapter are as follows:

- We identify three specific barriers that faceted navigation presents to users (Section 4.2).
- We propose an interaction and query technique called Summarized Facet Interaction (SFI) that effectively addresses each of the three barriers to using faceted navigation (Section 4.3).
- We present the algorithms and techniques necessary to create and present relevant SFI Views for users during navigation (Sections 4.4 and 4.5).
- We evaluate TPFacet on real data with a detailed user study. We find that users, on average, can perform tasks that require data understanding with 4-5

times greater efficiency and accuracy using our SFI View as compared to faceted interface (Section 4.6).

Finally, we conclude in Section 4.7.

4.2 Limitations in the Query Panel

In this section, we provide three concrete ways a faceted interface fails a typical user. These three barriers to successful use make up the desiderata that motivate our core technical contributions in the next section.

Let $\mathcal{D}$ be a relational database with n numerical or categorical attributes. Let $\mathcal{F} = \{f_1, f_2, ..., f_m\}$ be the set of m attributes that are shown as facets in the query panel, and the remaining $(n-m)$ attributes, denoted as $\mathcal{H} = \{h_1, h_2, ..., h_{(n-m)}\}$, be the set of attributes that are not shown as facets but are additional features shown for each result tuple. We call the sets $\mathcal{F}$ and $\mathcal{H}$ as *queriable* and *hidden* (or non-queriable) facets respectively.

Example IV.1. Reconsider Mary's example discussed in Example III.1 (see Chapter III). Furthermore, let's assume that Mary has narrowed her body style choices and is interested in one of the following five body styles: SUVs, Sedans, Convertibles, Hatchbacks and Coupes. Mary prefers SUVs, but she wants to make her decision based on how SUVs compare with the other four body styles. She wants to compare the different body styles in terms of features that she maximally cares about, such as `Price`. She also wants to compare other features that differentiate the different body styles before she commits to an SUV.

Mary's task highlights three distinct limitations of the faceted search interface. (Note: ICSummary is a partial solution for Limitation 1, but it cannot address Limitations 2 and 3)

Limitation 1. Comparing Facet Values — In the `Body Style` facet Mary can see all the available body styles, such as SUV, Sedan, Hatchback, etc., along with their tuple counts in the current query result. But a faceted interface does not provide any means to see the detailed difference or similarity between these body styles. The problem becomes more acute when facets have many values. For example, `Make` and `Model` have more than 50 and 200 facet values respectively.

Although faceted interface by itself is not designed for supporting comparison, users can use single selection to compare options. For example, Mary can have five separate windows where in each window she selects a particular body style with other query conditions being same. She can then compare the five body styles based on their summary digest. This kind of comparison is painful, particularly if one would like to compare many options. It is also limited to the counts available in the summary digest. One cannot use multi-selection option because it would commingle the list of facet values that are shown in other facets.

To allow comparison, some e-commerce sites provide users an additional interface to perform side-by-side comparison. In `cars.com` there is an option to compare different car models. Users can select a list of car models, and the system shows comparison across some fixed set of other attributes by showing simple statistical summaries, such as min-max range, mean etc. Since there are an exponential number of choices to compare and users are not even aware of what are the best options they should compare, even these additional interfaces provide very limited help to users.

Limitation 2. Finding Themes — Each facet value often represents a very large set of diverse items. For example, body style SUV has sub-categories Compact, Mid-size, Full-size etc. Even a single sub-category can have very diverse cars due to difference in car manufacturers and the specific car model. If Mary wants to compare the five body styles using the single selection option (as described above), she will not be able to see all these different sub-categories or themes within each body style. If we show simple

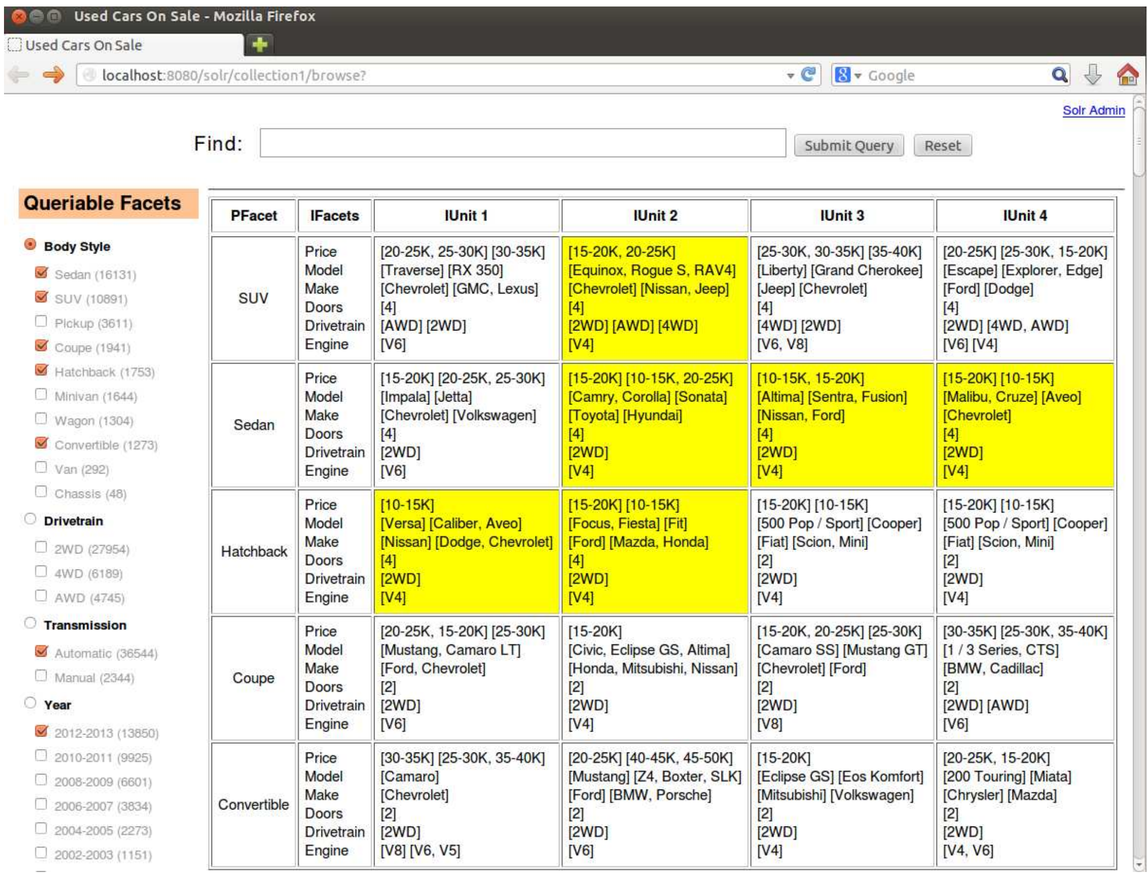
one column type of statistical summary (such as mean or median) for different body styles, it will seem that SUVs are quite costly compared to Sedans and Hatchbacks. Although there might be a small class of SUVs with price and other features similar to Sedans and Hatchbacks. This misleading summary may discourage Mary to explore SUVs further even though there are SUVs that meet her price criteria.

Limitation 3. Expressing Selections — In high-dimensional dataset, most of the dimensions are non-queriable or hidden facets. Users cannot express selection over such facets. For example, `cars.com` does not offer the number of cylinders in the engine as a queriable facet. If Mary wants to focus on V4 engines, she has no easy way to express that selection. Even worse, suppose Mary wants to choose a certain car body style, but this field is not encoded anywhere in the database, whether as a queriable or a hidden facet. There may be a way to express her preference as a selection on queriable facets (perhaps by indicating a certain combination of fuel efficiency and number of doors), but any such mapping from available queriable facets to her preferred dimension is entirely opaque.

All three limitations — Comparing Facet Values, Finding Themes and Expressing Selections — are addressed by our novel navigation technique, the Summarized Facet Interaction View. In the next section we will describe the SFI View in detail.

4.3 Summarized Facet Interaction

In this section we discuss our solution to the three limitations of faceted navigation for information exploration in complex databases. We describe our solution — the Summarized Facet Interaction View — in detail. We cover both its static and interactive qualities. We also identify the design and algorithmic challenges that will need to be solved in order for the SFI View to fulfill its goals.



PFacet	IFacets	IUnit 1	IUnit 2	IUnit 3	IUnit 4
SUV	Price Model Make Doors Drivetrain Engine	[20-25K, 25-30K] [30-35K] [Traverse] [RX 350] [Chevrolet] [GMC, Lexus] [4] [AWD] [2WD] [V6]	[15-20K, 20-25K] [Equinox, Rogue S, RAV4] [Chevrolet] [Nissan, Jeep] [4] [2WD] [AWD] [4WD] [V4]	[25-30K, 30-35K] [35-40K] [Liberty] [Grand Cherokee] [Jeep] [Chevrolet] [4] [4WD] [2WD] [V6, V8]	[20-25K] [25-30K, 15-20K] [Escape] [Explorer, Edge] [Ford] [Dodge] [4] [2WD] [4WD, AWD] [V6] [V4]
Sedan	Price Model Make Doors Drivetrain Engine	[15-20K] [20-25K, 25-30K] [Impala] [Jetta] [Chevrolet] [Volkswagen] [4] [2WD] [V6]	[15-20K] [10-15K, 20-25K] [Camry, Corolla] [Sonata] [Toyota] [Hyundai] [4] [2WD] [V4]	[10-15K, 15-20K] [Altima] [Sentra, Fusion] [Nissan, Ford] [4] [2WD] [V4]	[15-20K] [10-15K] [Malibu, Cruze] [Aveo] [Chevrolet] [4] [2WD] [V4]
Hatchback	Price Model Make Doors Drivetrain Engine	[10-15K] [Versa] [Caliber, Aveo] [Nissan] [Dodge, Chevrolet] [4] [2WD] [V4]	[15-20K] [10-15K] [Focus, Fiesta] [Fit] [Ford] [Mazda, Honda] [4] [2WD] [V4]	[15-20K] [10-15K] [500 Pop / Sport] [Cooper] [Fiat] [Scion, Mini] [2] [2WD] [V4]	[15-20K] [10-15K] [500 Pop / Sport] [Cooper] [Fiat] [Scion, Mini] [2] [2WD] [V4]
Coupe	Price Model Make Doors Drivetrain Engine	[20-25K, 15-20K] [25-30K] [Mustang, Camaro LT] [Ford, Chevrolet] [2] [2WD] [V6]	[15-20K] [Civic, Eclipse GS, Altima] [Honda, Mitsubishi, Nissan] [2] [2WD] [V4]	[15-20K, 20-25K] [25-30K] [Camaro SS] [Mustang GT] [Chevrolet] [Ford] [2] [2WD] [V8]	[30-35K] [25-30K, 35-40K] [1 / 3 Series, CTS] [BMW, Cadillac] [2] [2WD] [AWD] [V6]
Convertible	Price Model Make Doors Drivetrain Engine	[30-35K] [25-30K, 35-40K] [Camaro] [Chevrolet] [2] [2WD] [V8] [V6, V5]	[20-25K] [40-45K, 45-50K] [Mustang] [Z4, Boxter, SLK] [Ford] [BMW, Porsche] [2] [2WD] [V6]	[15-20K] [Eclipse GS] [Eos Komfort] [Mitsubishi] [Volkswagen] [2] [2WD] [V4]	[20-25K, 15-20K] [200 Touring] [Miata] [Chrysler] [Mazda] [2] [2WD] [V4, V6]

Figure 4.1: This screen capture shows a sample SFI View shown during query revision phase. It shows the top-4 IUnits for each of the five body types the user has selected in the Pivot Facet `BodyType`. The user has also selected `Year` $\geq$ 2012 and `Transmission` = Automatic.

4.3.1 The SFI View

To address the challenges described above, we introduce a novel *Summarized Facet Interaction* (SFI) View. The SFI View is a new component that we propose adding to standard faceted navigation (in addition to the existing query, result, and summary digest components). It can be seen as an extension of the query panel whose role is to provide deeper insight into the information shown in the query panel based on the current result set.

4.3.1.1 Overview

Figure 4.1 shows a sample SFI View, obtained from a real dataset, for the example query discussed in Section 4.2. Mary's goal was to compare cars of five different body styles with Automatic `Transmission` and `Year` $\geq$ 2012. To get to this SFI View, Mary selects `Year` as 2012-2013, `Transmission` as Automatic, and the five facet values in the `Body Style` facet using multi-selection option, as in a traditional faceted interface. The SFI View has several important components:

1. The Pivot Facet organizes the information that is shown in the SFI View. A user explicitly chooses one of the facets in the set of queriable facets $\mathcal{F}$ as *Pivot Facet* f_p and requests the system to create an SFI View that facilitates comparison among facet values selected from the Pivot Facet by showing their relationship with values across other facets. In Figure 4.1, Mary has chosen Body Style as the Pivot Facet.

2. Informative Facets are data facets (also: dimensions, or attributes) that interact with the Pivot Facet in "interesting" ways. Interestingness of a facet should be determined based on the result set and the Pivot Facet. For example, one can use correlation to quantify interesting interaction. Although the traditional faceted interface treats queriable and hidden facets differently, we treat them equivalently while determining the list of IFacets. In Figure 4.1, the system has given six IFacets:

Price, Model, Make, Doors, Drivetrain, and Engine.

3. An IUnit is an "interesting" group of values for the informative facets. In Figure 4.1, each IUnit is described using the six IFacets mentioned above. Each IUnit is chosen to be relevant to a Pivot Facet value: SUV, Sedan, Hatchback, etc. The top-left IUnit in the figure (containing Traverse and RX 350) identifies a set of midsize SUVs: they share an engine size and a drivetrain, and have similar prices. One can think of an IUnit as a cluster of database values with two special differences: it is a cluster on a partition of the database determined by each Pivot Facet value, and the cluster is labeled using the chosen IFacet labels and IFacet values.

The Overall SFI View is a tabular combination of these three interface elements. It displays one row for each user-selected Pivot Facet value. (The Pivot Facet itself is also explicitly chosen by the user.) In the second column the system chooses and shows an ordered list of Informative Facets, one for each row of the table. The rest of the table shows each row's top IUnits, sorted left-to-right in descending order of statistical relevance to the row's Pivot Facet value. If an IUnit cluster can be represented equally well by multiple values in a single IFacet, then an IUnit will show multiple facet values in square brackets (*e.g.*, Traverse and RX 350).

4.3.1.2 Interaction Support

Users select multiple values in the same Pivot Facet because they want to perform common browsing tasks such as compare-and-contrast, find-similar-options, etc. Our system can help the user with a number of these tasks by supporting two interaction patterns in particular: (i) Finding similar top ranked IUnits, and (ii) Finding similar facet values within the Pivot Facet.

For example, if a user likes a particular IUnit from one of the selected Pivot Facet values (*e.g.*, SUV), then the user may want to efficiently locate similar top-ranked IUnits that belong to other Pivot Facet values. Similarly, if the user likes multiple

IUnits of a particular Pivot Facet value, then the user might be interested to find out other Pivot Facet values that have similar IUnits.

Let $\mathcal{V}$ denote the set of user selected Pivot Facet values. Because the user selects $|\mathcal{V}|$ such values, and the top-k IUnits are shown for each of those selected values, the SFI View will present the user with $k|\mathcal{V}|$ IUnits to browse. The concept of browsing through IUnits is introduced for the first time in this chapter, so there is no existing work that describes how users might typically interact with them. However, it is possible that even after the system constructs the best possible SFI View, the user could still benefit from explicit system guidance on which of the $k|\mathcal{V}|$ IUnits to browse next.

Therefore, in addition to building the SFI View structure, we use interactive information visualization techniques, such as dynamic highlighting and reordering of rows in the SFI View, to facilitate these two interactions patterns.

4.3.1.3 Design Goals

We can now examine the extent to which the SFI View addresses each of our desiderata:

Limitation 1. Comparing Facet Values — With traditional faceted navigation Mary was unable to compare the five body styles. Using the SFI View one can easily compare all the five body styles in terms features that primarily differentiate the body styles. For example, it is a well-known fact that Coupes and Convertibles are very similar body styles. A user unaware of this fact could learn this similarity through their very similar IUnits.

Limitation 2. Finding Themes — Recall that Mary had higher preference for SUVs. But she was not able to find the sub-categories of SUVs that is similar to cheaper body styles, such as Sedans or Hatchbacks. The SFI View can help her to easily overcome the problem. Mary can see the different sub-categories of SUVs in

the form of IUnits. IUnit 1 represents midsize SUVs, IUnit 2 represents compact SUVs, and so on. The IUnit 2 of SUV is very similar to many IUnits in Sedans and Hatchbacks. When Mary clicks on IUnit 2 of SUV, the interface highlights all other IUnits that are similar to the clicked IUnit. This includes many IUnits in Sedans and Hatchbacks (shown shaded in Figure 4.1).

Limitation 3. Expressing Selections — Also recall that Mary was unable to choose cars with V4 engines, because the interface did not expose Engine type as an option in the query panel even though the information was contained in the database (*i.e.*, Engine was a *hidden* facet). Moreover, she was not familiar enough with the database to indirectly find V4 engines by selecting values in the *queriable* facets. In contrast, the SFI View identifies V4 engines as a characteristic of specific IUnits for each body style. Mary can select the desired tuples using either the corresponding queriable facets or through the IUnits.

4.3.1.4 Design Challenges

There are design challenges in presenting the SFI View, the set of IFacets and the top-k IUnits. In this section, we discuss those challenges and then present our solution.

To fit the SFI View within users' limited screen space, we propose a slightly changed interaction model for faceted navigation: at any one time, the interface will display either the results panel *or* the SFI View. The user explicitly toggles between them, though it is easy to imagine a system that intelligently chooses a default view, based on the size of query results. We imagine the user will interact with the system in two distinct phases: the *query revision* phase focuses on the SFI View, while the *result set* phase focuses on the results panel, with the user exploring individual items of interest in the result set.

If n is the total number of facets and c is the number of IFacets presented to the

user, then there are $\binom{n}{c}$ different possible IFacet choices. Using SFI View, Mary was able to overcome Limitation 3 because the system had luckily chosen `Engine` as an IFacet. To address the situation where the system may not choose the user's desired facet(s) as IFacet(s), we can allow the user to explicitly select facet(s) that the user would like to see as IFacet(s). For example, a user might want to see `Price` as an IFacet for all possible choices of Pivot Facet.

Note that there are competing ways to rank IUnits from left-to-right within each row. They can be ranked left to right in order of their salience for the row's Pivot Facet value. Or we could try to ensure that all of the IUnits in a single column can be compared across all Pivot Facet values so that, *e.g.*, the IUnit 1 for SUV is similar to IUnit 1 for Sedan (and thereby addressing Limitation 1).

However, not all Pivot Facet values may share comparable IUnits, forcing our system into an impossible tradeoff between IUnit quality and columnar IUnit "comparability." Thus, we chose to rank IUnits strictly by their relevance to the row's Pivot Facet value. We support that goal through user interaction with the SFI View, as described in Section 4.3.1.2.

4.3.2 Algorithmic Challenges

The SFI View is a complex structure. Our goals are (i) to populate this structure effectively, making the most of limited screen real estate available, and (ii) to arrange and present the information populated in this structure to maximize its value to the user.

For the first goal, we have to find the best (*i.e.*, most informative) IFacets, the best IUnit clusters, and (for each IUnit) the best value labels to describe the IUnit's data.

The SFI View structure already lays out IUnits in rows, one per facet value for the Pivot Facet. For the second goal, the system must further decide how to order

IUnits within each row, how to indicate similarity between IUnits in different rows, and how to indicate similarities and differences between rows as a whole.

In this section we will describe how we can model the above goals such that almost all steps are cast as standard data summarization or machine learning problems. We will also highlight the few steps that remain, describing our new algorithms for these steps in the next section.

4.3.2.1 The Static SFI View

An SFI View is created in the context of an ongoing navigation session, so there is always a set of selected facet values (this set could be empty) and an associated result data set $\mathcal{R}$. Additionally, we require the user to specify the Pivot Facet. We will show in the SFI View any values from the selected Pivot Facet that are present in the result set.

Populating the SFI View entails two main sub-tasks: obtaining the IUnits of interest, and labeling these IUnits. Each of these sub-tasks can be further divided into two distinct problems, as we shall see below. The width of the screen dictates how many IUnits we can show for each facet value. Let this number be k. Our sub-task then could be accomplished as finding k clusters with our favorite clustering algorithm. However, we find that results are much better if we solve the clustering problem with a larger number l, and then choose the top k IUnits from among these l clusters. Therefore, our sub-task can be written as:

Problem 1 (Generate Candidate IUnits): *Given a result set $\mathcal{R}$, a pivot-facet f_p, a set of facet values $\mathcal{V}$ selected from f_p, and a threshold value l, find for each facet value $v \in \mathcal{V}$ a list of l IUnits S^v, where $S^v = \{s_1^v, s_2^v, ..., s_l^v\}$ and s_j^v is the jth IUnit for facet value v.*

Since TPFacet is designed for user interaction, it has strict latency requirements. It might be possible to automatically choose l by iterating through all plausible l

values and evaluating the quality of the resulting SFI View (just as some machine learning algorithms perform search to infer their seed parameters), but doing so would typically be computationally costly. To avoid this step, one can use some simple heuristic, such as $l = 1.5k$. We recommend this heuristic based on our experience with various datasets.

The quality of an SFI View depends critically on the quality of its IUnits. So that users can compare multiple IUnits, they are labeled with the same set of IFacets. Thus we must not only identify good IUnits, but also identify good representative IFacet values for each IUnit. Thus in order to solve Problem 1, we need to solve the following two sub-problems:

Problem 1.1 (Informative Facets): *Given a result set $\mathcal{R}$, a Pivot Facet $f_p \in \mathcal{F}$, and set of facet values $\mathcal{V}$ selected in f_p, find a subset of informative facets $\mathcal{I}$ s.t. $\mathcal{I} \subseteq \mathcal{F} \cup \mathcal{H}$ and $\mathcal{I}$ generate the most contrast among values in $\mathcal{V}$.*

When there are multiple facet values within a facet, users often want to see the comparison between choosing one facet value vs another. To support comparison between facet values we choose informative facets that can show contrast among facet values $\mathcal{V}$. Choosing IFacets is thus a feature selection problem [45, 82] with a specialized way of evaluating the quality of a feature: good features (that is, IFacets) yield sharply contrasting IUnits across the different Pivot Facet values. One can discriminate among IFacets as follows: Given a multi-class problem, a feature X is preferred to another feature Y if X induces a greater contrast between the multi-class conditional probabilities than Y. X and Y are indistinguishable if they induce the same amount of contrast.

Problem 1.2 (Important Facet Interactions): *Given a set of informative facets $\mathcal{I}$, a subset of tuples $\mathcal{R}^v$ s.t. $\mathcal{R}^v \subseteq \mathcal{R}$ that satisfies a given facet value $v \in \mathcal{V}$, find the top-l facet interactions (or IUnits) S^v in $\mathcal{R}^v$.*

In Problem 1.2, we define the top-l facet interactions in terms of coverage. How-

ever, displays are limited in size, and users are limited in the number of IUnits they want to consume, we cannot show all l facet interactions to the user. The IUnits could be ranked based on a function that is rooted in the clustering algorithm; for example, we could prefer "tight" clusters by ranking them in ascending order of minimum pairwise similarity.

However, we can pursue some application-specific goals by ranking IUnit clusters in a manner that is distinct from the IUnit creation mechanism. For example, our car navigation interface might, by default, rank clusters in ascending order of cluster price. In contrast, the fleet manager for a taxi company might have a different preference function that ranks IUnits in descending order of car mileage.

Thus, our second problem is:

Problem 2 (Top-k IUnits): *Given a list of IUnits S^v for facet value v, and a preference function P, find the top-k IUnits T^v in S^v according to preference P, where $T^v = \{t_1^v, t_2^v, ..., t_k^v\}$ and $T^v \subseteq S^v$.*

4.3.2.2 The Dynamic SFI View

The two user interactions can be stated as following two problems:

Problem 3 (Similar IUnits): *Given two facet values x and y from the Pivot Facet, and an IUnit t_i^x from T^x, find all IUnits t_j^y s.t. $t_j^y \in T^y$ and $\mathsf{sim}(t_i^x, t_j^y) \geq \tau$.*

Intuitively, given two Pivot Facet values, and an IUnit from the top-k list of one of the Pivot Facet values, our goal is to find similar IUnits from the top-k list of the other facet value. For example, in Figure 4.1 IUnit 2 of SUV is very similar to IUnit 2, 3 and 4 of Sedan and IUnit 1, 2 of Hatchback. We say that two IUnits are similar provided their similarity is greater than a user-defined threshold value τ. Once the similar IUnits are highlighted, users can easily compare and determine their most suitable choice.

Problem 4 (Similar Facet Value): *Given two facet values x and y and their top-k list of IUnits T^x and T^y, find the similarity between x and y by measuring the similarity of their top-k IUnits.*

In the SFI View, users see the top-k list of IUnits for each facet value. If a user shows preference for a particular facet value, it implies that the user has liked most of the top-k IUnits that has been shown for that facet value. Users would be interested to see which other facet values have very similar IUnits. Although it is possible to find similarity between two categorical facet values using some instrinsic similarity measures, in this chapter, we measure their similarity based on their IUnits. Users will find it useful if the similarity between facet values is computed based on the IUnits that are being shown to the user, rather than some internal measure. Users trust a recommendation system when there is a visible proof of why certain option is recommended by the system. To compare the two top-k lists, we need a metric that measures similarity between IUnits of the two lists both in terms of content and rank.

4.4 The Static SFI View

In this section, we describe how we create and sort IUnits (problems 1 and 2 above) for the SFI View.

4.4.1 Generating Candidate IUnits

Generating uniformly labelled IUnits consists of two steps: finding good IFacets $\mathcal{I}$ that are most informative when distinguishing all the Pivot Facet values $\mathcal{V}$; and then generating l IUnits for each value $v \in \mathcal{V}$.

4.4.1.1 Finding Informative Facets

The problem of finding informative facets is similar to feature selection in a multi-class classification problem. Since our goal is to provide efficient user interaction and

understanding, and not to optimize a specific mining algorithm, we use a feature selection algorithm that is computationally efficient and returns all relevant features.

To determine the number of informative facets we consider two factors: the available screen space and the relevance score of each informative facet. The user's available screen space determines the maximum number c of informative facets that can be shown for any Pivot Facet. However, all Pivot Facet may not have c informative facets that have relevance greater than a required minimum threshold relevance score. A relevant informative facet always provides additional useful information. However, if a facet is not informative about the Pivot Facet, including it will lower the quality of generated IUnits and waste valuable screen space.

Pivot-Facet	**Informative Facets**
`Make`	`Model, BodyStyle, Engine, Drivetrain`
`Model`	`Make, BodyStyle, Engine, Drivetrain`
`Price`	`Year, Engine, Make, Model`
`Year`	`Model, Make, Price, Mileage`
`Mileage`	`Year, Price, Model, Make`
`BodyStyle`	`Model, Make, Drivetrain, Doors, Engine`
`Drivetrain`	`Model, BodyStyle, Make, Engine, Price`

Table 4.1: Sample informative facets for used-car dataset.

We use the chi-square feature selection algorithm [88]. Chi-square evaluates the worth of an attribute by computing the value of the chi-squared statistic with respect to the class. For chi-square test we can determine the threshold relevance using p-values, such as significance level equal to 0.01, 0.05, or 0.10. Table 4.1 shows, for each of several Pivot Facets (where the user has chosen *all* of the possible Pivot Facet values), a list of informative facets emitted by the chi-square algorithm. The IFacets are ranked by decreasing relevance from left-to-right. Even with this simple technique, ranking IFacets in order of decreasing relevance yields a few interesting observations that a typical user might not know. For example, it might seem that `Mileage` should be the best IFacet when distinguishing among different `Year` values: older cars will

naturally accrue more miles. However, it turns out that Model is better, as specific car models (Suburban 1500 LT, not simply Suburban) are released frequently, and a specific model is prominent in the database for only a short period of time.

4.4.2 Finding Important Facet Interactions

To create IUnits for a Pivot Facet value $v \in \mathcal{V}$, we take all tuples from the result set $\mathcal{R}$ that contain the given value v, and allocate those tuples to l clusters. We derive an IUnit from each of these l clusters. We cluster the tuples using only the above-chosen IFacets.

Since TPFacet is a user-facing application, we want to create the whole SFI View within interactive time limits, such as 0.5-1 secs. There are various factors that can slow down a clustering algorithm: (i) clustering a dataset with large numbers of tuples or dimensions, (ii) trying to infer the ideal number of clusters using the clustering algorithm, and (iii) clustering with large numbers of cluster centers. Since there are standard existing techniques to address each of these factors, we defer their discussion to experimental evaluation (Section 4.6.4).

The quality of IUnits depends on the quality of the clustering algorithm. Since both efficiency and quality are major concerns of our system, we use standard k-means algorithm. Our main contribution in the clustering step is the dynamic variation of system parameters to achieve real-time performance, as discussed later in Section 4.6.4.

Our key contribution in creating the IUnits is the post-clustering step of cluster labeling, which is often ignored in clustering research. Although clustering is a very nice data-categorization technique, it is not a popular data-exploration tool for normal users because it lacks structured presentation. It is very hard for most users to understand the large amount of information that is contained in each cluster, or be able to compare multiple clusters.

The problem of cluster labeling has been more deeply studied in relation to text documents as compared to structured data. In text clustering, the labels often consist of important terms from the clustered documents that not only summarizes the cluster but also differentiates a given cluster from other clusters [29, 88]. This kind of term based labeling is also available in Apache Solr. Since text documents have large amount of information and are of very diverse topics, users can often locate the relevant clusters by just seeing few keywords that differentiates each cluster from other clusters. However, this approach is not suitable for structured data because each tuple has very limited amount of information. (All tuples have the same set of attributes and each attribute has a small range of attribute values). In the rest of this subsection, we present a summary of existing cluster labeling techniques for relational data, their limitations, and finally our proposed cluster labeling technique.

There are existing systems to visually explore clusters of structured data [26, 81, 111]. Some of these systems are not easy to explore when data is high-dimensional or categorical. For normal end-users, the commonly used cluster labeling technique is to show the centroid of each cluster. Showing centroid is useful when all clusters are spherical. For complex shaped clusters, it is considered more informative to show multiple tuples that can show the whole cluster boundary [26]. It is very hard to understand a high-dimensional cluster by seeing just one centroid or some boundary points.

The way we label the clusters has many benefits. We label all IUnits uniformly and use ranking at all levels. We rank the IFacets to highlight what information is most significant. Similarly, in each IUnit we rank the IFacet values and show only the most important representatives. Instead of showing few representative tuples from each cluster, we try to summarize statistical distribution of each IFacet. One major drawback of clustering is that it can merge distinct clusters. The way we create the IUnits, even if different clusters are merged, users may still be able to see the merge.

For example, in IUnit 2 of Convertible we can see that there are two clusters that have got merged because of the number of clusters we had created. Although based on our feature, the cars of Ford look similar to that of BMW and Porsche, but they differ significantly in terms of price. Moreover, one can understand that most the cars in this IUnit are from Ford, with few from other manufacturers. Our labeling allows users to see the sub-categories even within the same IUnit, which is not feasible with simple min-max range type of summarization. To label both categorical and numerical attributes in uniform manner, we discretize the numerical attributes. We rank facet values based on frequency count and then group multiple values if they have similar distribution. We use two thresholds — max display count and statistical difference between frequency counts — to determine the representative IFacet values for each cluster.

4.4.3 Selecting Top-k IUnits

Without an explicit user preference function, we choose a preference function that depends on the size of the IUnit's underlying cluster, as well as overall result diversity. IUnits that represent large clusters are desirable because they summarize facet interactions for larger number of tuples. Moreover, they may give more reliable insight than smaller outlier-prone clusters. However, when we select the top-k IUnits based purely on the cluster size, many are quite similar and appear redundant to the user.

Thus we use the generic top-k algorithm proposed in [96] to compute diversified top-k IUnits. It requires the following measures: preference score of each IUnit s_i^x, denoted as $\mathsf{score}(s_i^x)$; similarity between between two IUnits s_i^x and s_j^y, denoted as $\mathsf{sim}(s_i^x, s_j^y)$; and a user defined threshold similarity value τ. Two IUnits s_i^x and s_j^y are considered similar, denoted as $s_i^x \approx s_j^y$, if $\mathsf{sim}(s_i^x, s_j^y) \geq \tau$.

Diversified Top-k IUnits: *Given a list of IUnits* $S^v = \{s_1^v, s_2^v, ...\}$ *for a facet value*

v, and an integer k, the diversified top-k IUnits for v, denoted as $T^v = \{t_1^v, t_2^v, ...\}$, is the list of IUnits that satisfy the following conditions:

1) $T^v \subseteq S^v$ and $|T^v| \leq k$

2) For any two different IUnits s_m^v and s_n^v, if $s_m^v \approx s_n^v$ then $\{i_m^v, i_n^v\} \not\subseteq T^v$

3) $\Sigma_{t_i^v \in T^v}$ $\mathsf{score}(t_i^v)$ is maximized.

To create the SFI View, we need to compute the diversified top-k IUnits for each facet value v. The diversified top-k problem can be reduced to the NP-Hard maximum independent set problem on graphs [96]. Greedy solutions often lead to good approximate results in many NP-Hard problems, but for this problem a greedy algorithm can lead to arbitrarily bad solutions, with no bounded constant factor solution [96]. Because in our problem the size $|S^v|$ is generally not large, Qin, *et al.'s* basic `div-astar` algorithm works well.

4.5 The Dynamic SFI View

In this section, we decribe how to find similar components in the SFI View. These are solutions to Problems 3 and 4.

4.5.1 Highlighting Similar IUnits

When a user clicks on one of the IUnits, say IUnit t_i^x from T^x, we highlight all IUnits t in the SFI View s.t. $t_i^x \approx t$ (in other words, $\mathsf{sim}(t_i^x, t) \geq tau$). This approach allows us to address the IUnit sorting problem mentioned in Section 4.3.1.1; we can now sort IUnits from left-to-right by order of salience to the row's Pivot Facet value, while still allowing the user to compare similar IUnits.

Computing similarity between IUnits is equivalent to computing similarity between clusters. For a numerical dataset, one can compute cluster distance by measuring the distance (such as Euclidean distance) between cluster centroids. For a cate-

gorical dataset, one can use any distance measure that is used in existing categorical clustering algorithms to compute cluster distance [44]. However, things become more complicated when we have a mixed dataset, having both numerical and categorical attributes. The distance measure that is used in categorical dataset is quite different compared to those used in numerical dataset. To compute similar IUnits, we propose a new distance measure that can treat both numerical and categorical attributes in a uniform manner. We use discretization to convert numerical attributes into categorical attributes. Then we use a modified form of *cosine-similarity* to compute IUnit similarity.

Algorithm 1 IUnit Pair Similarity

Input: t_i^x: IUnit 1
t_j^y: IUnit 2
$\mathcal{I}$: set of informative dimensions
Output: s: similarity between the two IUnits
Method:
1: $s \leftarrow 0$
2: **for all** $d \in \mathcal{I}$ **do**
3: $s \leftarrow s + \textit{cosine-similarity}(t_i^x.d, t_j^y.d)$
4: **end for**
5: return s

Let t_i^x and t_j^y be two top-k IUnits for selected facet values x and y, and $\mathcal{I}$ be their set of IFacets. We measure the similarity of t_i^x and t_j^y by summing their cosine similarities along each dimension d s.t. $d \in \mathcal{I}$. We use the frequency count of each facet value in the corresponding cluster as the facet value's term frequency. Since the range of *cosine-similarity* function is $[0, 1]$, the range of the above similarity function is $[0, |\mathcal{I}|]$. Based on a specific data domain, one can choose the IUnit similarity threshold value τ as some $\alpha.|\mathcal{I}|$, where $\alpha \in (0, 1)$.

4.5.2 Finding Similar Facet Values

When a user clicks on one of the Pivot Facet values in the SFI View's leftmost column, we reorder all the rows so that the clicked value becomes the topmost row, and all other rows are shown in decreasing order of similarity to the clicked (now top-ranked) value. Two facet values are considered similar if their top-k IUnits lists are similar. Two ranked IUnit lists T^x and T^y should be similar if they have similar IUnits, and similar IUnits have similar rank.

To the best of our knowledge, there is no existing distance metric to compute similarity between two ranked lists having disjoint set of items. In Algorithm 2, we propose a distance measure that can compute distance between two given ranked lists by taking into consideration the similarity between their items both in terms of information content and rank.

Algorithm 2 Facet Value Pair Similarity

Input: $T^x = \{t_1^x, t_2^x, ..., t_k^x\}$ top-k IUnits for facet value x
$T^y = \{t_1^y, t_2^y, ..., t_k^y\}$ top-k IUnits for facet value y
Output: d: distance between T^x and T^y
Method:

1: $d \leftarrow 0$
2: **for all** $t_i^x \in T^x$ **do**
3: **if** $\exists t \in T^y$ *s.t.* $t \approx t_i^x$ **then**
4: $index \leftarrow j$ s.t. $t_i^x \approx t_j^y$ and $\underset{j}{\text{argmin}} |j - i|$
5: **else**
6: $index \leftarrow |T^y| + 1$
7: **end if**
8: $d \leftarrow d + |i - index|$
9: **end for**
10: **for all** $t_j^y \in T^y$ **do**
11: **if** $\exists t \in T^x$ *s.t.* $t \approx t_j^y$ **then**
12: $index \leftarrow i$ s.t. $t_i^x \approx t_j^y$ and $\underset{i}{\text{argmin}} |j - i|$
13: **else**
14: $index \leftarrow |T^x| + 1$
15: **end if**
16: $d \leftarrow d + |j - index|$
17: **end for**
18: return d

In lines 2-9, we compare how IUnits in T^x compare to IUnits in T^y. In line 3, we check whether the list T^y has some IUnit which is similar to IUnit t_i^x from T^x. If there is no similar IUnit (line 6), we assume that t_i^x is similar to the IUnit that has highest rank amongst non-selected IUnits (i.e., $S^y \setminus T^y$), and thus has rank $|T^y| + 1$. In lines 3-4, if there are multiple IUnits in T^y that are similar to t_i^x, then we take the IUnit whose rank is closest to rank of t_i^x in T^x, which is i. In line 8, we sum the rank differences for all IUnits in T^x. Lines 10-16 show the same steps for list T^y.

The above distance measure only considers ranking of similar elements, and not the exact degree of similarity. We had considered another distance measure that combines rank distance with similarity by taking the rank of the most similar element from the other list. We chose the one presented in Algorithm 2 because for the two datasets we had evaluated our system, it seemed to give more intuitive results when compared with human expectations. For example, there are well-known facts about the car domain, such as similarity between different `Makes`, etc., and these facts were more correctly captured by our chosen distance measure.

4.6 Evaluation

The goal of TPFacet is to facilitate users' information exploration of a complex dataset through a faceted interface. As such, the primary evaluation of TPFacet is by means of a user study. We perform two types of user study: *quantitative* and *qualitative*. In the quantitative user study, discussed in Section 4.6.2, we compare the use of TPFacet with a standard faceted interface for three information exploration tasks. These tasks are designed in such a way that we can numerically compare the benefits of the two interfaces. To do the quantitative user study, we required users to have some amount of technical expertise to understand the quantitative measures. In order to show the benefits of TPFacet for even normal end-users, we perform an observational (qualitative) study in Section 4.6.3. This observational study shows

that even normal users can gain better insights into complex datasets by using the TPFacet interface as compared to the traditional faceted interface. As a baseline for comparison, we use Apache Solr [1], which is a popular open source enterprise search platform. Apache Solr has support for faceted navigation and is used by many e-commerce sites. Apache Solr has many configuration settings. We chose a setting that is closest to the TPFacet system, except the SFI View.

A secondary question is one of performance. Since the summaries required by TPFacet are quite complex, we have to make sure that they can be computed in reasonable (interactive) time for the data set complexities and sizes that we expect. We discuss this issue in Section 4.6.4.

4.6.1 Experimental Setup

Our TPFacet system is integrated with Apache Solr, using a backend server written in Java Servlet. We input users' query from faceted interface, compute the SFI View and all similarity scores in the backend server, and return the resulting SFI View and similarity information using HTML and Javascript. To do feature selection and clustering, we use chi-square and SimpleKMeans algorithm respectively. Both algorithms are available in Weka [46].

We used two real datasets—YahooUsedCar and Mushroom [38]—to evaluate TPFacet. We scraped Yahoo's used car site [3] to create a table comprising 40,000 tuples with 11 attributes. The Mushroom dataset has 8124 tuples with 23 attributes. These numbers are at the lower end of what one sees in a typical e-commerce dataset. TPFacet will become more valuable in datasets that have more number of attributes or tuples. The Mushroom dataset is very popular in machine learning. It is simple to understand for a non-expert, since it describes familiar properties, such as color and smell, but has data that most of us (and all of our users) have no knowledge of, forcing us to learn patterns by examining the data set afresh without reliance on

previous knowledge.

4.6.2 Quantitative Analysis of User Study

We devised a diverse set of carefully specified information exploration tasks, described in the subsections that follow, each of which tests (some aspects of) the users' understanding of the database. These tasks roughly correspond to the three motivating limitations discussed in Section 4.2. We used the Mushroom data set, which was unfamiliar to all our users.

We compare TPFacet and Solr in terms of their usability in users' efficiency and quality of response to given tasks. For all the tasks we report the efficiency and quality of response using statistical analysis.

We performed our user study using eight graduate students from our university. As discussed in the following subsections, statistical analysis show that the conclusions we draw from these eight users is statistically significant. To make sure that users clearly understand the tasks and significance of different quality measures, we chose participants who had all taken the graduate level machine learning class. We chose users with some amount of technical expertise to ensure that they can properly understand the quantitative measures by which their response would be evaluated. Although normal users also do these tasks during information exploration, they do not necessarily think of how they can quantitatively compare one choice vs. another.

We gave all users a demo for 1.5 hrs explaining all features of the TPFacet and the steps to do the tasks using both the interfaces. During the demo it took us around 5 minutes to explain to users the details of the SFI View. Significant time during the demo went in explaining the three very different types of tasks and their quantitative evaluation measures. We also had to explain how to use both the interface for doing those tasks. We allowed users to do the tasks remotely to minimize effect of any environmental factors. We created 3 matched pairs of tasks, one pair for each type

described below. We divided the eight users into two equal groups. We indicate each user by their user id $U1$-$U8$. Users with id $U1$-$U4$ were assigned to group 1 and $U5$-$U8$ to group 2. For a task pair (A, B) we asked one of the groups to do task A using TPFacet and task B using Solr. We reversed the task assignment for the other group. In other words, if a task was done by group 1 users using Solr, then the same task was done by group 2 users using TPFacet, and vice versa.

For all the three tasks we have performed linear mixed model statistical analysis [109]. We use Display type as fixed effect and User ID as random effect. Computing p-values for mixed models aren't as straightforward as they are for linear models. The most popular way to obtain p-value is to use the *Likelihood Ratio* test as a means to attain p-values. The logic of the likelihood ratio test is to compare the likelihood of two models with each other. First, the model without the factor that one is interested in (the null model) and then the model with the factor that one is interested in. By comparing these two models, one can determine whether the factor one is considering is significant or not. We use ANOVA to compare the two models.

4.6.2.1 Simple Classifier

This task illustrates how TPFacet can be used to overcome the limitation of not being able to compare facet values within a facet. For example, a user may not know what is the best way to differentiate a SUV from a Wagon. Users ability to come up with the best differentiating condition indicates their very good understanding of the dataset.

We asked users to build a simple classifier. Classification is an important machine learning problem where given a training data with multiple class labels, one builds a classification model by which one can find the set of classes (categories) a new test observation belongs. In this task, we build a classifier for binary class data. We assume a simple classification model that consists of selecting at most two facet values

that maximizes the number of tuples retrieved from a given target class, and minimizes the number of tuples from the other class. Although problems like classification are rarely done manually for large datasets, human ability in this task demonstrates an understanding of crucial database themes. We evaluate the goodness of the classifier using standard F1 accuracy score. A sample task was to build a classifier for target class `Bruises` = true, where the given classes were `Bruises` = {true, false}.

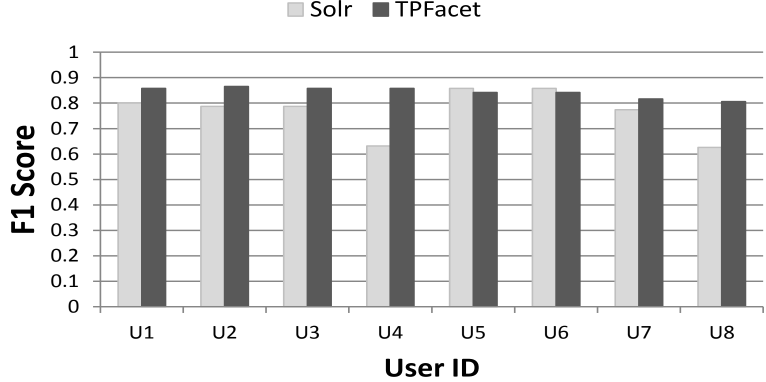


Figure 4.2: In this task users had to build a binary class classifier. Statistical analysis shows that TPFacet affects the quality of classfier by ($\chi^2(1) = 5.572,\ p = 0.018$), increasing the F1 score by about 0.078 ± 0.0285.

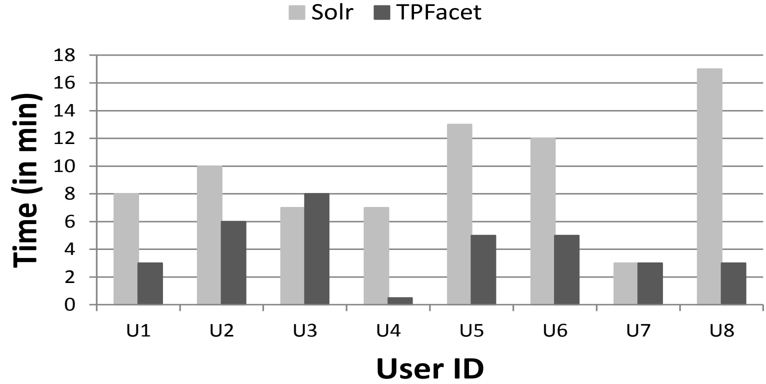


Figure 4.3: In this task users had to build a binary class classifier. Statistical analysis shows that TPFacet affects the time taken by ($\chi^2(1) = 8.54,\ p = 0.003$), lowering it by about 5.44 ± 1.56 minutes.

Figure 4.2 shows the F1 scores for the classifiers that users got for this task. Statistical analysis shows that TPFacet affects the quality of classfier by ($\chi^2(1) = 5.572,\ p = 0.018$), increasing the F1 score by about 0.078 ± 0.0285. Moreover, the

variation in F1 score is much lower when users use the TPFacet system as compared to Solr because the exploration using TPFacet is more methodical. In Figure 4.3, we show the time taken by the users to build the classifiers. Statistical analysis shows that TPFacet affects the time taken by ($\chi^2(1) = 8.54,\ p = 0.003$), lowering it by about 5.44 ± 1.56 minutes.

4.6.2.2 Most Similar Facet Value Pair

This task illustrates how TPFacet can be used to overcome the limitation of not being able to find similar (or equivalent) facet values, especially in categorical facets. For example, let's assume that a user is looking for cars from for a particular manufacturer and the database does not have good cars from that manufacturer. The user would like to look at cars from other manufacturers who are very similar to the particular manufacturer.

In this task, we gave users a list of four facet values from a facet and asked them to find the two most similar facet values. For example, given facet = `GillColor` and facet values = {buff, white, brown, green}, find the two most similar gill colors.

In a traditional faceted interface, users can compare facet values by comparing their summary digest. We gave users a cosine-similarity based distance metric to compare the summary digests. We asked users to select each of the given facet values, one at a time, and compare their summary digest. In SFI View, we didn't show the computed similarity scores, but allowed users to use interactive effects to find similar IUnits and facet values.

Figure 4.4 shows users response quality for this task. Since there are four facet values, there are 6 possible facet value pairs. We computed the defined similarity score for each pair and ranked them from 1 to 6, with the most similar pair being ranked as 1. Since computing exact similarity score is very hard for humans, we purposely chose facets and facet values that would make the task humanly feasible

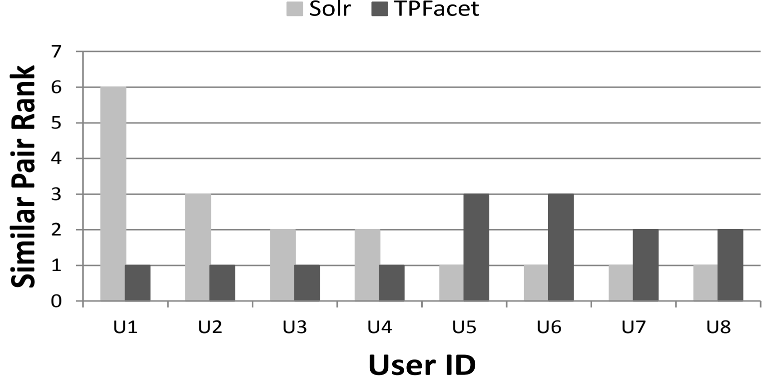


Figure 4.4: In this task users had to find the most similar facet value pair from a given list of four categorical facet values. Statistical analysis shows that there is no significant difference in users response quality by using the two types of interfaces.

in Solr. The similarity between gill colors brown and white was so high as compared to other choices that all the eight users got correct answer for this task. Group 1 users ($U1$-$U4$) did this task using TPFacet and group 2 users using Solr. However, the other similarity task was slightly harder. For the other task, users $U7$ and $U8$ got the most similar facet value pair according to facet value similarity we defined in Section 4.5, but according to the metric defined in this task, they turned out to be second most similar pair. Statistical analysis shows that there is no difference in users response quality by using the two types of interface.

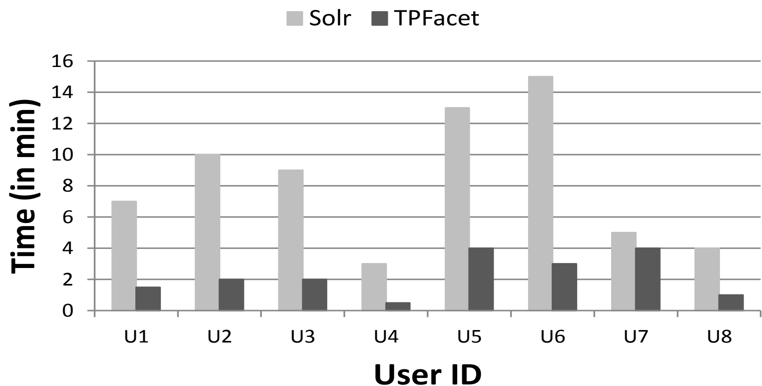


Figure 4.5: In this task users had to find the most similar facet value pair from a given list of four categorical facet values. Statistical analysis shows that TPFacet affects the time taken by ($\chi^2(1) = 12.04,\ p = 0.0005$), lowering it by about 6.00 ± 1.23 minutes.

Figure 4.5 shows the time users took to finish this task. Statistical analysis shows that TPFacet affects the time taken by ($\chi^2(1) = 12.04,\ p = 0.0005$), lowering it by about 6.00 ± 1.23 minutes. All users, except user $U7$, finished the task around four times faster using TPFacet as compared to Solr. Since the users were doing this task for the first time, some of them were trying to manually compare the IUnits. Users could have got the desired answer for this task much faster by just using the interactive effects, as seen in case of users $U4, U8$ and $U1$.

4.6.2.3 Alternative Search Condition

This task illustrates how TPFacet can be used to overcome the limitation of hidden facets. Users can query for hidden facets in terms of queriable facets. In this task, we gave users a set of selection conditions that lead to some result set $\mathcal{R}$. We asked users to find another set of selection conditions that would lead to same result set $\mathcal{R}$, but not using any of the already given selection conditions. One can see the given selection conditions as selection conditions on hidden facets that the users cannot query. Only an informed user can precisely access the desired result set using an alternate option. A sample task was to find an alternative selection condition using at most two facet values that would lead to the same result as selecting: `StalkShape` = enlarged and `SporePrintColor` = chocolate.

To evaluate users response quality, we checked the similarity between the query result obtained from the given selection condition and the users alternate selection condition. To measure similarity between the two results, we measured the similarity between their faceted summary digest.

Figure 4.6 shows users response quality for this task. Statistical analysis shows that TPFacet affects the users alternative search condition by ($\chi^2(1) = 3.28,\ p = 0.07$), lowering the retrieval error by about 0.329 ± 0.172. Using TPFacet most users were able to do the task with five times lower retrieval error. In this task pair, the

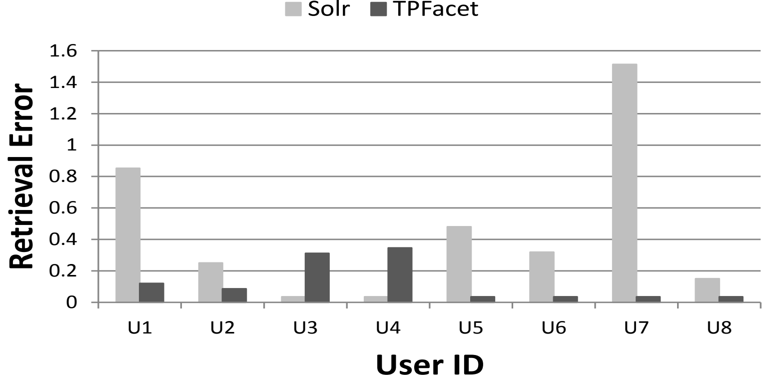


Figure 4.6: In this task users had to find an alternate search condition that would lead to the same result set as the given search condition. Statistical analysis shows that TPFacet affects the users alternative search condition by ($\chi^2(1) = 3.28,\ p = 0.07$), lowering the retrieval error by about 0.329 ± 0.172.

task that group 1 users did using Solr, turned out to be quite easier compared to the one they had to do using TPFacet. We can see this difference by seeing that users $U3$ - $U8$ have very low and similar error for this task. For the easier task, just one facet value was sufficient to get to the desired result set. All the users in group 2 had come up with slightly variant solutions, but exactly the same retrieval error (48 missing tuples out of 1344). Since TPFacet allows users to do this type of task in more methodical approach compared to Solr, we find much lower variation in users response quality. For the slighly harder task, we see slight variation in retrieval error among group 1 users who did this task using TPFacet, but the error variation is much higher for group 2 users who did it using Solr. Group 2 users, such as $U5, U6$ and $U8$, who had much lower error compared to user $U7$ had to spend significantly more amount of time as seen in Figure 4.7.

Figure 4.7 shows the time users took to finish this task. Statistical analysis shows that TPFacet affects the time taken by ($\chi^2(1) = 2.58,\ p = 0.108$), lowering it by about 2.00 ± 1.14 minutes. Most users were able to do the task 1.5 to 2 times faster using TPFacet as compared to Solr. This task required more time because users had to manually differentiate the IUnits. The main benefit of TPFacet was that users

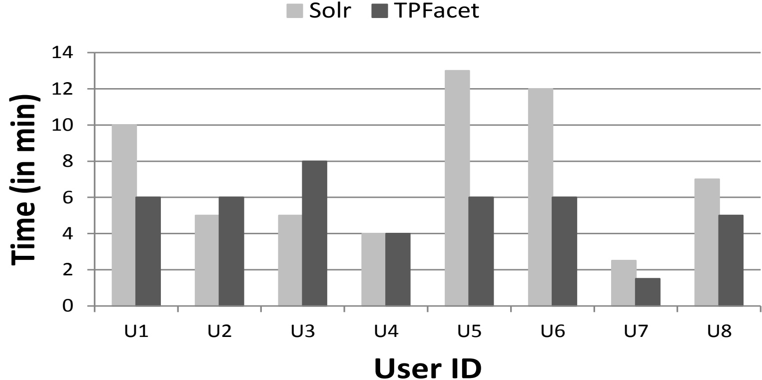


Figure 4.7: In this task users had to find an alternate search condition that would lead to the same result set as the given search condition. Statistical analysis shows that TPFacet affects the time taken by ($\chi^2(1) = 2.58,\ p = 0.108$), lowering it by about 2.00 ± 1.14 minutes.

didn't have to try various options using hit-and-trial. They had to look through the IUnits to find the discriminating facet values, but then it was just trying very few possible alternate choices to see which one gives the best result. For a slightly more complicated dataset and/or selection condition, this type of task can easily become infeasible using Solr.

4.6.3 Qualitative Analysis of User Study

In this section, we present an observational study that we performed using six users. We allowed all users to use both TPFacet and the traditional faceted interface for buying a used-car according to their own preference. Before the study we gave them a short demo of the type of insights one can gain through other datasets, such as buying a camera or a garbage disposal, using both the interfaces. We asked users to follow the think-aloud protocol, where we kept on prompting them to talk out loud while they were working. After the user study we asked them what they were thinking and why they did it. The purpose of the study is to show that normal end-users can gain insights of the kind that we claim TPFacet helps people notice when exploring a dataset. Following is a summary report of what we observed:

Queriable Facets

- ◉ **Body Style**
  - ☑ Sedan (16131)
  - ☐ SUV (10891)
  - ☐ Pickup (3611)
  - ☐ Coupe (1941)
  - ☐ Hatchback (1753)
  - ☐ Minivan (1644)
  - ☐ Wagon (1304)
  - ☐ Convertible (1273)
  - ☐ Van (292)
  - ☐ Chassis (48)
- ○ **Drivetrain**
  - ☐ 2WD (27954)
  - ☐ 4WD (6189)
  - ☐ AWD (4745)
- ○ **Transmission**
  - ☑ Automatic (36544)
  - ☐ Manual (2344)
- ○ **Year**
  - ☐ 2012-2013 (13850)

PFacet	IFacets	IUnit 1	IUnit 2	IUnit 3	IUnit 4
Price = 5-10K	Model Make Year Mileage Engine BodyType	['Impala LT'] ['Impala LS'] [Chevrolet] [2008] [2006] [2010] [110-100K] [80-90K, 90-100K] [6] [Sedan]	['Cobalt LS'] ['Cobalt LT'] ['Malibu LS'] [Chevrolet] [2008] [2006] [80-90K] [60-70K, 80-90K] [4] [Sedan]	['Impala '] ['Impala LS'] [Chevrolet] [2004] [2006] [2002] [90-100K, 80-90K] [6] [Sedan]	['Aveo LS'] ['Aveo LT'] [Chevrolet] [2010] [2012] [2004] [30-40K] [40-50K] [4] [Sedan]
Price = 10-15K	Model Make Year Mileage Engine BodyType	['Impala LT'] ['Impala LS'] [Chevrolet] [2012] [40-50K, 40-50K] [20-30K] [6] [Sedan]	['Malibu LT'] ['Aveo LT'] ['Aveo LS'] ['Malibu LS'] [Chevrolet] [2012] [40-50K, 40-50K] [4] [Sedan]	['Cobalt LT'] ['Malibu LT'] ['Malibu LS'] [Chevrolet] [2010] [50-60K] [40-50K] [4] [Sedan]	['Impala LT'] ['Impala LS'] ['Impala SS'] [Chevrolet] [2008] [2010] [2006] [60-70K] [70-80K] [6] [Sedan]
Price = 15-20K	Model Make Year Mileage Engine BodyType	['Impala LT'] ['Impala LTZ'] [Chevrolet] [2012] [10-20K] [30-40K, 20-30K] [6] [Sedan]	['Cruze LT'] ['Malibu LT'] ['Malibu LS'] [Chevrolet] [2012] [20-30K] [10-20K] [4] [Sedan]	['Malibu LT'] ['Malibu LTZ'] [Chevrolet] [2012] [2010] [30-40K] [20-30K] [4] [Sedan]	['Impala SS'] ['Malibu LTZ'] ['Impala LTZ'] [Chevrolet] [2010] [2008] [2006] [40-50K] [30-40K] [10-20K] [6] [8] [Sedan]
Price = 20-25K	Model Make Year Mileage Engine BodyType	['Malibu LT'] ['Malibu Eco'] ['Cruze LT'] [Chevrolet] [2014] [2012] [0-10K] [10-20K] [4] [Sedan]	['Impala LTZ'] ['Impala LT'] [Chevrolet] [2012] [10-20K] [20-30K] [6] [Sedan]	['Malibu LTZ'] [Chevrolet] [2012] [20-30K] [20-30K] [6] [Sedan]	['Cruze LTZ'] [Chevrolet] [2012] [10-20K] [4] [Sedan]

Figure 4.8: This screen capture shows a sample SFI View that a user had seen during the observational study. It shows the top-4 IUnits for each of the four price ranges the user had selected in the Pivot Facet `Price`. The user had also selected `Make` = Chevrolet, `BodyType` = Sedan and `Transmission` = Automatic.

Queriable Facets

- ◉ **Body Style**
  - ☐ Sedan (16131)
  - ☐ SUV (10891)
  - ☐ Pickup (3611)
  - ☐ Coupe (1941)
  - ☑ Hatchback (1753)
  - ☐ Minivan (1644)
  - ☐ Wagon (1304)
  - ☐ Convertible (1273)
  - ☐ Van (292)
  - ☐ Chassis (48)
- ○ **Drivetrain**
  - ☐ 2WD (27954)
  - ☐ 4WD (6189)
  - ☐ AWD (4745)
- ○ **Transmission**
  - ☑ Automatic (36544)
  - ☐ Manual (2344)
- ○ **Year**
  - ☐ 2012-2013 (13850)
  - ☐ 2010-2011 (9925)
  - ☐ 2008-2009 (6601)
  - ☐ 2006-2007 (3834)

PFacet	IFacets	IUnit 1	IUnit 2	IUnit 3	IUnit 4
Make = Chevrolet	Price Model Engine BodyType Year Drivetrain Doors	[10-15K] ['Aveo LT'] ['Aveo LS'] [V4] [Hatchback] [2012] [2WD] [4]	[5-10K] ['Aveo LS'] ['Aveo LT'] [V4] [Hatchback] [2010] [2WD] [4]	[15-20K] ['Sonic LTZ'] ['Sonic LT'] [V4] [Hatchback] [2012] [2WD] [4]	[5-10K] [10-15K] ['Aveo '] ['Aveo LS'] [V4] [Hatchback] [2008] [2WD] [4]
Make = Dodge	Price Model Engine BodyType Year Drivetrain Doors	[10-15K] ['Caliber SXT'] [V4] [Hatchback] [2010] [2008] [2WD] [4]	[10-15K] ['Caliber Mainstreet'] ['Caliber Heat'] [V4] [Hatchback] [2012] [2WD] [4]	[5-10K] ['Caliber SXT'] ['Caliber SE'] [V4] [Hatchback] [2008] [2WD] [4]	[10-15K] ['Caliber R/T'] [V4] [Hatchback] [2008] [AWD] [4]
Make = Ford	Price Model Engine BodyType Year Drivetrain Doors	[10-15K] [15-20K] ['Fiesta SES'] ['Fiesta SE'] ['Focus SE'] [V4] [Hatchback] [2012] [2WD] [4]	[15-20K] ['Focus SE'] ['Focus SEL'] [V4] [Hatchback] [2012] [2WD] [4]	[5-10K] [0-5K] ['Focus '] [V4] [Hatchback] [2006] [2004] [2002] [2008] [2WD] [2] [4]	
Make = Honda	Price Model Engine BodyType Year Drivetrain Doors	[15-20K] ['Fit Sport'] [V4] [Hatchback] [2012] [2WD] [4]	[10-15K] ['Fit Sport'] [V4] [Hatchback] [2010] [2008] [2WD] [4]	[10-15K] ['Fit '] [V4] [Hatchback] [2010] [2WD] [4]	

Figure 4.9: This screen capture shows a sample SFI View that a user had seen during the observational study. It shows the top-4 IUnits for each of the four manufacturers the user had selected in the Pivot Facet `Make`. The user had also selected `BodyType` = Hatchback and `Transmission` = Automatic.

User 1: This user started the search by selecting all the conditions that he would definitely want: Transmission = Automatic; Price $<$ 10K; Year $>$ 2006; and BodyType = Sedan, Hatchback. The user was left with 920 cars in the result panel after specifying all the choices that he definitely wanted. The user said, "I will like to go through all the 920 cars and then select my most preferred car." He went through some of the cars in the result panel and then aborted his search. The user said, "I am facing maximum difficulty in choosing values from facets Make and Model because they have too many values. Although I know which Make I would prefer given all other choices being same. But I cannot get this information easily from either the query panel or the result panel. Since I cannot infer the Make from which I get cars with reasonably good choices in all other attributes, I have to go through all the 920 cars."

The user then used the TPFacet interface. In the query panel he selected the conditions that he definitely wanted. Then he created an SFI View (similar to Figure 4.9) using Make as the Pivot Facet. The user said, "Using SFI View I can easily compare the different manufacturers based on my already made choices. I can look at the top-k sub-categories and see if any of those sub-categories are interesting to me." The user further said, "SFI View is useful because if I can select a preferred manufacturer, I can easily locate all other manufacturers that are similar to my preferred manufacturer using the 'find similar facet values' option."

User 2: This user made the following selections: Make = Chevrolet, Price = 5-10K, BodyType = Sedan, and Color = Red. She was left with 40 cars in the result panel. She said, "I will go through these cars and choose one." She said, "For buying a used-car, I do not expect to get much benefit from relational search interfaces. For buying a car, I would trust more on information I get from web documents, friends and family members. The query panel of faceted interface has very limited impact on my buying decision because they show me only a list of facet values from different facets. I already know what are my choice facet values from other external sources."

She said that for buying a used-car, she would prefer to use the interface more for lookup search, as compared to information exploration. The user, however, felt that SFI View could help her more than the traditional query panel because she could see the relationship of one facet value with other facet values using the SFI View. She also said, “I think that SFI View has bit too much information than what I would like to look at.”

User 3: This user had bought a car one year back which he had to buy within a very short time due to his constrained situation. He explained the struggles he had while using the search interface provided by e-commerce sites and how the SFI View addressed some of his problems. The user said, “When I started my search, I knew that I definitely want to buy a Sedan or Hatchback model from Toyota or Honda because of their less required maintenance over time. My choices were based on my friends' suggestions. I tried to perform an initial search for a car online, but was overwhelmed with all the available options and therefore decided to visit the local car dealers who gave me the prices for the cars in stock. I got an idea of a reasonable price to pay for the locally available cars by searching over different e-commerce sites. The e-commerce sites, however, could not help me much in understanding the car dataset. I was able to use these e-commerce sites simply to verify the typical car cost for specific configuration(s) which took some time as I had to search through all the tuples individually.”

The user did the following to evaluate the two interfaces. He selected 10 Toyota Sedans from an e-commerce site and assumed them to be the cars that he had seen at his local car dealer. In order to verify if the price of those cars were reasonable, he used the traditional faceted interface to select cars with `Make` = Toyota, `BodyType` = Sedan and `Mileage` < 50K. He had 480 cars in the result panel. He said, “I will go through all the 480 cars and compare cars from my local dealer with those posted on the e-commerce site.” He said, “Comparing the 10 cars with 480 cars in the result

panel is quite difficult because the tuples in the result panel are not organized in a suitable manner for performing comparison." He then created an SFI View similar to the one shown in Figure 4.8 (except for Toyota instead of Chevrolet and Sedan instead of Hatchback). The user was very happy to see that SFI View could show him the typical price range summary according to different types of categorizations which he could not find in the traditional faceted interface. Although he is a good statistician and highly educated, he said that the faceted search interface was too complex for him make good buying decisions based on that. He felt that SFI View would help him to easily rule out the costlier cars and get an easy understanding of the cars available on the online websites.

User 4: This user is a car dealer. He is very well-informed about the car domain. The user said, "For most of the facets I am quite informed of how facet values are related with other facet values within and across facets. However, I often have to help many sellers decide what should be the ideal price for selling a used-car. Since price of a car has very complex relationship with other attributes I would be interested to see whether TPFacet or the traditional faceted interface is more suitable for determining the optimal price of a car with given specific configurations." His problem was somewhat similar to User 3. However, unlike a buyer, his aim was to find the price which is slightly higher than the normal expected price. He wanted to determine the selling price of a car with the following configuration: `Make` = Chevrolet, `Model` = Impala, `BodyType` = Sedan and `Year` = 2012. After selecting the above conditions, he was left with 83 cars. He sorted the cars in the result panel twice: low-to-high and high-to-low. He saw that the prices of the cars vary between 11K to 19K. He observed that cars were not priced in a uniform manner. He said, "I can see that many cars with relatively bad mileage are priced much higher compared to cars with better mileage. Very similar cars sometimes have a price difference greater than 3K." He said that since sellers are not very clear of what should be the right price, very

similar cars are being sold at different price ranges. To determine the expected price, he created an SFI View as shown in Figure 4.8. In order to see the expected price of different types of Chevrolet Sedans, he didn't explictly select Model and Year values. In the SFI View shown in Figure 4.8, he saw that 2012 Impala LT cars are typically sold in the range 10-15K and 15-20K. He could get this information just by looking at 2 IUnits instead of having to look at 83 cars. He further said, "Using this one SFI View I can determine the good selling price for Chevrolet Sedans with different configurations." The user expressed that obtaining the right selling price for a particular car configuration is quite difficult and non-reliable from both the query panel and the result panel of the traditional faceted interface.

User 5: This user looked at cars with following configurations: Make = Chevrolet, BodyType = Hatchback, Price = 5-15K. He had 87 cars in the result panel. During the user study, this user was quite resistant to go through many pages of the result panel. In order to avoid the pain of having to browse through many pages of the result panel, he just wanted to select very specific conditions from the query panel and quickly get to a small result set. He was not very eager to explain why he was making such constrained choices. He said, "Humans cannot behave rationally like computers in making decisions. It is not possible for humans to retain and compare large number of choices like computers." The user skimmed through five pages of tuples in the result panel. Then he used the TPFacet to construct the SFI View shown in Figure 4.9. The user said, "SFI is very useful because I can obtain the information that I saw by browsing through five pages in the result panel by just seeing a one page SFI View data summary. I do not have to struggle with remembering lots of information spread across many pages. Using SFI View I can make my choices more rationally because it gives a very nice summary of the whole result set." The user said that in relational tuples even sorting results by certain attributes does not help because the tuples that are at top rank may not always be the best choices. He liked the set level summary

shown in the form of SFI View.

User 6: This user was interested in buying a low-priced car with good fuel efficiency. He knew that Sedans and Hatchbacks are two suitable body types for his requirements. He started his search by selecting: `BodyType` = Sedan, Hatchback; `Price` < 10K; and `Make` = `Chevrolet, Toyota, Honda, Ford`. There were thousands of cars that satisfied his query. He went through few pages of the query result. He was willing to spend a significant amount of time in trying to make each selection choice in a very rational manner. Although he was clear of his choices, he was eager to get the help from the interface to find similar other choices. For example, while using the normal faceted interface he didn't realize that the body type Wagon is also quite appropriate for his requirements. He missed Wagons because the number of Wagons sold is much less compared to other popular body styles such as Sedans and SUVs. The user could see the similarity of Wagons with Sedans and Hatchbacks by seeing the similarity in their IUnits. By being able to see the interesting price and other features of Wagons in the SFI View, he was happy to include it in his high preference options. The user expressed at the end, "This SFI View is very useful because I can always use it to further enhance my preference function based on the actual available choices in the database. I can easily see what is available in the database and accordingly make my choices."

4.6.4 Performance

Computational time is a crucial constraint for all user interface applications because users expect almost instantaneous response. In this subsection, we evaluate whether TPFacet can provide interactive responses. We performed all our performance experiments on the YahooUsedCar dataset with 40K tuples and 11 facets. When users do faceted browsing over e-commerce sites, they rarely deal with result size that is more than 30K-40K tuples and 5-10 queriable facets. Thus we evaluate

our system using all the tuples of our used-car dataset as query result, with all its facets being used as queriable facets.

Our experiments show that TPFacet can give acceptable performance by just using computationally efficient feature selection and clustering algorithms. Each of our experimental graphs are based on average readings of 50 simulations, where for each simulation we generate a different query result by randomly selecting a subset of tuples and/or facets. The default parameters in these experiments are: the number of informative facets $\mathcal{I} = 11$, the number of generated IUnits $l = 10$, the number of IUnits shown $k = 6$, and the number of facet values selected in the Pivot Facet $\mathcal{V} = 5$. In these experiments, we assume that if the total size of the query result set is $|\mathcal{R}|$, then each facet value $v \in \mathcal{V}$ has $|\mathcal{R}|/|\mathcal{V}|$ tuples.

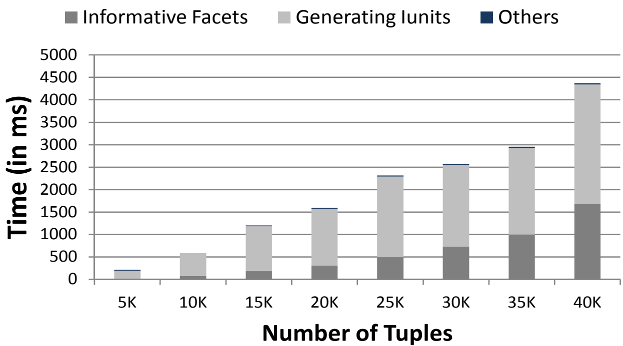


Figure 4.10: Time breakdown to compute SFI View for different result set sizes. Here the system parameters are chosen to demonstrate the worst-case performance of our system.

Figure 4.10 shows the total time to compute the SFI View for different sizes of query result. In the shown graph, we do not do any optimizations, except using a computationally efficient feature selection and clustering algorithm, that can lead to better system performance. Moreover, we chose system parameter values to demonstrate the worst-case performance of our system. For example, we kept $|\mathcal{I}| = 11$ and $l = 15$. When we consider interaction between many facets (large $|\mathcal{I}|$) or try to compute many interactions (large l), then it decreases system performance, as shown in

later experiments. We divide the total time into three parts: time to compute informative facets, time to generate IUnits and time for all remaining steps, such as top-k ranking, and similarity between IUnits and facet values, that we represent collectively as others. We can see that the most computationally intensive parts of TPFacet is computing the top informative facets and generating candidate IUnits. Total time for all other steps is negligible because of the small values of k and $|\mathcal{V}|$ established due to user's display constraint. We can see that even this naive solution is acceptable when the result size is less 15K. But as we increase the result set size, we can see that the time to compute SFI View increases and becomes almost 4.5 secs for 40K tuples. Since the result set size is likely to be the largest in the initial stages of exploration, and since this is also likely to be when the user really needs interactive response to freely try alternatives, a multi-second response time is too slow. To alleviate this problem, we developed several optimizations.

Optimization 1. Sampling — Sampling can improve both feature selection [83] and clustering [93]. For all our facets, when we computed the set of top ranked informative facets using a small random sample of size 5K-10K, we always got almost the same set, as we got from any larger sample size, including the full dataset. As shown in Figure 4.10, computing informative facets takes only 20-50 ms for 5K-10K tuples, as compared to 1700 ms for 40K tuples. Quality of informative facets is more crucial when users are towards the end of their exploration, and at that time even the exact computation will take very short time due to small result size. Even if there were some degradation in quality due to sampling, it may not matter much in the intial stages. Similarly, we can also reduce the time for generating IUnits by generating IUnits from a small sample.

Optimization 2. Varying Generated IUnits — Figure 4.11 shows the effect of number of generated IUnits l on computation time for different result sizes. We observed that as we increased the number of generated IUnits, it increases computation

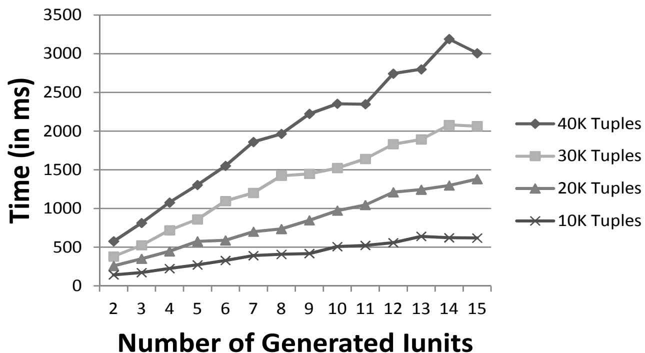


Figure 4.11: System performance with varying number of generated candidate IUnits.

time due to increased time for clustering. For small 10K result size, computation time is small, less than 500 ms, even when we generate 15 IUnits per facet value. However, if the result set is large and we generate large number of IUnits, as shown in Figure 4.10 for 40K tuples with $l = 15$, then it slows down system performance. When users are in their beginning stages of exploration, it is hard to know their preference because their query is too broad. Generating more IUnits and finally ranking is meaningful when we know users' preference more precisely, which typically happens near the end-stages of exploration. Thus we generate fewer IUnits when the result set is very large, so that we can provide a good summary of all options. As users narrow down their exploration, we increase the number of generated IUnits and return better top-k IUnits.

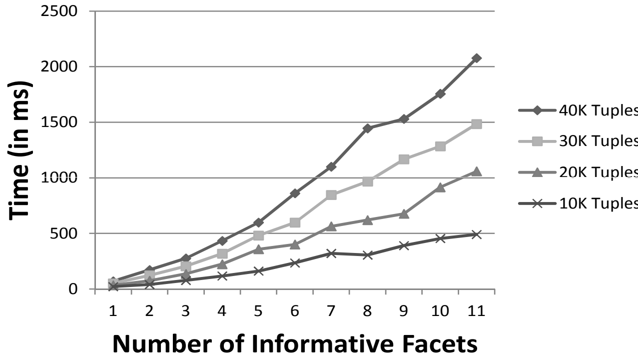


Figure 4.12: System performance with varying number of informative facets.

Optimization 3. Fewer Informative Facets — Figure 4.12 shows the effect of number of informative facets on computing clusters for different result sizes. As we increase the number of informative facets, it increases computation time because we need to look at the interaction between larger number of facets. However, as explained in Section 4.4.1.1, to create good IUnits we need to determine the actual number of IFacets based on the available screen space and the relevance score of each IFacet. By showing few informative facets we can cluster even 40K tuples in less than 500 ms.

By combining all the above optimizations in creating the SFI View, we can greatly increase the performance of TPFacet system. For example, we can get an SFI View for 40K tuples in less than 500 ms.

4.7 Conclusion

In this chapter, we introduced a novel two-phased faceted navigation model that can be used for ambitious information discovery in complex datasets. Our proposed navigation model relies on SFI View, which is an extension of the existing query panel. The SFI View helps users to systematically explore the relationship between different facet values which are currently shown as disconnected pieces of information. We also showed through an extensive user study that our SFI View can greatly facilitate user exploration and understanding of complex datasets. Although computing the SFI View is computationally intensive, we provided optimizations that enable it to be easily integrable with existing faceted interfaces, without compromising system performance.

CHAPTER V

Skimming through Relational Query Result

5.1 Problem Overview

The two add-on extensions we presented in the last two chapters were designed to facilitate query formulation during faceted navigation by providing better feedback to users through the query panel. However, there are phases during navigation where users like to look at the query result to understand how they can alter, narrow down or expand their search condition. Not surprisingly, many user interface studies [41, 16, 18, 17] have shown that users tend to browse quite often while searching for information, in addition to querying. These studies show that browsing is a rich and fundamental part of human information seeking behavior, and that querying is not the only mode of interaction with data. Users often formulate a more precise query after they have attained an understanding of the underlying data through quick exploratory browsing.

As shown in Figure 1.1, a faceted search often yields a large query result having hundreds, and even thousands, of tuples. Typically, these tuples are presented to a user through a pagewise scrolling interface. Since these tuples, typically, contain alphanumeric information and are devoid of visual cues, the user is easily overwhelmed. It becomes difficult for the user to perform meaningful tasks by scrolling through such large result set. In this chapter, our goal is to make the commonly used scrolling in-

terface more usable for browsing large relational result sets.

To help the user quickly get a sense of large result sets, various sophisticated data reduction techniques, such as clustering, information visualization, etc., are available (see Chapter VI for details). While there are situations in which such techniques can be effective, they also have significant limitations. First, clusters do not help the user unless they have labels (or other means) that clearly identify what can be found inside each cluster. Thus, a full scan is still required. For example, if the user were looking to buy a car, she would often insist on browsing the result set rather than only relying on the seller's clustering software. Secondly, it is important to let the user control the rate of consumption of information. For tabular data, the user can slow down or speed up her reading speed by moving the scrollbar accordingly. In a clustering-based system, one would need to allow the user to dynamically vary the granularity of each cluster in an interactive fashion. Such a solution is not only computationally challenging but is also unintuitive for quick browsing.

Where user preferences are well understood, results may be scored and ranked. Information retrieval systems routinely present large result sets in this fashion. Users can then focus on the few top-ranked results and skim, or even ignore, the rest. However, in the database context there is usually insufficient information to rank with confidence. A common solution in the case of databases is to sort the result set on some attribute of importance: say `Price`, in our `cars.com` or `amazon.com` example. The user then has to scroll through this large result, reading through each tuple. While the data is sorted on the `Price` column, the user still has to focus on reading the other columns, which may not correlate with `Price`. Clearly, reading through each non-sort-key attribute of each tuple in a entire large result set is a slow and arduous task. Therefore, our task becomes that of supporting fast browsing (and more specifically, scrolling) over such a sorted relational result set.

Humans browse all the time, for example with newspapers and magazines. Visual

cues in the layout assist users in browsing data quickly. Relational data tuples tend to comprise dense alphanumeric data, with few visual markers. Thus, there is a rather low maximum rate at which a human can skim tuples from a database. The only way to browse a database faster than this rate is to have the human eye see less than all the information.

To improve the usability of scrolling interfaces, we present in this chapter a variable-speed scrolling interface that can automatically adjust the amount of information displayed based on the user's scrolling rate. Our interface displays only a few selected tuples from each page, where the number of tuples is determined by the user's current scrolling speed. If some results seem interesting to the user while scrolling fast, the user can reduce the speed of scrolling, making the system show more tuples from the currently-viewed section of the data. In contrast to a typical sampling problem, our goal is to identify tuples that provide the most information with respect to the *entire scrolling session*, and not just the page at hand.

While the specific sorting attribute is not material for our algorithms, the fact that data is sorted/clustered in some way is crucial. For fast browsing to work, we must have "similar" records close together, for some user notion of similarity. If we did not have this, then selecting any set of representative tuples would permit only global conclusions, but not local ones: there would be no reason to expect other interesting entries in the vicinity of an interesting observed sample. There are many ways interesting records could have been placed together—for example, a clustering algorithm could have been used. The algorithms in this chapter will work in such a scenario as well. We chose to focus on sorting because this is so commonly used in database user interfaces, including faceted interface.

Contributions and Organization — Putting these ideas together, we extend the traditional scrolling interface by presenting a new variable-speed scrolling interface. Instead of showing users a fast-changing blur during rapid scrolling, our variabl-

speed scrolling interface shows few representative tuples to users so that they can get a "good impression" of the query result even during very fast scrolling. We show selected representative samples only when users scroll beyond a certain threshold speed, otherwise we retain the same characteristics as of traditional scrolling interface.

The main contributions of this chapter are as follows:

- We present a novel scrolling-aware browsing interface that shows representative tuples in the context of scrolling through groups of similar relational data.
- We provide precise information loss metrics that can quantify the loss of information that happens while browsing through only representative tuples.
- We develop and compare five new scrolling-based sampling algorithms that minimize information loss.
- Due to the interactive nature of our use-case, the efficiency of our algorithms is a key consideration. We provide efficiently computable algorithms that satisfy fast scrolling requirement.
- We perform a detailed user study comprising of three diverse tasks. Our study shows that users can perform all the tasks around twice faster and with higher accuracy using our scrolling-aware browsing interface as compared to traditional scrolling interface.

Chapter Layout: The rest of the chapter is organized as follows. In Section 5.2, we formally define our problem. We then describe the scrolling interface and define metrics for information loss. In Section 5.3, we introduce two naïve algorithms and propose five new algorithms to solve the problem at hand. We present experimental results in Section 5.4 and a user study in Section 5.4.3, followed by conclusions in Section 5.5.

5.2 Paging Interface with Scrolling

In this section, we describe our formal problem set up, user interface, and metrics used to measure information quality.

5.2.1 Problem Definition

A user gives an `ORDER BY` SQL query $\mathcal{Q}$, which is executed on database $\mathcal{D}$ and it generates a result set $\mathcal{R}$. The query can be over a single table or join over multiple tables. $\mathcal{R}$ contains N tuples and it requires S pages for display, i.e., pages $\{P_1, P_2, \cdots, P_S\}$. All pages, except the last page P_S, contain M tuples, where M is determined by the page and tuple size. For the remainder of the chapter we use the page size to indicate the number of tuples per page (i.e., M).

Our preliminary user study experiments (not reported here) showed that for alphanumeric, relational data, it is easier for users to see information using intermittent pagewise scrolling, as compared to continuous scrolling. In intermittent pagewise scrolling, the display contents are refreshed only when the scrollbar moves to adjacent pages. We believe this is because it is harder for a user to read continuously moving tuples, as found in a continuous scrolling interfaces. Based on this design, in our proposed scrolling interface, a user goes through a much smaller result set $\{D_1, D_2, \cdots, D_S\}$, where D_i is the set of representative tuples shown from page, P_i.

Our system computes the set D_i, for page P_i, based on user's current browsing speed, i.e., in our implemented system it is the average browsing speed at page P_{i-2}. Due to computational constraints, the tuples are selected when the user is currently browsing through page P_{i-1}. We compute K_i, the size of D_i, based on a straightforward inverse function of scrolling speed, discussed in Section 5.4.3.4. Our central problem is to select D_i that gives the user a good impression (defined later in Section 5.2.3) of page P_i, and at the same time, avoids showing redundant information. Furthermore, this selection has to be performed very fast, so that the user is not

hindered by our system even when browsing the data set at high speed.

5.2.2 User Interface

It should be noted that designing a user interface is not the primary aim of our work—rather, our goal is to design algorithms that can identify representative tuples that provide high quality information to the user within the real-time constraints of fast browsing.

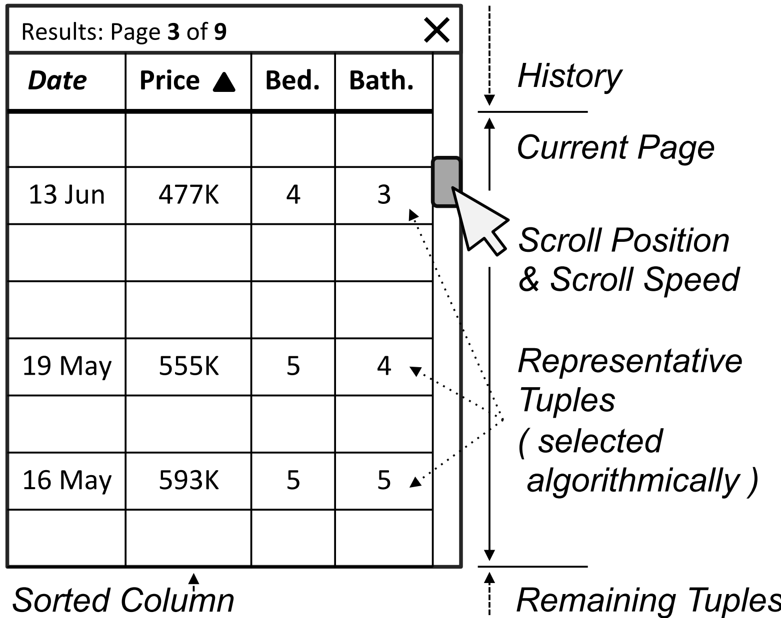


Figure 5.1: *User interface:* Results are displayed in a paginated interface, browsable using the scrollbar. The scroll position determines the currently displayed page. Instead of overwhelming the user with a full display where all tuples are shown, the user is presented with a condensed display featuring a set of representative tuples. These tuples are selected from the current page based on contents of the page itself, the history of pages seen so far, the scroll position and the speed of scrolling.

A user can scroll through the result pages, $\{P_1, P_2, \cdots, P_S\}$, arbitrarily varying her scrolling speed. At a *slow speed*, all the tuples are shown; whereas in the *high speed* case, only a few selected tuples are shown. In Figure 5.1, we show a page of tuples containing properties from a realtor database. Through three tuples, our system is able to give a good overall impression of the full page. If the scroll speed changes

and a page is redisplayed with more tuples, the relative location of the originally displayed tuples remains fixed, giving the user a point of reference and allowing her to read information that is stationary. In addition to the tuples, we show the page number of currently displayed page.

In our interface, when a user scrolls backwards (scrolling towards beginning of the result set), we display all the tuples (full display), but when a user scrolls in forward direction (towards the end of result set), we display only selected representative tuples. During the forward scroll, if the user's average scroll speed is below a threshold speed V_t, then all the tuples will be shown. Thus, a user can see all the tuples in page P_i, by either scrolling in forward direction below the threshold speed at page P_{i-2}, or scrolling in backward direction at page P_i. The reason to maintain this asymmetry is because users are expected to scroll forward most of the time, but we want to give them a way to stop suddenly when something catches their eye.

5.2.3 Goodness Measure

We address the issue of information overload encountered during fast browsing by displaying only a select set of representative tuples from each page, instead of showing all tuples that satisfy the user's query. In other words, we show the user less information, which brings up the question of information loss. Can we quantify the information we did not show the user?

Entropy-based information loss measures are well studied in the area of Information Theory. However, we were not able to use these measures for our specific problem because defining row wise entropy, or significance of an individual tuple with respect to the whole result set, is hard, since all values in a row are distinct. To the best of our knowledge, there is no existing work that mathematically quantifies this kind of information loss. In this section, we develop some natural metrics for this purpose. To do so, we first define two preliminary concepts: *scroll log* and *history.*

Scroll Log: A scroll log (*SL*) maintains the sequence in which pages of a query result were visited by a user. It logs three things: sequence number *sid*; page number *pid*; and a list of tuple ids *tupleList*, of all displayed tuples from page *pid*. Thus, the log has the form (*sid, pid, tupleList*), where sequence id *sid* is incremented with every new page visit during a forward scroll.

History: We call already displayed information as *history* and denote it by *H(sid)*. *H(sid)* contains the ids of all tuples that have been shown to the user, prior to scrolling in the *sid* page in sequence. The list of tuple ids shown from this page is denoted as *tupleList(sid)*. Thus $H(sid) = \bigcup_{i=1}^{sid-1} tupleList(i)$. Note that with a succession of forward and backward scrolls, the history of a page P_i could contain information from pages after page P_i.

We now go on to define our metrics for information loss in the context of browsing data: *Tuplewise Information Loss*, *Pagewise Information Loss* and *Cumulative Information Loss*.

Definition V.1. The **Tuplewise Information Loss (TIL)** score of a non-displayed tuple t_{nd}, from page P_i having sequence id '*sid*' in scroll log, is the dissimilarity of t_{nd} with respect to the most similar tuple t_d, from history $H(sid)$ and tuples shown from page P_i. Thus,

$$TIL(t_{nd},\ sid) = \mathcal{V}(t_{nd},\ t_d) \tag{5.1}$$

Here, $\mathcal{V}$ gives the dissimilarity between t_{nd} and t_d, where $t_d \in H(sid) \cup tupleList(sid)$.

We can use any distance function $\mathcal{V}$ to measure dissimilarity. In the experiments reported in this chapter, we use simple Euclidian distance after scaling dimensions to comparable ranges. Intuitively, $TIL(t_{nd},\ sid)$ gives a measure of the most similar information with respect to t_{nd} that has been shown to the user from either already displayed tuples $H(sid)$ or tuples that has been shown from page P_i, *tupleList(sid)*.

Definition V.2. The **Pagewise Information Loss (PIL)** score for a page P_i, is

defined as the sum of TIL scores of all tuples from P_i. For pages which are not scrolled or visited, PIL score is zero. Thus,

$$PIL(P_i,\ sid) = \sum_{t_p \in P_i} TIL(t_p,\ sid) \tag{5.2}$$

Since the *TIL* score of all displayed tuples is zero, the *PIL* score of a page is the sum of the *TIL* score of all non-displayed tuples.

To minimize information loss even during fast scrolling, we need to ensure that we do not show redundant information, i.e., information that is similar to what has already been shown. To avoid redundancy, while selecting tuples from any page P_i, we need to take into consideration the information that has been shown prior to display of P_i, which we maintain in the form of history.

The relationship between history and *PIL* score can be understood more clearly through Figure 5.2, where we have shown the selected tuples (current representatives) from two types of sampling algorithms—one that considers no history (local sampling) vs one that consider displayed history (history based sampling). All the circles represent tuples in a page of a binary relation, and we have to select two representative tuples from this page. Clearly by selecting the two representative tuples as shown in Figure 5.2 (a), we reduce the overall *PIL* score for the page. The problem with local sampling can be understood by seeing the already displayed tuples (i.e., historical representative) in Figure 5.2 (b). The two selected tuples in local sampling do not provide as much additional information as compared to what the user had already seen in the past. As shown in 5.2 (b), considering history enables us to provide new information that is substantially different from previously shown information.

Definition V.3. The **Cumulative Information Loss (CIL)** score for a scroll log

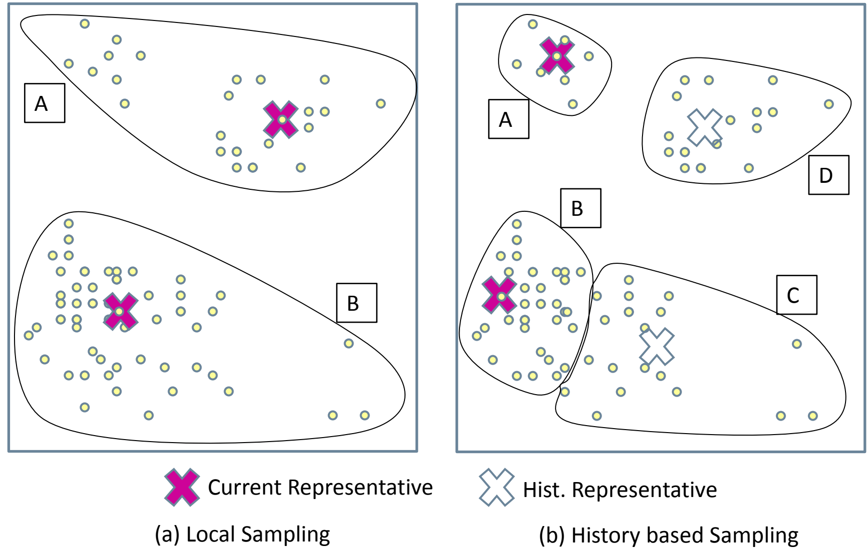


Figure 5.2: Comparison between computing representative tuples without considering user's browsing history vs. considering browsing history.

'SL' and result set $\mathcal{R}$, is defined as the sum of PIL score of all entries in 'SL'. Thus,

$$CIL(SL, \mathcal{R}) = \sum_{sid=1}^{|SL|} PIL(pid,\ sid) \tag{5.3}$$

Thus, *CIL* score is the sum of *PIL* scores of all entries in scroll log *SL*. Intuitively, given two sampling algorithms A and B, one can say that algorithm A is better than B in terms of minimizing information loss, if $CIL_A(SL, \mathcal{R}) < CIL_B(SL, \mathcal{R})$, where $CIL_X(SL, \mathcal{R})$ denotes the cumulative information loss incurred by sampling according to algorithm X, given the result set $\mathcal{R}$ and scroll log SL.

5.3 Algorithms

In this section, we present two naïve and five novel sampling algorithms that minimizes *CIL* score. Since *CIL* score is sum of pagewise *PIL* scores, these algorithms minimize individual pagewise *PIL* score. For the sake of simplicity, we assume that we have a data page P, with associated history H, and that we have to select K representatives from page P. The collection of selected representatives is called the

display set D. The sizes of the page P and history H are denoted by M and $|H|$, respectively. Literature in the clustering domain generally use the terminology *object*, which is equivalent to our *tuples*. We use these two terms interchangeably when describing our sampling algorithms.

5.3.1 Naïve Sampling

To the best our knowledge, there is no existing work that supports variable-speed scrolling for relational data, and thus we use two sampling techniques, namely random and uniform sampling, as our naïve solutions to evaluate and compare against. In these naïve solutions, we pick K random or uniformly spaced tuples from page P. We refer to these naïve sampling techniques as RS and US, respectively. Since the data is sorted, we expected uniform sampling to give a good overall impression because it will select tuples at regular interval.

5.3.2 K-medoids Based Sampling

5.3.2.1 Relationship between PIL Score and K-medoids

Minimizing the *PIL* score (i.e., Definition V.2) is equivalent to solving the standard K-medoids clustering algorithm, with some modifications. K-medoids is a partitioning based clustering algorithm that divides a given set of objects into K partitions and returns an actual representative object from each partition or cluster.

In K-medoids, we try to minimize the following absolute error criterion:

$$E_{K\text{-medoids}}(P) = \sum_{j=1}^{K} \sum_{p \in C_j} \mathcal{V}(p, o_j) \tag{5.4}$$

Here, $E_{K\text{-medoids}}$ is the sum of absolute error for all objects in the dataset P; p is an object in cluster C_j, which is represented by object o_j. $\mathcal{V}(p, o_j)$ measures the error in representing object p by object o_j. $\mathcal{V}$ could be any generic dissimilarity function,

including functions over objects having categorical features.

Minimizing the PIL score of page P is equivalent to minimizing the following equation:

$$PIL(P) = \sum_{j=1}^{L} \sum_{p \in C_j} \mathcal{V}(p, o_j) \tag{5.5}$$

where, $L = |H| + K$. While minimizing Equation 5.5, we can only select K new representatives from page P, the $|H|$ representative objects shown in previous pages cannot be changed.

The mathematical formulation for K-medoids and PIL score looks quite similar, with the difference that in the PIL score, we have fixed cluster centers from previous pages and that we want to avoid picking new centers which are close to the already displayed cluster centers, as discussed earlier in Section 5.2.3.

5.3.2.2 Local K-medoids

In the Local K-medoids (LKMed) sampling algorithm, we do not consider the effect of history while minimizing Equation 5.5. We compute the representatives using the normal K-medoids formulation, i.e., Equation 5.4. Any standard K-medoids algorithm can be used for selecting the K representatives from page P, we use PAM (Partition Around Medoids) [69]. For large datasets, more efficient algorithms such as CLARA [69] and CLARANS [93] are generally used, which use sampling and randomization, respectively, to reduce the computational cost of PAM. To get reasonable results, these algorithms still need at least several dozen points after the sampling. Since we are clustering one page of tuples at a time, we only have a few dozen tuples to cluster. For these reasons, the extra machinery to handle large data sets is not required and PAM is our preferred algorithm.

PAM is an iterative clustering algorithm that requires $O(K.(M-K)^2)$ distance computations per iteration, where M is dataset size. For typical values of M (say

50) and K (say 10) in our application, this is still several thousand computations that must be performed within time so short (milliseconds) that the interface feels responsive to the user. Since M is small enough that we can afford M^2 storage in memory, we improved the performance of PAM by computing the distance between all tuples of a page once, and then using it in all iterations. This avoids the need to repeatedly compute distances between tuples, as required in classic PAM [69].

5.3.2.3 Historical K-medoids

Historical K-medoid (HKMed) modifies LKMed by keeping history H and modifying the PAM algorithm in light of H. Samples in HKMed are computed according to Algorithm 3. While HKMed seems to have additional computation steps as compared to LKMed because it takes history into account, both our theoretical and experimental evaluation show that HKMed is better than LKMed, both in terms of information quality and computation time (a more detailed explanation is provided in the forthcoming paragraphs).

Algorithm 3 differs from the basic PAM algorithm in the addition of steps 2 and 7. In basic PAM, we perform step 4 to select the initial representatives and then keep repeating steps 6, 8-14 till convergence. In the above algorithm, '\' refers to the set difference operator. In step 6, we compute the nearest neighbor of all non-displayed objects (tuples) from page P with respect to the currently selected representatives D. In steps 8-14, the goal is to swap one of the currently selected representative objects o_m with a non-selected object o_p that gives the greatest reduction in error cost. The best pair is one for which the TC_{mp} value is minimum and this value should be negative. We would refer the readers to [69, 93] for more details of the basic PAM algorithm. For both LKMed and HKMed algorithm, each iteration involves computing *SwapCost* $K.(M-K)$ times. In the *SwapCost* algorithm, we need to consider the effect of swap on all the $(M-K)$ non-selected objects, and thus the overall complexity is

Algorithm 3 Historical K-medoids

Input: K: size of display set
P: a data page
H: displayed history
Output: D: display set from page P
Method:

1: $flag \leftarrow true$
2: map $NH \leftarrow NearestNeighbor(P,\ H)$
3: map $TD \leftarrow AllPairDistance(P)$
4: $D \leftarrow K$ random objects from P as initial representatives
5: **repeat**
6: map $NM \leftarrow NearestNeighbor(P \setminus D,\ D)$
7: map $NC \leftarrow MergeNearestNeighbor(NH,\ NM)$
8: for all pairs of objects o_m, o_p, where $o_m \in D$ and $o_p \in P \setminus D$, compute $TC_{mp} \leftarrow SwapCost(o_m,\ o_p)$
9: **if** $(Min(TC_{mp}) < 0)$ **then**
10: $D \leftarrow (D \setminus o_m) \cup \{o_p\}$
11: **else**
12: $flag \leftarrow false$
13: **end if**
14: **until** (flag = true)
15: return D

O($K.(M - K)^2$).

In step 2, we compute the nearest neighbor for all tuples in page P with respect to history H. We keep both tuple id and distance in the map NH. In step 4, to avoid repeated distance computation between tuples in P, we pre-compute and store the distance of all tuples from each other in the map TD. Step 4 is useful even in basic PAM, as it can reduce the cost for computing the nearest neighbor in step 6, and second nearest neighbor in step 8, i.e., *SwapCost* algorithm. In step 7, we merge the nearest neighbor from historically displayed tuples H and the currently selected representative tuples D. The *SwapCost* algorithm has the same four cases as that of basic PAM algorithm [69, 93], with the exception that only the representatives from D (and not H) can be swapped with non-selected representatives from $P \setminus D$. The swap costs can be efficiently computed by using the information in map NC of step 7 and map TD of step 4.

As we will see experimentally in Section 5.4, HKMed is better than LKMed both in terms of information quality and computation time. The better information quality is because HKMed exactly minimizes the *PIL* score (LKMed returns samples like Figure 5.2 (a), whereas HKMed returns samples like Figure 5.2 (b)). While HKMed seems to have two additional steps, i.e., step 2 and 7, it is computationally faster than LKMed. HKMed is faster due to reduced computation in *SwapCost* algorithm of step 8. In the *SwapCost* algorithm, there are four cases to consider for each non-selected object o_j, when we replace an existing medoid o_m by a new medoid o_p (see details in [69, 93]). For a non-selected object (o_j) that is in o_m's cluster, we have to consider cases 1 and 2, where we need to compute the second nearest neighbor from D to compute the replacement cost. In case o_j is not in o_m's cluster, we need to consider cases 3 and 4, which are faster to compute. In case of HKMed, we have a number of fixed medoids from history H and thus the number of times we need to consider cases 1 and 2 for the selected medoids from P (i.e., D) would be many fewer than for LKMed.

5.3.3 K-means Based Sampling

Each iteration of K-medoids based clustering algorithms requires $O(K.(M-K)^2)$ distance computations, and thus may not be suitable for very fast scrolling. The cost of K-medoids based algorithms increase with an increase in page size or sampling rate. To address these computational constraints, in this subsection, we present three K-means approximated to K-medoids based sampling algorithms, which are computationally much faster, with $O(K.M)$ distance computations per iteration. One of these algorithms return representatives that are almost as good as LKMed.

K-means is also a partition based clustering algorithm, but it returns the mean of the objects in a cluster as cluster representative, unlike K-medoids that returns an actual object from the cluster. A cluster center obtained from K-means may not be

a real object in the cluster. To get a K-medoids approximation, we return K-actual objects that are nearest to each of these cluster centers. K-means is restricted to Euclidean distance functions, whereas K-medoids can support any distance function $\mathcal{V}$. K-means minimizes the following square-error criterion:

$$E_{K\text{-means}}(P) = \sum_{j=1}^{K} \sum_{p \in C_j} |p - m_i|^2 \tag{5.6}$$

Here, $E_{K\text{-means}}$ is the sum of square error of all objects in the dataset P; p is an object assigned to cluster C_j, and C_j is represented by m_j, i.e., mean of all the points in C_j.

5.3.3.1 Local K-means

Local K-means (LKMeans) is similar to LKMed, where we do not consider the effect of history H. We compute K cluster centers from page P using the basic K-means clustering algorithm [86]. For each cluster center, we compute the nearest neighbor from P and these tuples constitute the display set D. This algorithm takes O($K.M$) distance computations per iterations and is suitable for very fast scrolling. Note that for large datasets, more efficient K-means algorithms are available, such as grid based optimizations proposed in [77]. However, in our problem, since pagewise data is small, basic K-means would be quite fast and give the best quality result.

LKMeans has the same disadvantage as LKMed, discussed in Section 5.2.3 Figure 5.2 (a); that it may display tuples which are very similar to already shown tuples. To avoid redundancy, we now present two K-means based algorithm that take history in account.

5.3.3.2 Historical K-means

For Historical K-means (HKMeans), see Algorithm 4, we make similar modifications to LKMeans, as we did from LKMed to HKMed.

Algorithm 4 Historical K-means

Input: K: size of display set
P: a data page
H: displayed history
Output: D: display set from page P
Method:
1: $D \leftarrow K$ random tuples from P as initial cluster centers
2: map $NH \leftarrow NearestNeighbor(P, H)$
3: **repeat**
4: re(assign) each tuple in P to its closest cluster center in $D \cup H$
5: **for all** cluster centers d in D **do**
6: if d has empty assignments, split the largest cluster assignment in D into two equal parts and assign half of the tuples to d
7: **end for**
8: update cluster centers in D to the mean of its constituent instances
9: **until** no change
10: $D \leftarrow NearestNeighbor(D, P)$
11: return D

Algorithm 4 is similar to basic K-means clustering algorithm [86], with a difference in steps 2 and 4, where we need to consider the effect of history H. In step 4, we assign each tuple in P to the nearest cluster center in $D \cup H$. If computed, as in case of normal K-means, step 4 would require $O((|H| + K).M)$ distance computation per iteration. However, this can be computed more efficiently by using the nearest neighbor map NH, computed in step 2. In step 4, we should compute the nearest neighbors of P with respect to D only, and then see if these nearest neighbors are closer than the nearest neighbor from history H (i.e., map NH). Since the history is fixed, we do not need to repeatedly compute the nearest neighbor with respect to H. This efficient computation will require $O(K.M)$ distance computations per iteration in step 4, which is same as basic K-means. For step 2, we would need an additional $O(|H|.M)$ distance computations for computing the nearest neighbors of P with respect to H. In step 8, we update cluster centers in D, by taking the mean of all the tuples assigned to each cluster center in D. In step 10, to get a K-medoids approximation, we return K-actual tuples from P which are nearest to each of the

computed cluster centers in D.

To make the HKMeans algorithm terminate with desired number of representatives, we need to address the empty cluster assignment problem. In HKMeans, tuples whose nearest neighbor from history H is closer than the nearest neighbor in currently selected representatives D, do not play any role in determining the new cluster centers D for next iteration. Since the page size is generally small and history may be large, the cluster assignments may often become empty, causing the algorithm not to converge or to terminate with desired number of representative tuples. In order to solve this empty assignment problem, in steps 5-7, if there is any empty cluster in D, then we split the largest cluster in D into two equal parts and assign half of the tuples to the empty cluster.

5.3.3.3 Two Phase K-means

In our empirical experiments, we were expecting HKMeans to yield a lower *CIL* score as compared to LKMeans, but to our surprise, the *CIL* score from HKMeans was quite high; in fact it was quite close to random sampling. By observing individual pagewise *PIL* scores, we observed that even though for most pages LKMeans and HKMeans were comparable in terms of *PIL* score, there were pages where the *PIL* score of HKMeans was quite high as compared to LKMeans. This apparently high *PIL* score was due to selection of outliers as representative tuples, as shown in Figure 5.3 (a).

Figure 5.3 (a) shows the effect of initialization (step 1) on Algorithm 4. In HKMeans tuples whose nearest neighbor from history H are closer than the nearest neighbor in currently selected representatives D do not play any role in determining the cluster center for next iteration. As shown in Figure 5.3 (a), if we pick the initial representatives from outliers (i.e., cluster A and B in Figure 5.3 (a)) and have historical representatives as shown, then since most of the tuples are closer to histor-

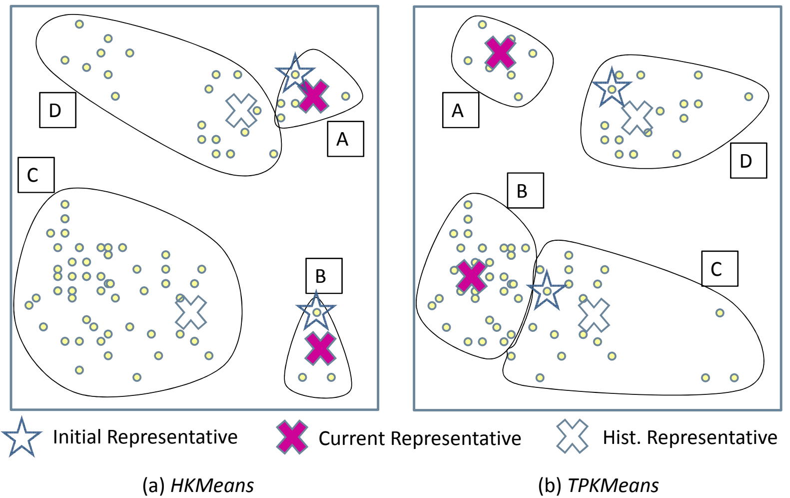


Figure 5.3: Effect of initial cluster centers on the two history based K-means sampling algorithm.

ical representatives, Algorithm 4 will converge with current representatives as mean of cluster A and B. Clearly, the selected representatives from B do not give a good overall impression as it is a cluster of outliers, and the representative from A is very similar to already displayed center from D. Unlike HKMed where the algorithm selects the medoids by computing the best possible swap in each iteration, HKMeans does cluster assignment simply based on nearest neighbor and thus may end up in a bad local optimum solution. Since the fixed centers from the history are not updated, and only a few tuples may play a role in selecting the new cluster center, HKMeans returns bad representatives quite often. This effect of initialization increases with increase in size of history (i.e., high sampling rate or size of truncated history L).

To address this problem of wrong initial cluster centers, we present a Two Phase K-means (TPKMeans) algorithm, which uses LKMeans in the first phase to compute proper initial cluster centers, and then in the second phase uses HKMeans algorithm to select final representative tuples. Figure 5.3 (b) shows that if we select proper initial cluster centers and then run Algorithm 4 using these as initial cluster center, we can avoid selecting outliers as representatives and also avoid showing redundant information.

TPKMeans is expected to perform quite well, as is confirmed through our experiments, because it initially chooses the centers from most dense locations and then moves to those portions of the data where there is no historical representative. The information quality from TPKMeans is quite close to HKMed. Computational complexity of LKMeans is O($t.K.M$), HKMeans is O($t.K.M + |H|.M$) and TPKMeans is also O($t.K.M + |H|.M$). Here t indicates the number of iterations.

5.3.4 Truncating History for Fast Performance

Using the entire history to minimize information loss, as discussed in Section 5.2.3, may not be practical from a system design perspective. This is because the size of history goes on increasing as the user continues to scroll. Ultimately, the history could become as large as the entire result set. Therefore, we consider using a truncated history that contains only information that a user has seen recently.

The truncated history can be computed by using the scroll log SL. For a page visit with sequence id *sid*, we scan SL backwards starting from *sid* and take the union of all *tupleList*s from the first L distinct pages that appear in a backward scan. If a page appears multiple times while looking for L distinct pages, we take the *tupleList* corresponding to the most recent visit of that page in SL. Of course, in a running system we do not need to go through this exercise repeatedly—it is straightforward to incrementally maintain the truncated history for the most recent L page visits, dropping the oldest one when a new page is visited.

The size of this truncated history, L, is a system parameter. We show experimentally that, as long as the value of L is not too small, having a large L does not lead to lower information loss. Thus, we can truncate history without hurting the quality of our results.

5.4 Evaluation

In this section, we report on the experimental evaluation of our variable-speed scrolling interface. We implemented the user interface and all sampling algorithms as described in Section 5.2 and Section 5.3, respectively. We performed experiments to compare the performance of all seven sampling algorithms in terms of their computation time and information quality. We also conducted a user study to measure the usability of a traditional full display vs. our sampling-based variable-speed scrolling interface, which we report in the next section.

5.4.1 Experimental Setup

We implemented our system in Java using the MySQL 5.5 RDBMS. Experiments were run on a dual-core Pentium 2.5 GHz PC with 2 GB of RAM. We used three datasets—Stock [70], Abalone [38] and ImageSegmentation [38]—for our user studies.

The Stock dataset has 2150 tuples, where each tuple represents stock details of a company for a day during year 1994-2003. It has 6 attributes, such as `Date`, `Starting Price`, `Max Price`, `Min Price`, etc. The Abalone dataset is generally used for regression analysis, where one predicts the age (age in years = number of rings + 1.5) of abalone from physical measurements, such as, `length`, `diameter`, `height` and different types of weights. This dataset has 4177 tuples with 8 attributes. ImageSegmentation is generally used for classification tasks. It has 2310 tuples from seven outdoor images: grass, path, window, cement, foliage, sky and brickface, where each tuple corresponds to a 3x3 region with 19 attributes. Each of the seven types of images has 330 instances.

5.4.2 Performance

In this subsection, we present the performance comparisons—both in terms of computation time and information quality—of all seven sampling algorithms on the Abalone dataset. While we conducted performance experiments on all datasets with similar results, we present our performance only on one dataset due to space constraints.

For our experiments, we assume a simple scrolling motion: a one-pass, constant speed, forward scrolling action from first to last page, with each page displaying a fixed number of tuples.

For these experiments, we assume the whole Abalone dataset as a sample query result. We present the effect of parameters such as page size, number of dimensions, sampling rate on the performance of these algorithms. Each of these plots are based on average readings of 25 simulation, where for each simulation we generate a different query result by selecting different random combination of attributes and ordering it by one randomly chosen attribute. The default parameters in these experiments are *page size* = 40 tuples per page, *number of dimensions* = 4 and *sampling rate* = 5 tuples per page. We have kept the size of truncated history, $L = 10$.

5.4.2.1 Computational Comparison

Figures 5.4 (a)-(c) shows the computation time of all the seven sampling algorithms with varying page size, number of dimension and sampling rate. We plot the average time for selecting tuples from each page, which is equal to the total time for a single pass simulation divided by the total number of pages. The average computation time gives a measure of the maximum allowed scrolling speed that a particular sampling algorithm can support.

Trends in Figures 5.4 show that K-medoids based sampling algorithms take considerably more time as compared to the other five sampling algorithms. As discussed

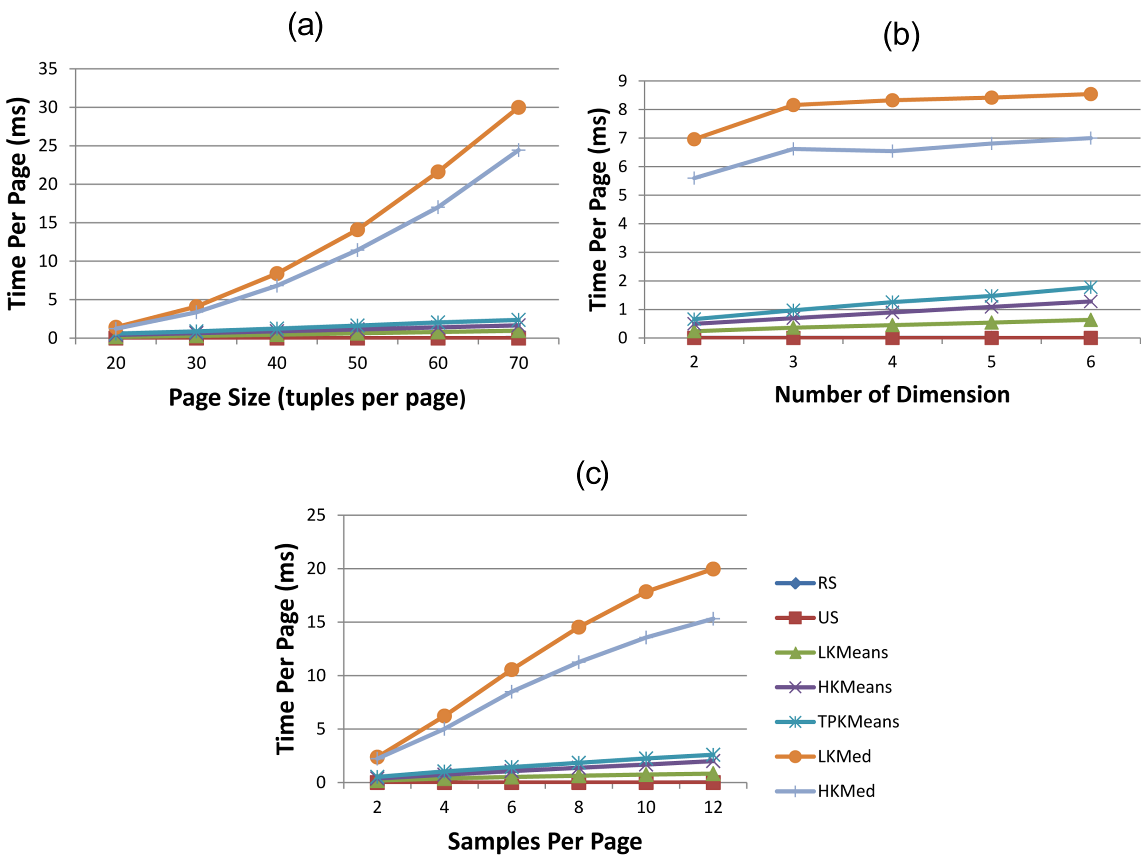


Figure 5.4: Computational time of all seven sampling algorithms with varying system parameters such as page size, number of dimension and sampling rate.

earlier, the computational complexity of LKMed, HKMed, LKMeans and HKMeans are O($t.K.(M-K)^2$), O($t.K.(M-K)^2+|H|.M$), O($t.K.M$) and O($t.K.M+|H|.M$), respectively. Here, t is number of iterations and O($|H|.M$) is the complexity of computing nearest neighbors with respect to history. HKMeans and TPKMeans have the same computational complexity. All three graphs show that the time taken by LKMed is greater than HKMed, even though in terms of computational complexity LKMed seems to be better than HKMed. As we had discussed earlier in Section 5.3.2.3, HKMed is faster than LKMed due to reduced amortized cost of *SwapCost* algorithm, in step 8 of Algorithm 3. HKMeans takes more time as compared to LKMeans because HKMeans requires additional computation for computing nearest neighbor in step 2 and during cluster assignment in step 4 of Algorithm 4. TPKMeans has slightly more computation time as compared to HKMeans because of the additional run of LKMeans needed in TPKMeans to get the initial cluster centers.

Figures 5.4 (a) and (c) shows that K-medoids based sampling algorithms cannot support fast scrolling for large page sizes or high sampling rates. K-means based sampling algorithms are quite fast even for reasonably large page sizes and high sampling rates. We will see later in Section 5.4.2.2 that the information quality obtained from TPKMeans is quite close to the K-medoids based sampling algorithms. Further, we can conclude from the three computation graphs that TPKMeans is suitable for very fast scrolling in all practical range of parameters.

Figures 5.4 (c) shows that for both HKMeans and TPKMeans sampling algorithms, the computation time increases rapidly with an increase in sampling rate. This increase is due to large size of history (truncated history) for which these algorithms need to compute the nearest neighbors. Moreover, in the presence of a large number of previous fixed centers, these algorithms require more iterations to converge. When the history is large, the cluster assignments do not converge quickly because in each iteration, only a few tuples, which are not near to any of the historical centers,

are used to compute the next step's cluster centers.

Figures 5.4 (b) shows that for K-medoids based algorithms the cost remains almost constant with the number of dimensions, but it increases in the case of K-means algorithm. This is because in K-medoids, we use pre-computed distance between all tuples, whereas in K-means, we need to actually compute these distances during each iterations. As the number of dimensions increases, each distance computation becomes more expensive.

5.4.2.2 Information Quality Comparison

The metric for the quality of information result presented is the *Cumulative Information Loss (CIL)* defined in Section 5.2. As we change the problem set up parameters, such as page size or number of dimensions, the value of *CIL* changes greatly. Furthermore, *CIL* is not scaled to anything—there is no way to tell whether a given absolute value is good or bad. What we can say, however, is that smaller values are better. For these reasons, we define the notion of inverted information loss below. The inverted information loss is what we plot in our figures.

Definition V.4. The **Inverted Information Loss** of an Algorithm A with respect to another Algorithm B is defined as:

$$IIL(A, B) = \frac{CIL_B(SL, \mathcal{R})}{CIL_A(SL, \mathcal{R})} \tag{5.7}$$

Figures 5.5 (a)-(c) shows the inverted information loss of each of the seven sampling algorithms with respect to random sampling, i.e., $IIL(X, RS)$, where X represents a sampling algorithm. These graphs are obtained by varying page size, number of dimension and sampling rate, respectively. All the three graphs show that HKMed gives the highest inverted information loss followed by TPKMeans and LKMed. In

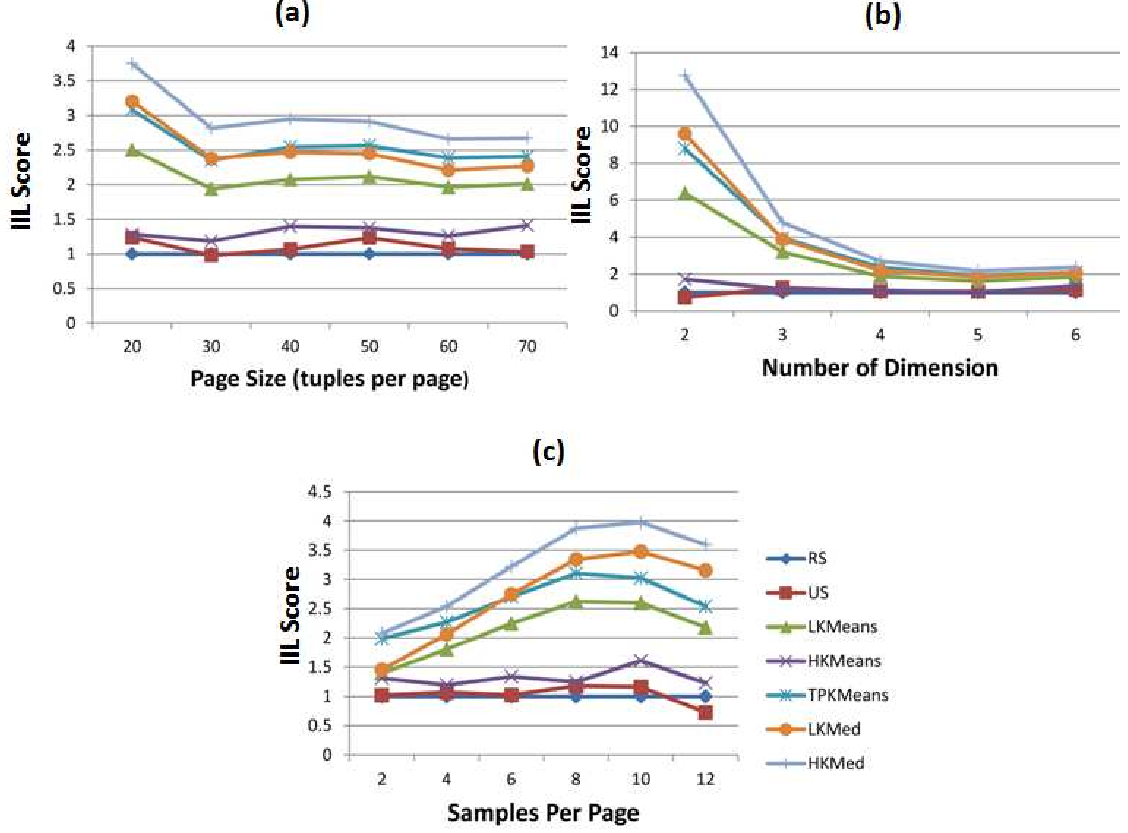


Figure 5.5: Information quality of all seven sampling algorithms with varying system parameters such as page size, number of dimension and sampling rate.

all the graphs, we can also see that the inverted information loss from HKMeans is quite low and is very close to random sampling.

The large inverted information loss difference between HKMeans and TPKMeans in Figures 5.5 (a)-(c) clearly indicates the effect of initial cluster centers on the history-based K-means algorithm (see Section 5.3.3.3 for details). When the history is large and/or dataset (i.e. page size) is small, HKMeans may end up choosing outliers as representative tuples. This problem is due to a choice of wrong initial cluster centers. Figure 5.5 (a) shows that as the page size increases for a fixed sampling rate, the performance of HKMeans and TPKMeans seem to improve—this is because of reduced history and a large dataset. When the page size is very small, LKMed seems to become slightly better as compared to TPKMeans—this is due to known problem

that K-means are effected by outliers more than K-medoid. By selecting proper initial cluster centers in TPKMeans, we are able to make its inverted information loss quite close to the ideal HKMed algorithm. From Figure 5.5 (c) we can see that as the history size increases with the increase in sampling rate, the performance of TPKMeans worsens, as compared to both the K-medoids based sampling algorithms. With a high sampling rate, the history size becomes larger, and thus affects the K-means based historical sampling algorithms.

Figure 5.5 (b) shows that the inverted information loss for all algorithms decreases with the increase in number of dimensions. This decrease is due to curse of dimensionality. As the number of dimensions increases, all the tuples in a page become quite far from each other and thus none of the clustering algorithms are very effective in giving a good overall impression. Even at a high dimension such as six, our sampling algorithms can give an inverted information loss of more than 2.

5.4.2.3 Effect of Truncated History

Figures 5.6 and 5.7 show the effect of truncated history's size on computational performance and information quality. In these experiments, we kept the other parameters at the default values and varied the size of truncated history L.

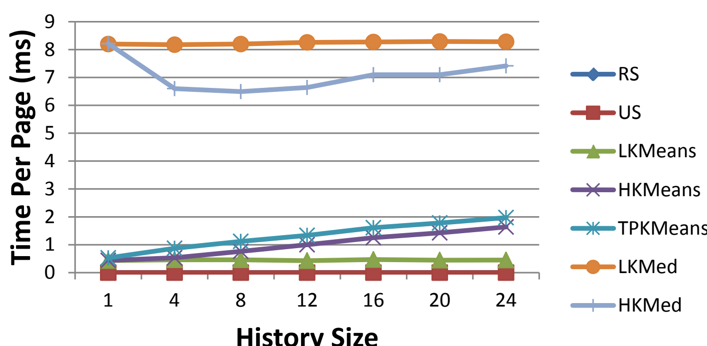


Figure 5.6: Effect of the user's browsing history size on computation time.

Figure 5.6 shows that when we have a very small history, i.e., $L = 1$, the two

K-medoids and the three K-means based algorithms have similar computation time. As we increase history, we see that computation time of HKMed becomes better than LKMed, but as we increase L further, the time to compute the nearest neighbor increases and the computation time of both the algorithm seems to converge. For HKMed, the dip in computation cost is due to a reduced amortized cost of the *Swap-Cost* algorithm, as discussed earlier. The computation time of history based K-means algorithm goes on increasing because of the nearest neighbor computation.

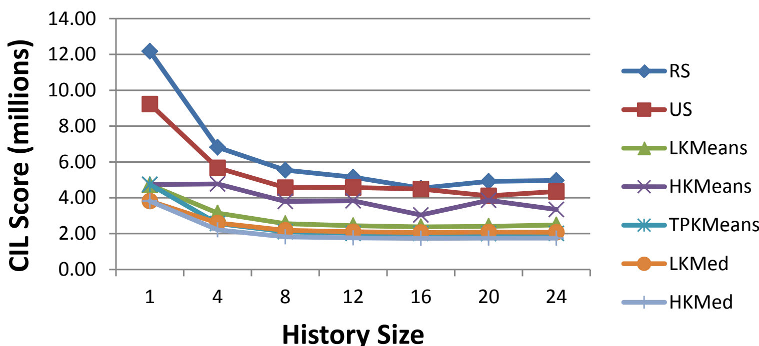


Figure 5.7: Effect of the user's browsing history size on information quality.

Figure 5.7, shows that the *CIL* score of all sampling algorithms become stable for L greater than 10 (the information gain also becomes constant because it is ratio of two *CIL* scores). Based on this, we used $L = 10$ for all other experiments. Note that we plotted the *CIL* score in Figure 5.7 rather than *Inverted Information Loss*. We did so to make clear that the flattening of benefits with larger L is not just a flattening of relative benefits between the algorithms, but rather an absolute flattening.

5.4.2.4 Summary Recommendations

K-medoids based sampling algorithms are fast enough to support normal browsing in all parameter ranges. Studies have shown that for a user interface to seem instantaneous from a cognitive perspective, the system has a bound of reacting within 100 ms [14]. Our own studies showed that users often spend less than 100ms on a page

when they are browsing quickly. Some users spend as little as a few tens of milliseconds on each page. We see from Figures 5.4 (a)-(c) show that the computation time of all proposed sampling algorithms is well within the 100 ms limit. The LKMed and HKMed algorithms do start to take time that may become noticeable when the page size is large and the sampling rate is very high. In general, most display screens can show 20-40 tuples per page, and when a user scrolls quickly, our system is expected to show only a few tuples per page (i.e a low sampling rate).

In short, we should choose the sampling algorithm based purely on information quality. In other words, HKMed can be used for scrolling in most parameter ranges. If computation time, especially for the server side, becomes a concern, TPKMeans is the preferred algorithm since it sacrifices slightly in information quality compared to HKMed, but is much faster to compute.

5.4.3 User Study

The goal of our user study is to measure the usability of a full display vs. the sampling-based variable-speed scrolling interface. Usability can be measured in terms of users' ease-of-use, efficiency and quality of response to a given task.

We requested 8 users from our university to participate in sessions comprising two similar tasks on each of the three datasets—STOCK, ABALONE and IMAGESEGMENTATION—once using the condensed display and once using the full display. To reduce the effect of learning, we created the following four sequences of sessions that distribute and reorder the tasks, datasets and user interfaces evenly. We asked two users to perform each of the following session sequences:

- **Set0:** S1F, A2S, I1F, S2S, A1F, I2S
- **Set1:** S2S, A1F, I2S, S1F, A2S, I1F
- **Set2:** S2F, A1S, I2F, S1S, A2F, I1S

- **Set3:** S1S, A2F, I1S, S2F, A1S, I2F

For a session code 'XYZ', X indicates the dataset's name i.e., S for Stock, A for Abalone, I for ImageSegmentation; Y indicates task code, i.e, Task 1 or 2 from the dataset X (two tasks were designed for each data set, as we explain below); and Z indicates the display mode, i.e., F for full display and S for condensed display. We indicate the users as $U_0, U_1, ..., U_7$. We gave user U_i the s session sequence $i\%4$. For each session we measured the time taken by the user to finish the task, the quality of user's response and user's scrolling pattern.

For the condensed display we used the HKMed sampling algorithm. To measure the quality of our samples, we turned off the full display mode while the users scrolled backward. In our user studies, when the users scrolled backward, we showed those tuples from a page that were shown in the most recent forward scroll through that page. We performed all user studies with page size = 30 tuples/page.

For all the three tasks we have performed linear mixed model statistical analysis [109]. We use Display type as fixed effect and User ID as random effect. Computing p-values for mixed models aren't as straightforward as they are for linear models. The most popular way to obtain p-value is to use the *Likelihood Ratio* test as a means to attain p-values. The logic of the likelihood ratio test is to compare the likelihood of two models with each other. First, the model without the factor that one is interested in (the null model) and then the model with the factor that one is interested in. By comparing these two models, one can determine whether the factor one is considering is significant or not. We use ANOVA to compare the two models.

5.4.3.1 Interesting Patterns (Stock)

There are many applications where users are interested in finding trends or patterns, such as increasing or decreasing trends, periods with high and low data variance, cyclical periods, etc., over time series data.

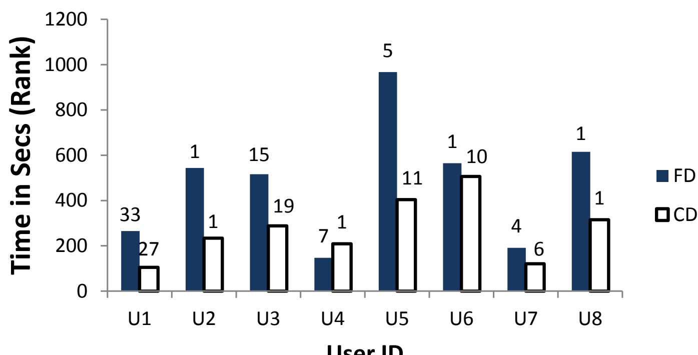


Figure 5.8: In this task users had to find the two pages with minimum and maximum information variation in stock prices. Statistical analysis shows that users response quality is almost similar for both the interfaces. But condensed display affects the time taken by ($\chi^2(1) = 8.78,\ p = 0.003$), lowering it by about 272.5 ± 68.2 seconds.

We measured the usability of our system in identifying interesting patterns through the Stock dataset. We asked the users to scroll through the dataset, which was ordered with respect to time, and identify pages with minimum and maximum variance, defined as the difference between the maximum value of `Max Price` column and minimum value of `Min Price` column. Figure 5.8, shows the time taken by all 8 users using full display (FD) and condensed display (CD) interface. To evaluate the user's response quality, we computed the actual variance of all the pages and then measured the quality of user's response with respect to the actual variance. For example, if user's max variance page is 5th highest in terms of actual max variance, then we say that quality is 5. At the top of each time bar, we have displayed the quality of user's response. Statistical analysis shows that users response quality is almost similar for both the interfaces. But condensed display affects the time taken by ($\chi^2(1) = 8.78,\ p = 0.003$), lowering it by about 272.5 ± 68.2 seconds. Users were able to finish the task around two times faster using condensed display as compared to full display.

5.4.3.2 Simple Regression Task (Abalone)

There are many applications where given certain feature measurements, one is interested in finding the expected value for a missing feature. For example, given descriptions of a car or a house on sale, one would like to have an expected guess of what should be a reasonable price.

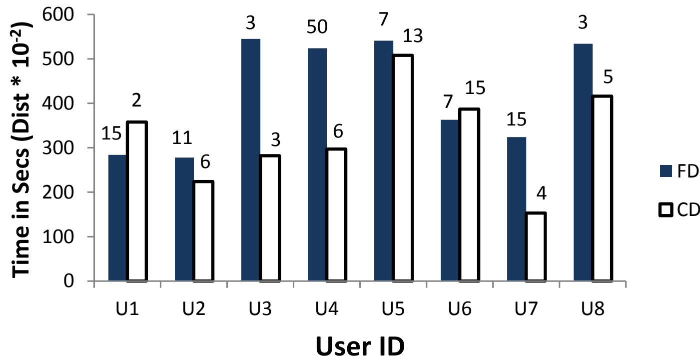


Figure 5.9: In this task users had to predict a missing attribute value for a given tuple using manual regression. Statistical analysis shows that users response quality is almost similar for both the interfaces. But condensed display affects the time taken by ($\chi^2(1) = 6.18,\ p = 0.013$), lowering it by about 125 ± 40.9 seconds.

We measured the usability of our system in assisting regression tasks using the Abalone dataset. As compared to general housing or automobile datasets, the abalone dataset is less noisy and thus it is comparatively easy for a user to manually perform the regression task. In this task, we picked two instances as test data from the Abalone dataset and removed their `length` and `shell weight` features. We sorted the data according to `length`. We asked each user to fill in these two missing features for one instance using the full display interface and other using the condensed display interface. For the user studies, we removed the `diameter` feature from the dataset as this feature was very closely correlated with `length`, and thus users could predict the `length` closely using this feature itself. Other features were not that closely related with the `length` feature. We measured each user's response quality by computing the

Manhattan distance of user's response for the two missing features with respect to the actual values. The results are shown in Figure 5.9. The bars show the time taken by each user and the values on top of the bar indicates the Manhattan distance of user's response to the actual missing values. For most users, both response quality and time are better using the condensed display interface than the full display interface.

Since this task involved a larger number of features and comparatively larger dataset, this was very indicative of how users struggle with large relational query results. In this task, the tuples that were similar to task 1 appeared in the first half of the ordered data, whereas for task 2, they appeared in the second half. In the condensed display interface, users were quick to appreciate a good overall impression of each page through a few selected samples and compare with the test data. We can see that when task 2 was given to users U_3,U_4, U_7 and U_8 using the full display, they took much more time as compared to users U_1,U_2, U_5 and U_6 who had done the same task using condensed display.

Statistical analysis shows that users response quality is almost similar for both the interfaces. But condensed display affects the time taken by ($\chi^2(1) = 6.18,\ p = 0.013$), lowering it by about 125 ± 40.9 seconds. For the task that required users to browse through many pages, users able to do it two times faster using condensed display interface compared to full display.

5.4.3.3 Discriminating Columns (ImageSegmentation)

In many applications, a user does not have a priori knowledge of the underlying data. Thus, they try to figure out columns which can help them make discriminating judgements by browsing through the data, e.g. identifying features that can help them the most in classifying houses, cars, hotels etc., at different price ranges. Although each house, car etc., can have a large number of features, one needs to identify features that are important for making decisions, a task that is often best done by

visual inspection. By knowing good discriminating features and suitable ranges, a user can either form a new, more precise query or easily predict the class of a new data record.

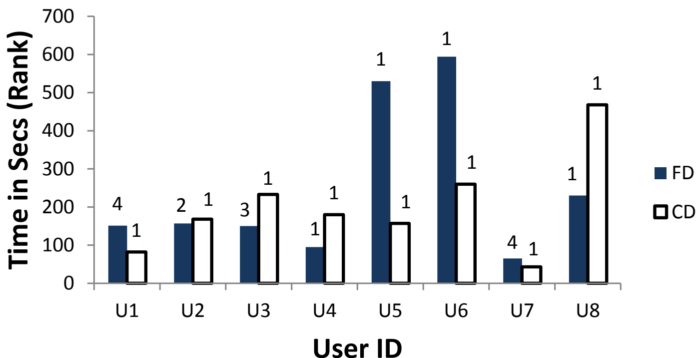


Figure 5.10: In this task users had to find the maximum discriminating column for data classification. Statistical analysis shows that time taken is almost similar for both the interfaces. But condensed display affects the users' ability to find the most discriminating column by ($\chi^2(1) = 5.31,\ p = 0.021$), increasing the quality by about 1.13 ± 0.45 rank.

We evaluated user performance on a task of identifying good discriminating columns using the ImageSegmentation dataset. This data set has 19 columns, many of which are rather technical in nature. 19 columns is too many for use in a simple scroll without horizontal panning. Therefore, we selected 7 out of 19 attributes that were easy for non-expert users to understand. Specifically, we selected attributes 10–16, i.e., `Mean Intensity`, `Raw Red`, `Raw Blue`, `Raw Green`, `Excess Red`, `Excess Blue` and `Excess Green`. The were 330 images in the dataset for each of seven different types, such as brickface, sky, and grass. We asked users to find the most discriminating columns for image types: brickface and sky; and brickface and grass. We measured the quality of a discriminating column using the Fischer Linear discriminant.

$$\mathcal{J}(C_k, i, j) = \frac{|m_{ik} - m_{jk}|^2}{{s_{ik}}^2 + {s_{jk}}^2} \tag{5.8}$$

Here, $\mathcal{J}(C_k, i, j)$ is a measure of column C_k's discriminating ability for class i and

class j; m_{ik} and s_{ik} represents the mean and standard deviation of class i in column C_k.

Figure 5.10 shows the user's response time and quality of answer at top of the time bar. The quality is measured in terms of the true rank of the user's response column using the discriminating score defined in Equation 5.8. All users were able to identify the best column using condensed display interface, but they made errors using the full display because of noisy data and too much information. Statistical analysis shows that time taken is almost similar for both the interfaces. But condensed display affects the users' ability to find the most discriminating column by ($\chi^2(1) = 5.31,\ p = 0.021$), increasing the quality by about 1.13 ± 0.45 rank.

5.4.3.4 Relationship between Sampling Rate and Scrolling Speed

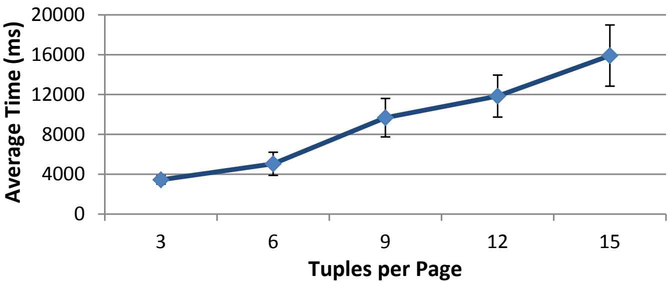


Figure 5.11: Relationship between users average scrolling speed and suitable sampling rate

The results in this section shows how verbosity affects users reading time. We conducted this study over 6 users. Each user was asked to scroll through 5 pages of a relation, where each page had 30 tuples and 5 columns, and each cell of the relation was randomly assigned an integer value between 0 and 999. Users were tasked to locate a predetermined value on each page of the dataset (this forced the user to read each page completely). In Figure 5.11, we plot the average time spent per page and standard error, while varying the verbosity of our display from 3 to 15 tuples per page. In addition to the linear, inverse relationship between reading speed and

tuples-per-page, we also observe that users take approximately 1 second to read a tuple on average for our dataset. When browsing quickly, users are often not looking at all the data values—rather, they may focus on the single sorted attribute. In such cases, they can browse much faster, possibly spending less than 100 ms per tuple. From Figure 5.11, one can see that as verbosity increases, the user takes more time to read a page.

5.5 Conclusion

In this chapter, we describe the design of a system for browsing relational data by scrolling through it at a high speed. Rather than showing the user a fast-changing blur, the system presents the user with a small number of representative tuples. Representative tuples are selected to provide a "good impression" of the query result. We discussed several scrolling-aware algorithms that can select representative tuples of a higher quality as compared to random sampling. We show that the information loss to the user is limited, even at high scrolling speeds, and that our algorithms can pick good representatives fast enough to provide for real-time, high-speed scrolling over large datasets. Through extensive user study we show that our variable-speed scrolling interface provides a better browsing experience.

CHAPTER VI

Related Work

6.1 Faceted Search

Faceted search was originally introduced for browsing image and text collections [35, 112, 30, 105, 31]. Recently, there have been efforts on creating a faceted search interface for structured data [15, 67, 33]. Although the basic faceted interface is a very simple browsing model, the usability of faceted interface depends highly on the way the facets are chosen or the way search results are presented [94]. There are many interesting research areas in faceted search, such as identifying interesting facets and facet values [15, 67, 33, 98], automatic construction of faceted interface [30, 78, 105], discovering OLAP type of interesting information [19, 33, 110].

Faceted interface has a very simple modular structure. As a result, it can be easily extended using add-on extensions. One of the most successful add-on extensions is the integration of faceted interface with keyword search [49, 110, 33]. Most of the research that has been done in faceted navigation can be seen as addressing two main aspects: *navigational* [15, 67, 33] and *statistical* [19, 110, 33] aspect. In this dissertation, we enhanced both navigational and statistical aspect of faceted interface. Existing research that addresses the statistical aspect has mainly focused on OLAP type of information discovery. OLAP is quite different compared to clustering based summarization done in SFI View. In our ICSummary extension, we use concepts from

OLAP [100, 99, 110, 39] to create the three likelihood categories based on observed and expected counts. There are many innovations to faceted navigation that have been done by different e-commerce sites, which are not research publications. These innovations are summarized in surveys [50, 106, 51].

As Hearst mentions in her two survey papers [50, 51] the problem of ambitious information discovery is a major unaddressed limitation of faceted navigation. Both academic research and e-commerce sites have focused more on addressing the design issues. In this dissertation, we showed through many examples the limitations of faceted navigation in relation to information discovery. Our two query panel extensions, namely ICSummary and SFI View, assist users to perform ambitious information discovery. Based on our proposed user-interaction techniques, such as pop-up window and two-phased faceted browsing, it is possible to provide other types of useful feedback information to users.

6.2 Lookup vs. Exploratory Search

There are two types of search: *lookup* and *exploratory* [107, 108]. In lookup search a user has a specific and well-defined search goal. In contrast, in exploratory search the user's goal is to learn about an unfamiliar information landscape through diverse, ill-defined and open-ended search queries. In exploratory search the user's aim is to gain a comprehensive understanding of data that will enable him to make more informed lookup queries. Users often have to go through an exploratory search phase before they can form informed lookup queries. The survey paper [36] presents a nice summary of how researchers have evaluated benefits of faceted browsing for both lookup and exploratory search. The survey paper [36] summarizes many user studies that have been performed to show that for lookup search faceted navigation is much better compared to other navigation techniques. In this dissertation, we showed that the traditional faceted search interface is not suitable for certain types

of exploratory search, and we presented three add-on extensions to facilitate those exploratory search.

6.3 Parallel Coordinates

In this section, we show the close relationship between parallel coordinates [59, 60] and faceted navigation. Faceted navigation can be seen as a simplified presentation of parallel coordinates, which is commonly used to analyze multivariate data. Although parallel coordinates can show interaction between attributes, it cannot scale to high-dimensional datasets or datasets with many tuples. The summary digest that is shown in faceted interface is a modified form of parallel coordinates that can scale to large dataset, but it cannot show interaction between attributes. The two add-on extensions, namely the ICSummary and the SFI View that we presented in Chapter III and Chapter IV respectively, provide much better attribute interaction information compared to even the parallel coordinates.

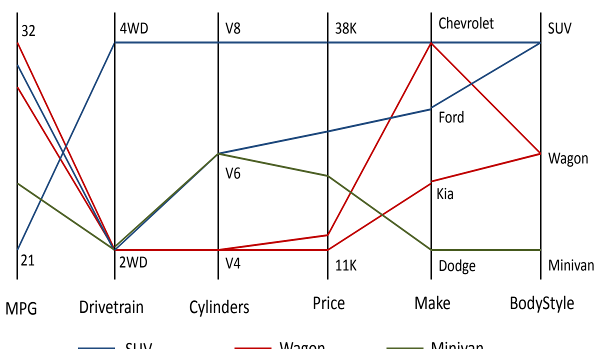


Figure 6.1: Parallel coordinates with matching.

In parallel coordinates, we represent an n-dimensional data using n parallel axes, where each axis represents a dimension, and data points are represented as a polyline with vertices on the parallel axes. In Figure 6.1, we show a parallel coordinate representation of the data shown in Table 3.1. (For ease of visualization, we have shown the polylines for each body type using different color code). Using parallel coordi-

nates, users can easily see the domain of all the displayed numerical and categorical attributes. By following a particular polyline, they can see how one tuple compares with respect to other tuples. By seeing the polyline trends between two adjacent axes, they can infer the dependency between the two attributes.

In parallel coordinates, each axis can have at most two neighboring axes (one on the left, and one on the right). For a n-dimensional data set, at most n-1 relationships can be shown at a time. The information quality in parallel coordinates is very sensitive to the relative ordering of axes. If we place attributes that are independent as adjacent axes, then there will be not be any specific polyline patterns. There is large amount of information loss because parallel coordinates can only show 2D interactions, and not the n-dimensional interactions.

Parallel coordinates are effective for low dimensional dataset having few tuples. In order to deal with large number of tuples other versions, such as parallel sets [73], have been proposed. Parallel sets uses width of lines to show interaction between axes. If some categorical attributes have many values, then even such attributes cannot be easily visualized using parallel coordinates. The user's screen size determines the number of dimensions that can be laid out as parallel axes, which is typically less than 10 dimensions. If there are many tuples (of the order of few hundreds or thousands), then the visualization will become cluttered. Some important categorical attributes such as `Make` and `Model` have more than 50 and 300 values respectively. If we try to show them as axes, then they will waste valuable screen space and are also very hard to browse.

Apart from parallel coordinates, one can use other ways to summarize structured data. For example, as shown in Figure 6.2, one can remove all the polylines and show the vertices on parallel axes with their incident polyline count. For numerical attributes with large number of values, such as `MPG` and `Price`, one can discretize them in few bins and then treat the discretized ranges as categorical values. We call

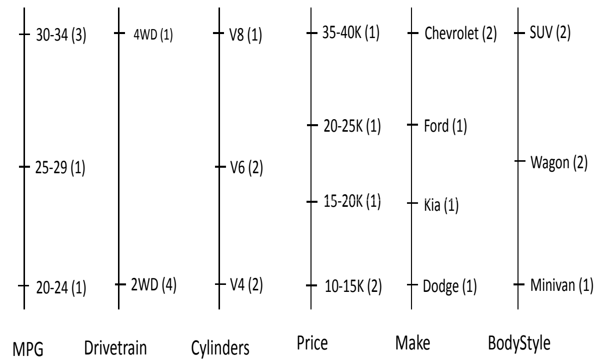


Figure 6.2: Parallel coordinates with aggregation.

this summarization as *parallel coordinates with aggregation*, and the one shown in Figure 6.1 as *parallel coordinates with matching*.

The summarization done in Figure 6.2 has many advantages over the summarization done in Figure 6.1. It can summarize large dataset without getting cluttered. Users can easily see all the attribute values that are present in the dataset, along with their counts, grouped by the corresponding attribute. The greatest limitation of aggregation based parallel coordinates is that users can no longer see the flow of information across attributes, which they could easily see through polyline trends in traditional parallel coordinates.

Implicit selections are much easier to see in parallel coordinates with matching. For example, if a user wants to see how Wagons compare with Minivans or SUVs, then the user can just compare the corresponding polylines. For example, in Figure 6.1, selecting Wagon leads to implicit selection of attribute values `Price` equals 10-15K, `MPG` equals 30-35, `NumCylinders` equals V4 etc. Even in parallel coordinates with polylines the user has to go through the attributes in a sequential order from explicitly selected attribute to the destination attribute to get the implicit dependencies. If the visualization is cluttered or polylines are not properly color coded, then one cannot learn the dependencies. Parallel coordinates are primarily meant for seeing the dependencies between adjacent axes only. Typically, they show polylines with same color, and in that case one will not be able to see the flow of information across

multiple attributes.

There is exponential number of ways users might be interested to look at the available information. Instead of creating summarization that shows all possible summarization at once, we can create a summarization that shows only the relevant information as desired by the specific user. In Figure 6.1, all possible interactions between attributes are shown at once, and as a result the visualization cannot scale to large dataset. Instead we can have a clean summarization as shown in Figure 6.2, and then show attribute interactions as requested by the user. We achieved this goal by extending the query panel of faceted navigation with ICSummary and SFI View.

6.4 Interaction Between Attributes

In this dissertation, many of our contributions rely on finding important interactions between attributes (or facets). Measuring attribute interaction is a part of broader feature selection problem [45, 82, 72] in machine learning. In databases, attribute interactions are often measured in form of functional dependencies [87, 56, 37, 5, 58] and referential integrities. Although standard feature selection can find the interaction between features, a Bayesian network [65, 52] can give a more accurate description of feature interactions by giving probabilistic dependencies between features.

6.5 Query Formulation

In Chapter III and Chapter IV, we presented two modifications to existing faceted interface that can aid users in their query formulation. Query formulation is a challenging task, especially formulating structured query for structured data. Although faceted search has made this task easier by allowing users to easily create selection queries over few given queriable facets. But there are many other interesting

techniques for query formulation, such as query by example [116, 117], visual query interface [91, 90, 85, 80, 48, 11], keyword search [6, 54, 55, 10, 103], query recommendation [115, 89, 21], iterative querying [12, 79], aids for query construction [92, 71], forms [63, 28, 64].

6.6 Search Result Presentation

The need for fast browsing has been established through numerous user interface studies [49, 41, 16, 18, 17, 13], done in the area of Human Computer Interaction. Browsing large result sets has been a long time common pain for both information retrieval and database community. There are many existing techniques that have been proposed to help users do fast browsing through such result sets. All the following existing techniques can be used to improve browsing over the result panel:

Data Summarization: Large relational data is often summarized using data warehousing and OLAP technology [42, 100, 24]. There are also many data mining techniques, such as clustering [77, 43, 114, 62, 20] and decision trees [23, 27], that can group data into meaningful groups according to some user given notion of similarity. For relational data some clustering based fast browsing interfaces are [104, 81, 111]. Many interesting data summarization techniques are presented in [47]. Web search result are often summarized and combined with some information visualization techniques to support fast browsing [75, 76].

To browse clustered data, we need to label the clusters. Cluster labeling is commonly done in text clustering [88, 22] where one tries to select descriptive labels for the clusters obtained through a clustering algorithm. The labels often consist of a set of important terms from the clustered documents, so that it not only summarizes the central concept of the cluster but also differentiates it from other clusters. This kind of term based labeling is also available in Apache Solr's faceted interface. Since

existing clustering or cluster labeling algorithms are not meant for specific user goal, they are often not very usable for a specific task. In Chapter IV, we created IUnits, which is similar to cluster labeling, for structured data. Our main goal in creating IUnits was to compare facet values selected in pivot facet. For the same query result, users can get totally non-comparable set of IUnits based on their choice of pivot facet.

Data Categorization: Data categorization is also a type of data summarization technique that was commonly used prior to faceted search [23, 27, 68]. Unlike faceted search that allows users to place selection conditions independently from any of the available queriable facets, the data categorization techniques restrict users to follow a fixed hierarchical category structure. Users can determine whether a category is relevant or not by simply examining its label, and then explore just the relevant categories and ignore the rest. Although data categorization techniques could not become very successful because of their constraints on users exploration path, they are designed more closely based on users' preference compared to faceted search.

Top-k Result Set: Evaluating top-k query has been an important research aspect in all areas of information retrieval. Many results may satisfy a user's query. But by ranking the result set, these ranking algorithms enable the user to quickly locate the desired information by just browsing the top few results. For relational data, top-k ranking algorithms have been proposed in [9, 25].

Top-k Diverse Set: It is hard to design an automatic top-k ranking algorithm that can simultaneously satisfy the information need of all users, because rankings change with change in users' preference. To satisfy most users, algorithms for computing top-k diverse set has been proposed, for example web queries [8, 40] and SQL queries [27, 96, 113].

Automatic Zoom-in and Zoom-out: In Chapter V, we presented sampling algorithms that can automatically reduce the amount of information that is shown to

the user based on user's browsing speed. Users can browse quickly if they have less information to see. The variable-speed scrolling interface we presented in Chapter V is similar to the variable-speed scrolling interface, proposed by Igarashi et al. [57], for text documents. While browsing text documents, our eyes follow visual hints, such as different font sizes, section headings, graphics, knowledge of structural outline etc. When a user scrolls too fast through a text document even the prominent visual hints start getting distorted and the user loses track of her position in the document. In [57], the authors assume that there is a maximum threshold scrolling speed beyond which things would start getting distorted. When a user scrolls below the threshold speed, their system shows full information, but when the scroll speed is beyond the threshold speed, the system starts showing only important markers and hides the finer details. The markers are easy to identify in text documents because of different font sizes, section headings etc. They had also tried to use their technique for structured data, such as dictionaries, but they found out that it was not useful because everything was very homogenous and markers were hard to identify.

Recommendation Systems: Recommendation systems are commonly used in many e-commerce sites to support fast browsing [97, 95, 101, 7, 53]. Recommendation systems can take users preference into consideration to find items that would be of interest to a specific user. The recommendations are given based on the user's past behavior and/or behavior of similar users.

6.7 Information Visualization

We have designed the add-on extensions to provide insightful feedback to users about the database content. There are many popular information visualization techniques, such as tag clouds [76, 75], histograms, scatter plots, tree maps [66, 102], parallel coordinate plots [59, 60], etc., that can convey abstract information to users

in intuitive ways. These techniques use visual representations and interaction techniques to help users see, explore and understand large amount of information at once. Some of these techniques convey important information through distinct text properties, such as font, color, etc. We have used some of these visualization techniques in our add-on extensions to improve their feedback quality.

CHAPTER VII

Conclusion and Future Work

7.1 Contributions

During the last few decades, database researchers have greatly improved the capability of databases both in terms of performance and functionality. But users' limited knowledge of what is contained in various databases and lack of technical expertise makes it very hard for them to directly interact with databases in a meaningful way. Managing a database system thus usually requires an army of database administrators, consultants, and other technical experts all busily helping users get data in and out of the database. To make databases more accessible to users, researchers in the past few years have started to look at the usability aspect of databases [61], such as usability in data storage, data access, data presentation, etc. In this dissertation we have focused on improving the usability in data presentation. We use a combination of machine learning, data mining, statistics, and information visualization to help users effectively explore through huge volumes of relational data.

There are many popular data navigation techniques, such as clustering, ranking, faceted navigation, etc., with each one having its own strengths and weaknesses. Instead of focusing on a single navigation technique, we have intelligently combined multiple such techniques in a way that enhances their strengths and cancels their weaknesses.

We have presented various challenges associated with faceted navigation, and presented three add-on extensions, namely ICSummary, SFI View and Skimmer, to address those challenges without destroying the fundamental structure of faceted navigation. The first two add-on extensions are for the query panel to facilitate exploratory search. The third add-on extension is for the result panel to make it more users personalized and efficiently browsable. The add-on extension that we have created for the result panel can be used in any structured data result panel, and is not restricted to faceted navigation.

Implicit Choices Summary (ICSummary) — In complex databases, when a user explicitly selects a facet value, the user has very limited understanding of how values are implicitly selected in other facets. This understanding is crucial to understand a chosen exploration path. ICSummary is an automatically computed summary of important implicit facet value selections. ICSummary is integration of following four navigation techniques: OLAP, categorization, faceted navigation and ranking.

Summarized Facet Interaction View (SFI View) — In the traditional faceted interface users cannot see in the query panel the overall space of available exploration paths, and thus end up choosing an inferior exploration path. We have designed a two-phased faceted interface, known as TPFacet, which can provide facet wise comparison of top-k available exploration paths in the form of SFI View. SFI View is integration of following three navigation techniques: clustering, faceted navigation and ranking.

There are various design and algorithmic challenges in creating and presenting additional feedback information to users in the form of ICSummary and SFI View. We have proposed changes to the faceted navigation interaction model so that users can see these types of additional information, without having a cluttered search interface.

Variable-speed Browsing (Skimmer) — Users cannot browse quickly in the existing result panel because it shows a large amount of information. To help users

get a quick sense of data, we have designed a novel variable-speed scrolling interface, called Skimmer, which provides users a good impression of the data through selected representative tuples that are chosen based on the users' scrolling speed and browsing history. In Skimmer, we integrate scrolling with clustering to dynamically select the informative representative tuples.

7.2 Future Work

Some illustrative ideas for future work includes:

Improving the Result Panel in Faceted Interface — There are many existing techniques to efficiently browse through large result sets, such as recommendation systems, data summarization, data categorization, skylines, etc. During faceted navigation users often have thousands of tuples in the result panel and they do not know how to further constrain the query to get to their desired subset of tuples. The existing result panel shows result tuples in either random order or by using some simple global ranking function. One can adapt and develop techniques to create a more personalized result panel by dynamically taking into consideration the user's selections in the query panel.

Querying, Browsing and Analyzing Big Data — Faceted navigation is not the only way to explore databases. There are many other interesting data access techniques, such as OLAP, keyword search, NLP, relevancy-ranked, etc. One can think of ways to make these data access techniques more usable. For example, we use the autocompletion feature, provided by most web search engines, as a very useful extension of basic keyword search. To aid non-technical users one can create new query tools that will help them form precise and complex queries with less technical effort. At present, it is difficult for normal users to do data-analysis because they cannot easily extract relevant data from the database and run suitable data-mining

algorithms. In this dissertation we presented data-exploration extensions for faceted navigation. One can develop similar extensions for other navigation techniques.

Data Exploration in Text and Graph Data — This dissertation is mostly focused on relational or structured data. One can design data-exploration tools for unstructured data (such as text documents) and graph data (such as social networks). There are users who have large amount of unstructured text documents or graph data, but they lack technical expertise to process and analyze such data. Although such users might have theoretical data-analysis knowledge, they often do not have the required expertise to process such data through their own created software tools. Some data-access tools provide ease of data access but are not suitable for complex data analysis, and vice-versa. One can combine existing data-access tools for text documents and graph data so that these non-computer science experts can easily access and analyze their data.

Workflow in Healthcare and Similar Domains — In order to build the various add-on extensions proposed in this dissertation, we had studied the user workflow in relation to e-commerce. Since there are many popular e-commerce sites with many users, and their interface is publically accessible, the workflow of e-commerce users and their need is quite well-understood. However, there are many other domains, such as healthcare, scientific and business intelligence, etc., where users are not looking for e-commerce type of information. To design good data access and data presentation techniques for these domains, we need to gain more detailed understanding of their need and workflow. It would be an interesting future work to investigate the development of effective data-navigation techniques for such important domains, where advanced exploration methods have not yet been considered.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] (2013), Apache solr, `http://lucene.apache.org/solr/`.

[2] (2013), Cars.com, `http://www.cars.com`.

[3] (2013), Yahoo used-car, `http://autos.yahoo.com/used_cars.html`.

[4] (2014), The apache commons mathematics library, `http://commons.apache.org/proper/commons-math/`.

[5] Abiteboul, S., R. Hull, and V. Vianu (1995), *Foundations of databases*, vol. 8, Addison-Wesley Reading.

[6] Aditya, B., G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, P. Parag, and S. Sudarshan (2002), Banks: Browsing and keyword searching in relational databases, *VLDB*.

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

[8] Agrawal, R., S. Gollapudi, A. Halverson, and S. Ieong (2009), Diversifying search results, *WSDM*.

[9] Agrawal, S., and S. Chaudhuri (2003), Automated ranking of database query results, *CIDR*.

[10] Agrawal, S., S. Chaudhuri, and G. Das (2002), Dbxplorer: A system for keyword-based search over relational databases, in *ICDE*, pp. 5–16, IEEE.

[11] Angelaccio, M., T. Catarci, and G. Santucci (1990), Qbd*: A graphical query language with recursion, *Software Engineering, IEEE Transactions on*, *16*(10), 1150–1163.

[12] Anick, P. G., and S. Tipirneni (1999), The paraphrase search assistant: terminological feedback for iterative information seeking, in *SIGIR*, pp. 153–159, ACM.

[13] Baeza-Yates, R., B. Ribeiro-Neto, et al. (1999), *Modern information retrieval*, vol. 463, ACM press New York.

[14] Bailey, B., et al. (2001), The Effects of Interruptions on Task Performance in the User Interface, *INTERACT*.

[15] Basu Roy, S., H. Wang, G. Das, U. Nambiar, and M. Mohania (2008), Minimum-effort driven dynamic faceted search in structured databases, *CIKM*.

[16] Bates, M. (1986), Subject access in online catalogs: A design model, *ASIS J.*

[17] Bates, M. (1989), The design of browsing and berrypicking techniques for the online search interface, *Online Information Review*.

[18] Belkin, N., R. Oddy, and H. Brooks (1982), Ask for information retrieval, *Journal of Documentation*.

[19] Ben-Yitzhak et al., O. (2008), Beyond basic faceted search, in *WSDM*, pp. 33–44, ACM.

[20] Berkhin, P. (2006), A survey of clustering data mining techniques, in *Grouping multidimensional data*, pp. 25–71, Springer.

[21] Bruza, P., R. McArthur, and S. Dennis (2000), Interactive internet search: keyword, directory and query reformulation mechanisms compared, in *SIGIR*, pp. 280–287, ACM.

[22] Carmel, D., H. Roitman, and N. Zwerdling (2009), Enhancing cluster labeling using wikipedia, in *SIGIR*, pp. 139–146, ACM.

[23] Chakrabarti, K., S. Chaudhuri, and S.-w. Hwang (2004), Automatic categorization of query results, in *SIGMOD*, pp. 755–766, ACM.

[24] Chaudhuri, S., and U. Dayal (1997), An overview of data warehousing and olap technology, *ACM Sigmod record*, *26*(1), 65–74.

[25] Chaudhuri, S., G. Das, V. Hristidis, and G. Weikum (2004), Probabilistic ranking of database query results, *VLDB*.

[26] Chen, K., and L. Liu (2004), Clustermap: Labeling clusters in large datasets via visualization, in *CIKM*, pp. 285–293, ACM.

[27] Chen, Z., and T. Li (2007), Addressing diverse user preferences in sql-query-result navigation, in *SIGMOD*, pp. 641–652, ACM.

[28] Chu, E., A. Baid, X. Chai, A. Doan, and J. Naughton (2009), Combining keyword search and forms for ad hoc querying of databases, in *SIGMOD*, pp. 349–360, ACM.

[29] Cutting, D. R., D. R. Karger, J. O. Pedersen, and J. W. Tukey (1992), Scatter/gather: A cluster-based approach to browsing large document collections, in *SIGIR*, pp. 318–329, ACM.

[30] Dakka, W., P. Ipeirotis, and K. Wood (2005), Automatic construction of multifaceted browsing interfaces, *CIKM*.

[31] Dakka, W., P. G. Ipeirotis, and K. R. Wood (2007), Faceted browsing over large databases of text-annotated objects, in *ICDE*, pp. 1489–1490, IEEE.

[32] Das, G., V. Hristidis, N. Kapoor, and S. Sudarshan (2006), Ordering the attributes of query results, in *SIGMOD*, pp. 395–406, ACM.

[33] Dash, D., J. Rao, N. Megiddo, A. Ailamaki, and G. Lohman (2008), Dynamic faceted search for discovery-driven analysis, in *CIKM*, pp. 3–12, ACM.

[34] Dougherty, J., R. Kohavi, M. Sahami, et al. (1995), Supervised and unsupervised discretization of continuous features, in *ICML*, pp. 194–202.

[35] English, J., M. Hearst, R. Sinha, K. Swearingen, and K. Yee (2002), Hierarchical faceted metadata in site search interfaces, *CHI*.

[36] Fagan, J. C. (2013), Usability studies of faceted browsing: A literature review, *Information Technology and Libraries*, *29*(2), 58–66.

[37] Fan, W., F. Geerts, J. Li, and M. Xiong (2011), Discovering conditional functional dependencies, *Knowledge and Data Engineering, IEEE Transactions on*, *23*(5), 683–698.

[38] Frank, A., and A. Asuncion (2010), UCI Machine Learning Repository.

[39] Geng, L., and H. J. Hamilton (2006), Interestingness measures for data mining: A survey, *ACM Computing Surveys (CSUR)*, *38*(3), 9.

[40] Gollapudi, S., and A. Sharma (2009), An axiomatic approach for result diversification, *WWW*.

[41] Goodchild, A. (1995), An evaluation scheme for trader user interfaces, *IFIP*.

[42] Gray, J., S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh (1997), Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals, *Data Mining and Knowledge Discovery*, *1*(1), 29–53.

[43] Guha, S., R. Rastogi, and K. Shim (1998), Cure: an efficient clustering algorithm for large databases, in *ACM SIGMOD Record*, vol. 27, pp. 73–84, ACM.

[44] Guha, S., R. Rastogi, and K. Shim (1999), Rock: A robust clustering algorithm for categorical attributes, in *ICDE*, pp. 512–521, IEEE.

[45] Guyon, I., and A. Elisseeff (2003), An introduction to variable and feature selection, *JMLR*, *3*, 1157–1182.

[46] Hall, M., E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten (2009), The weka data mining software: an update, *ACM SIGKDD Explorations Newsletter*, *11*(1), 10–18.

[47] Han, J., and M. Kamber (2006), *Data Mining: Concepts and Techniques*, Morgan Kaufmann.

[48] Hanrahan, P. (2006), Vizql: a language for query, analysis and visualization, in *SIGMOD*, pp. 721–721, ACM.

[49] Hearst, M. (2009), *Search user interfaces*, Cambridge University Press.

[50] Hearst, M. A. (2008), Uis for faceted navigation: Recent advances and remaining open problems, Citeseer.

[51] Hearst, M. A., P. Smalley, and C. Chandler (2006), Faceted metadata for information architecture and search, *CHI Tutorial*.

[52] Heckerman, D. (2008), *A tutorial on learning with Bayesian networks*, Springer.

[53] Herlocker, J. L., J. A. Konstan, L. G. Terveen, and J. T. Riedl (2004), Evaluating collaborative filtering recommender systems, *ACM Transactions on Information Systems (TOIS)*, *22*(1), 5–53.

[54] Hristidis, V., and Y. Papakonstantinou (2002), Discover: Keyword search in relational databases, in *Proceedings of the 28th international conference on Very Large Data Bases*, pp. 670–681, VLDB Endowment.

[55] Hristidis, V., L. Gravano, and Y. Papakonstantinou (2003), Efficient ir-style keyword search over relational databases, in *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pp. 850–861, VLDB Endowment.

[56] Huhtala, Y., J. Kärkkäinen, P. Porkka, and H. Toivonen (1999), Tane: An efficient algorithm for discovering functional and approximate dependencies, *The Computer Journal*, *42*(2), 100–111.

[57] Igarashi, T., and K. Hinckley (2000), Speed-dependent automatic zooming for browsing large documents, *UIST*.

[58] Ilyas, I. F., V. Markl, P. Haas, P. Brown, and A. Aboulnaga (2004), Cords: automatic discovery of correlations and soft functional dependencies, in *SIGMOD*, pp. 647–658, ACM.

[59] Inselberg, A. (1985), The plane with parallel coordinates, *The Visual Computer*, *1*(2), 69–91.

[60] Inselberg, A., and B. Dimsdale (1991), Parallel coordinates, in *Human-Machine Interactive Systems*, pp. 199–233, Springer.

[61] Jagadish, H., A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu (2007), Making database systems usable, *SIGMOD*.

[62] Jain, A. K., M. N. Murty, and P. J. Flynn (1999), Data clustering: a review, *ACM computing surveys (CSUR)*, *31*(3), 264–323.

[63] Jayapandian, M., and H. Jagadish (2008), Expressive query specification through form customization, in *EDBT*, pp. 416–427, ACM.

[64] Jayapandian, M., and H. Jagadish (2009), Automating the design and construction of query forms, *TKDE*, *21*(10), 1389–1402.

[65] Jensen, F. V. (1996), *An introduction to Bayesian networks*, vol. 210.

[66] Johnson, B., and B. Shneiderman (1991), Tree-maps: A space-filling approach to the visualization of hierarchical information structures, in *Visualization, 1991. Visualization'91, Proceedings., IEEE Conference on*, pp. 284–291, IEEE.

[67] Kashyap, A., V. Hristidis, and M. Petropoulos (2010), Facetor: cost-driven exploration of faceted query results, in *CIKM*, pp. 719–728, ACM.

[68] Kashyap, A., V. Hristidis, M. Petropoulos, and S. Tavoulari (2011), Effective navigation of query results based on concept hierarchies, *TKDE*, *23*(4), 540–553.

[69] Kaufman, L., P. Rousseeuw, and E. Corporation (1990), *Finding groups in data: an introduction to cluster analysis*, John Wiley.

[70] Keogh, E., X. Xi, L. Wei, and C. A. Ratanamahatana (2006), The UCR Time Series Homepage.

[71] Khoussainova, N., Y. Kwon, M. Balazinska, and D. Suciu (2010), Snipsuggest: context-aware autocompletion for sql, *VLDB*, *4*(1), 22–33.

[72] Kononenko, I. (1994), Estimating attributes: analysis and extensions of relief, in *ECML*, pp. 171–182, Springer.

[73] Kosara, R., F. Bendix, and H. Hauser (2006), Parallel sets: Interactive exploration and visual analysis of categorical data, *Visualization and Computer Graphics, IEEE Transactions on*, *12*(4), 558–568.

[74] Kotsiantis, S., and D. Kanellopoulos (2006), Discretization techniques: A recent survey.

[75] Koutrika, G., Z. Zadeh, and H. Garcia-Molina (2009), Data clouds: summarizing keyword search results over structured data, *EDBT*.

[76] Kuo, B., T. Hentrich, B. Good, et al. (2007), Tag clouds for summarizing web search results, *WWW*.

[77] Li, C., M. Wang, L. Lim, H. Wang, and K. Chang (2007), Supporting ranking and clustering as generalized order-by and group-by, *SIGMOD*.

[78] Li, C., N. Yan, S. B. Roy, L. Lisham, and G. Das (2010), Facetedpedia: dynamic generation of query-dependent faceted interfaces for wikipedia, in *WWW*, pp. 651–660, ACM.

[79] Li, G., S. Ji, C. Li, and J. Feng (2009), Efficient type-ahead search on relational data: a tastier approach, in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pp. 695–706, ACM.

[80] Liu, B., and H. Jagadish (2009), A spreadsheet algebra for a direct data manipulation query interface, in *ICDE*, pp. 417–428, IEEE.

[81] Liu, B., and H. Jagadish (2009), Using trees to depict a forest, *VLDB*.

[82] Liu, H., and L. Yu (2005), Toward integrating feature selection algorithms for classification and clustering, *TKDE*, *17*(4), 491–502.

[83] Liu, H., H. Motoda, and L. Yu (2004), A selective sampling approach to active feature selection, *Artificial Intelligence*, *159*(1), 49–74.

[84] Long, X., and T. Suel (2003), Optimized query execution in large search engines with global page ordering, in *VLDB*, pp. 129–140, VLDB Endowment.

[85] Mackinlay, J. D., P. Hanrahan, and C. Stolte (2007), Show me: Automatic presentation for visual analysis, *Visualization and Computer Graphics, IEEE Transactions on*, *13*(6), 1137–1144.

[86] MacQueen, J., et al. (1967), Some methods for classification of multivariate observations, *BSMSP*.

[87] Mannila, H., and K.-J. Räihä (1994), Algorithms for inferring functional dependencies from relations, *Data & Knowledge Engineering*, *12*(1), 83–99.

[88] Manning, C. D., P. Raghavan, and H. Schütze (2008), *Introduction to information retrieval*, vol. 1, Cambridge University Press Cambridge.

[89] Mei, Q., D. Zhou, and K. Church (2008), Query suggestion using hitting time, in *CIKM*, pp. 469–478, ACM.

[90] Mohan, L., and R. L. Kashyap (1993), A visual query language for graphical interaction with schema-intensive databases, *TKDE*, *5*(5), 843–858.

[91] Murray, N., N. Paton, and C. Goble (1998), Kaleidoquery: a visual query language for object databases, in *AVI*, pp. 247–257, ACM.

[92] Nandi, A., and H. Jagadish (2007), Assisted querying using instant-response interfaces, in *SIGMOD*, pp. 1156–1158, ACM.

[93] Ng, R., and J. Han (2002), A method for clustering objects for spatial data mining, *TKDE*.

[94] Nudelman, G. (2011), *Designing search: UX strategies for ecommerce success*, Wiley.

[95] Park, D. H., H. K. Kim, I. Y. Choi, and J. K. Kim (2012), A literature review and classification of recommender systems research, *Expert Systems with Applications*, *39*(11), 10,059–10,072.

[96] Qin, L., J. X. Yu, and L. Chang (2012), Diversifying top-k results, *Proceedings of the VLDB Endowment*, *5*(11), 1124–1135.

[97] Ricci, F., L. Rokach, and B. Shapira (2011), Introduction to recommender systems handbook, in *Recommender Systems Handbook*, pp. 1–35, Springer.

[98] Roy, S. B., H. Wang, U. Nambiar, G. Das, and M. Mohania (2009), Dynacet: Building dynamic faceted search systems over databases, in *ICDE*, pp. 1463–1466, IEEE.

[99] Sarawagi, S. (2000), User-adaptive exploration of multidimensional data., in *VLDB*, pp. 307–316.

[100] Sarawagi, S., R. Agrawal, and N. Megiddo (1998), *Discovery-driven exploration of OLAP data cubes*, Springer.

[101] Sarwar, B., G. Karypis, J. Konstan, and J. Riedl (2001), Item-based collaborative filtering recommendation algorithms, in *Proceedings of the 10th international conference on World Wide Web*, pp. 285–295, ACM.

[102] Shneiderman, B. (1992), Tree visualization with tree-maps: 2-d space-filling approach, *ACM Transactions on graphics (TOG)*, *11*(1), 92–99.

[103] Simitsis, A., G. Koutrika, and Y. Ioannidis (2008), Précis: from unstructured keywords as queries to structured databases as answers, *The VLDB Journal*, *17*(1), 117–149.

[104] Singh, M., A. Nandi, and H. Jagadish (2012), Skimmer: rapid scrolling of relational query results, in *SIGMOD*, pp. 181–192, ACM.

[105] Stoica, E., M. Hearst, and M. Richardson (2007), Automating creation of hierarchical faceted metadata structures, *NAACL HLT*.

[106] Tunkelang, D. (2009), Faceted search, *Synthesis lectures on information concepts, retrieval, and services*, *1*(1), 1–80.

[107] White, R. W., and R. A. Roth (2009), Exploratory search: Beyond the query-response paradigm, *Synthesis Lectures on Information Concepts, Retrieval, and Services*, *1*(1), 1–98.

[108] Wilson, M. L., R. W. White, et al. (2009), Evaluating advanced search interfaces using established information-seeking models, *Journal of the American Society for Information Science and Technology*, *60*(7), 1407–1422.

[109] Winter, B. (2001), A very basic tutorial for performing linear mixed effects analyses.

[110] Wu, P., Y. Sismanis, and B. Reinwald (2007), Towards keyword-driven analytical processing, in *SIGMOD*, pp. 617–628, ACM.

[111] Wu, T., X. Li, D. Xin, J. Han, J. Lee, and R. Redder (2007), DataScope: viewing database contents in Google Maps' way, *VLDB*.

[112] Yee, K.-P., K. Swearingen, K. Li, and M. Hearst (2003), Faceted metadata for image search and browsing, in *SIGCHI*, pp. 401–408, ACM.

[113] Yu, C., L. Lakshmanan, and S. Amer-Yahia (2009), It takes variety to make a world: diversification in recommender systems, *EDBT*.

[114] Zhang, T., R. Ramakrishnan, and M. Livny (1996), Birch: an efficient data clustering method for very large databases, in *ACM SIGMOD Record*, vol. 25, pp. 103–114, ACM.

[115] Zhang, Z., and O. Nasraoui (2006), Mining search engine query logs for query recommendation, in *WWW*, pp. 1039–1040, ACM.

[116] Zloof, M. M. (1977), Query-by-example: A data base language, *IBM systems Journal*, *16*(4), 324–343.

[117] Zloof, M. M. (1982), Office-by-example: A business language that unifies data and word processing and electronic mail, *IBM Systems Journal*, *21*(3), 272–304.