DIRECT MANIPULATION QUERYING OF DATABASE SYSTEMS

by

Bin Liu

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
2011

Doctoral Committee:

Professor Hosagrahar V. Jagadish
Assistant Professor Michael John Cafarella
Assistant Professor Kristen R. Lefevre
Assistant Professor Mark W. Newman

DEDICATION

To my parents and brother.

ACKNOWLEDGEMENTS

I would like to sincerely thank my advisor, Prof. H. V. Jagadish, for being an incredibly wonderful advisor. Without him, this thesis would not have been remotely possible. His wisdom, patience, and encouragement has and will continue to benefit me for the rest of my life. When I needed help in career or life, he has always been firmly standing behind me. I can continue indefinitely on how thoughtful and amazing he is through everyday details, which I am not going to do due to limited time and space. I sincerely appreciate his genuine care and generous help in all aspects of my life.

I would also like to thank my collaborators at IBM Almaden Research Center. Laura Chitacariu and Frederick Reiss, thanks to both of you for the fruitful collaboration. Thank you for introducing me to top-notch industrial research and development. Ru Fang, Bin He, Hui-I Hsiao, and C Mohan, thank you for giving me a happy and rewarding summer.

I would also like to thank my thesis committee members, Prof. Michael Cafarella, Prof. Kristen Lefevre, and Prof. Mark Newman. Thank you for spending your valuable time on my thesis. I appreciate your insightful comments and guidance along the way.

The fantastic people of the database group have been a great part of my life at Michigan. Beside the faculty I mentioned above, Professor Jignesh Patel, who overlapped with me for two years at Michigan, has always been very kind and inspirational to me. I would like to thank all students in Professor Jagadish's group

for the support all these years. Ever since I joined Michigan, senior students in the group, including Adriane Chapman, Magesh Jayapandian, Yunyao Li, Stelios Paparizos, Nuwee Wiwatwattana, and Cong Yu, have been a never-ending source of life and professional help, even after their graduation. Their selfless support has constantly been a great boost for me to forge ahead. I would also like to thank alumni in Professor Patel's group, including Yun Chen, Nate Derbinsky, You Jung Kim, Willis Lang, Michael Morse, Sandeep Tata, and Yuanyuan Tian, for their friendship and comradeship. Current members of the group, including Zhe Chen, Daniel Fabbri, Fernando Farfan, Lujun Fang, Arnab Nandi, Li Qian, Anna Shaverdian, Manish Singh, Glenn Tarcea, and Jing Zhang, I thank you for being there when the deadlines were looming and when cakes were cut.

My friends at Michigan and all over the world, you are a source of inspiration and delight. I want to give special thanks to Joseph Xu, my great friend and the best roommate, for being there in the past five years. Xu Chen, Xin Hu, Xiaolin Shi, Feifei Wang, Ying Zhang have also been wonderful friends to me. Younger generation, including Junxian Huang, Feng Qian, Zhaoguang Wang, Yudong Gao, Fangjian Jin, Caoxie Zhang, Xinyu Zhang, Xiaoen Ju, Yunjing Xu, and Jie Yu, it has been wonderful to be with you at Michigan. Outside of Michigan, I want to thank my friends Edward Au and Jing Yan for always believing in and supporting me.

I also want to thank my friends in the database community. It is impossible to list all the names so I will not attempt to do so. Thank you for being both kind and inspirational all these years. I look forward to continue the interaction and collaboration with you.

Last but not least, I want to thank my parents and my brother, to whom I have owned too much. I thank you for your unconditional love and support all these years. You are the best family one could hope for.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

CHAPTER I

INTRODUCTION

Databases are tremendously powerful, but their poor usability has been well-documented [50]. From an information-seeker's perspective, database systems today pose insurmountable barriers for use by non-experts. Current database interfaces usually consist of a structured query language input and a tabular list of output tuples. How does this type of interface perform in information seeking? A standard model [95] for information seeking contains four stages: query formulation, action (running the query), review of results, and query refinement. A database user is required to learn a programming language in order to formulate a query; when there are a large number of results, which happens frequently with today's overload of information, it is difficult for user to make sense of the output when reviewing the results; when the user finds some undesirable output, she is offered little help to refine the query. Through out the information seeking process, today's database user interface provides help to users much less than desired. This also explains why database software remains largely unused by average computer users.

While the database community has always focused on functionality and performance and thus left a gap between database and users, our neighbors in the human computer interaction (HCI) community have developed one of the most popular and user-friendly paradigm – *Direct Manipulation* [93, 94, 92]. Examples of a Direct Manipulation interface include the tremendously popular Microsoft Powerpoint and Adobe Photoshop, where users manipulate objects of interest

through mouse-clicks and see what they get immediately. Direct Manipulation has three principles:

1. Continuous representation of the object of interest.
2. Physical actions or labeled button presses instead of complex syntax.
3. Rapid incremental reversible operations whose impact on the object of interest is immediately visible.

The success of Direct Manipulation inspired us to bring this paradigm to database querying. We seek to apply Direct Manipulation principles in each stage of the information seeking process, which is the central goal of the thesis.

The rest of the chapter is organized as follows. In Sec. 1.1 we further motivate the thesis by examining the pain points in the information seeking process. In Sec. 1.2 we briefly present our solutions and contributions that relieve users from those pain points.

1.1 Motivation

We now examine the stages of information seeking and evaluate possible places to improve. Recall the four stages of information seeking [95]: query formulation, query execution, review of results, and query refinement. Query execution rarely involves user action, unless the query is a long-running one. While long-running queries are very important for industrial tasks, average users are more likely to issue short-running queries where results are obtained in a matter of seconds. Thus we focus on the remaining three steps of the process.

- Query formulation. Currently users are required to write a SQL query. This contradicts the second principle of Direct Manipulation, which dictates that users use physical actions rather than complex syntax. User inputs a query

to the database system, and hoping that the query is correct. Note that user has to write a query before seeing the actual data – the most the user can see is the schema of the database being queried. This is against the first principle of Direct Manipulation. Ideally, users will have some data to see even before a query is issued. If a query is complex, the user has to make sure that it is correct by repeatedly debugging – submitting it to the database and observe the output for both syntactical errors and undesirable data output. This is against the third principle of Direct Manipulation. It would be much easier for users to build a complex query with small manageable incremental steps. It is even more desirable to be able to reverse course if an error has occurred.

- Review of results. Users can review results either after the query is specified or after their intermediate steps, if the query is complex. The latter does not yet exist for current database systems. They typically display results either in the order they are produced or sorted by attributes specified by the user, either in the SQL query or through a graphical user interface. When the query is not selective enough, frequently there are a large number of results returned (can easily be thousands of tuples). A computer screen can typically display fewer than one hundred rows of results. Users have to manually flip through multiple sheets of results. For users to make sense of what is actually in the result set is a daunting task.

- Query refinement. If users find desirable and undesirable results in the output, and they want to refine the query such that undesirable output tuples are removed while desirable tuples are kept, they have to manually re-write the query. This manual process can be tedious and frustrating, especially when the query is long. When a query involves multiple steps, it is difficult to identify which step should be adjusted just from the output. Users need more help and

automation in this difficult task.

1.2 Contributions

In this thesis, we make the following contributions.

For the query formulation stage, our contribution is a spreadsheet algebra upon which a direct manipulation interface can be built upon [67]. We develop a spreadsheet algebra that is powerful (capable of expressing at least all single-block SQL queries) and can be intuitively implemented in a spreadsheet. Based on the algebra, we build a spreadsheet interface where users i) always see the data she is querying, ii) query a database using mouse-clicks, and iii) build a complex query through small steps and receive feedback for each step. User study shows that this interface is more usable than a popular graphical query builder. This contribution brings direct manipulation to the query specification process.

Our second contribution brings direct manipulation to the query result review stage [68, 69]. When a query has many results, the user can only be shown one page of results at a time. One popular approach is to rank results such that the "best" results appear first. However, standard database query results comprise a set of tuples, with no associated ranking. An alternative approach to the first page is to help users learn what is available in the whole result set and direct them to finding what they need. In this chapter, we demonstrate through a user study that a page comprising one representative from each of k clusters (generated through a k-medoid clustering) is superior to multiple alternative candidate methods for generating representatives of a data set. After seeing some results, users often refine query specifications based on returned results by manipulating the representatives. We propose a tree-based method for efficiently generating the representatives, and smoothly adapting them with query refinement. Experiments show that our algorithms outperform the state-of-the-art in both result quality and efficiency.

Our third contribution brings direct manipulation to the query refinement stage through a provenance-based automatic query refinement framework. It is terribly difficult to specify a perfect query at once without running it on the actual data. Often a query can be built incrementally through many intermediate steps or views. Users observe the output tuples and make necessary modifications to the query and hope the result will be correct. So far this has been a tedious manual process. We propose a new framework for automatic query modification based on result provenance. Users interact with the system through simple mouse-clicks: drag unwanted results into recycle bin and click on desirable results to indicate they like them. Our framework takes all labeled results, trace their provenance, and identify spots in the query where modification can be made to remove unwanted results while preserving as many as possible desired results. We suggest a ranked list of exact modifications that can be made, where ranking is based on performance in precision and recall. This part of the work was collaborated with IBM Research and it was conducted in the context of rule-based information extraction, but the theory and principles we used and developed also apply to refining database queries.

Our fourth contribution is to provide assistance to user querying through useful query suggestions. This part of the work takes into account all three stages of information seeking. Users often do not start querying with a precise picture of what she wants and what is available in the data. They often search while browsing. To take advantage of this scenario, we propose an interaction paradigm where users query a database through browsing sample tuples. She labels the tuples as desirable or not, and we take such labels to refine our suggestions. Our approach is based on data mining techniques applied to query logs, data, and schema of a database. This work completely relieves the user from writing SQL.

The rest of the thesis is organized as follows. We present the spreadsheet algebra in Chapter II, which is followed by query result review through representatives in

Chapter III. In Chapter IV we discuss the query refinement work (in progress), and we present assisted querying by browsing in V. We conclude the thesis with future work in Chapter VI.

CHAPTER II

QUERY SPECIFICATION: A SPREADSHEET ALGEBRA FOR A DIRECT MANIPULATION QUERY INTERFACE

2.1 Introduction

2.1.1 Motivation

Non-technical users find it challenging to specify queries against structured databases. This is true even with visual query builders, which provide a largely "point-and-click" means to develop query specifications. One possible reason for this is the separation of query specification from result presentation in most databases available today. The theory is that humans are good at manipulating things that they can "touch", but it requires substantially greater technical sophistication to be able to abstract the specification into a query that must be fully specified separate from the data being operated on before it can be executed.

To address this need, the term direct manipulation was coined by Shneiderman [93, 94, 92] to refer to systems that support: i) continuous representation of the object of interest, ii) physical actions or labeled button presses instead of complex syntax, and iii) rapid incremental reversible operations whose impact on the object of interest is immediately visible.

In the text editing context, systems with similar properties have been called *WYSIWYG* (What You See Is What You Get). In the database context, with a direct manipulation interface the user always has on hand some data set, currently

being analyzed or manipulated. A chunk of the data set is visible on the screen – all of it is not likely to fit, except for the smallest data sets. Initially, the current data set may be source data – say a relation in a database. After each manipulation is performed, the user has intermediate result data available. Eventually, the final results become available to the user. Modifications to the data set at hand are performed by "directly" specifying operators to be applied to it. To meet the second and third desiderata above, each such user-specified manipulation must be fairly simple, and the result of applying it reflected immediately by updating the current data set.

Spreadsheets (for example, Microsoft Excel and OpenOffice Calc) are popular means for analyzing data through direct manipulation. They are frequently used to analyze data extracted from database systems, particularly in the context of decision support. However, spreadsheets are not designed for querying databases (although it is possible to use them to load data from a database). Our objective is to create a spreadsheet-like interface to directly query and access relational databases through direct manipulation. To accomplish this we need to overcome several challenges, which we describe next, beginning with an example.

2.1.2 Conceptual Challenges

Sam, a new graduate student at the University of Michigan, is looking for a sedan in a used car database, which maintains common attributes of cars (e.g., year, model, mileage, price, condition). Some sample records are shown in Table 2.1.

Query Division Challenge Sam is interested in late model cars (2005 or later) in good or excellent condition, and he would like the results grouped by Model and ordered by Price. This is a simple query to specify in SQL. But, on a spreadsheet, just by pointing and clicking, there is no straightforward way to state all these requirements at once. Instead, Sam has to break down his need into parts, and

Table 2.1: Sample Used Car Database

ID	Model	Price	Year	Mileage	Condition
304	Jetta	$14,500	2005	76,000	Good
872	Jetta	$15,000	2005	50,000	Excellent
901	Jetta	$16,000	2005	40,000	Excellent
423	Jetta	$17,000	2006	42,000	Good
723	Jetta	$17,500	2006	39,000	Excellent
725	Jetta	$18,000	2006	30,000	Excellent
132	Civic	$13,500	2005	86,000	Good
879	Civic	$15,000	2006	68,000	Good
322	Civic	$16,000	2006	73,000	Good

specify one part at a time (e.g. "select late model cars"). Upon asking for this, Sam will immediately see all late model cars, irrespective of condition, and not grouped and ordered as he would like, since these requests have not yet been made. This division of a single query into pieces is not in itself challenging, but it is the root of other difficulties we discuss below.

Grouping Challenge Whenever data is displayed, issues of ordering and grouping must be considered. In relational systems, grouping is "hidden" in that it is paired with aggregation, and ordering is frequently outside the main algebraic query specification, treated as a post-operation for display purposes. Due to query division, we have to worry even about "intermediate results" in the case of a spreadsheet. Since these are shown to the user, we have to be concerned with ordering and grouping even for the results of each intermediate operation. This takes us from the realm of sets (or multi-sets) in the case of relational systems to collections that support grouping and ordering.

Aggregation Challenge Aggregate computation is a common operation invoked during data analysis. However, computation of a relational aggregate query results in the definition of a new relation which is typically not union compatible with the original relation, making it hard to store the data and its aggregates all on a single "spreadsheet." For example, suppose Sam wants to see cars whose price is lower than the average for that Model and Year. It is not immediately obvious

how to accomplish this through direct manipulation starting from a display similar to Table 2.1. (A SQL query to accomplish this would involve nesting, and a join between two copies of the base table).

Query Modification Challenge In interactive data analysis, users may often find the need to make small modifications to previously issued queries. For example, upon finding too many results in model"Jetta" in year 2005 and price less than $18k (and meeting many other conditions he specified), Sam may feel optimistic and desire to change the year to 2006. In a direct manipulation interface, complex queries are developed one operator at a time, and a complete query expression is never explicitly articulated. We would like to provide the user with a facility that is the equivalent of making a small modification to a large query expression and re-submitting it for evaluation.

Operator Ordering Challenge A complex query specification may involve the specification of a sequence of multiple operators. A non-technical user may find it strange if applying the same operators in a different sequence produces different results, particularly since there is no query expression showing explicit operator ordering or parentheses. For example, suppose Sam first computed the average price of all cars of each model, and then realizes he is only interested in cars in 2005, he should be able to just specify that condition and the average price should correct itself immediately. In effect, it should be the same as if the two operations were specified with the selection first and the average afterward. If such automatic recomputation is not feasible, there should at least be a suitable notice given to the user so that unintended wrong results are not quietly computed.

2.1.3 Contributions

The first contribution of this chapter is the development of a spreadsheet algebra introduced in Sec. 2.3. We developed precise semantics of a spreadsheet model

and all algebraic operators that manipulate data in the spreadsheet. The unit of manipulation in the algebra is a recursively grouped ordered multi-set of tuples to address the grouping challenge. Operators in this algebra have been carefully designed so that all unary data manipulation operators commute to address the operator ordering challenge.

Spreadsheets support aggregate computation, and permit storage of aggregates in cells with "computed attributes" – the value of such a cell is defined by means of an expression in terms of the values of the cells being aggregated. This is a notion that spreadsheet users are used to and can be expected to exploit readily. Exploiting this notion, aggregation is defined not as an operator directly, but as the creation of a corresponding computed attribute. This addresses the aggregation challenge.

All single block SQL queries with selection, projection, join, grouping, aggregation, group selection (the HAVING clause), and ordering, can be expressed in this spreadsheet algebra. These, and other (e.g., operator commutativity), properties of the spreadsheet algebra are explored in Sec. 2.4.

The query modification challenge is addressed in Sec 2.5 through a novel proposal for re-writing query history by exploiting the *query state* retained in the spreadsheet interface, and exploiting spreadsheet algebra properties discussed in Sec. 2.4.

The third contribution of this chapter is a spreadsheet interface to an RDBMS that implements the spreadsheet algebra. The core of this interface is the design of specific implementations for spreadsheet algebra operators. The design is presented in Sec. 2.6.

Our final major contribution is an empirical assessment of the spreadsheet interface. We built a prototype, SheetMusiq, with ideas in this chapter. Experiments with human subjects not familiar with SQL show that our implementation is easier to work with than a representative popular visual query builder. Specifically, for a wide assortment of queries given to them in English, users were able to express

the query and obtain results faster with the spreadsheet presentation than with the visual query builder. Details are given in Sec. 2.7.

The rest of the chapter is organized as follows. We first define the spreadsheet model in Sec. 2.2, and then present the spreadsheet algebra for this model in Sec. 2.3. In Sec. 2.4, we study the expressive power of the algebra and the commutativity among unary operators, which lays a solid foundation for query modification in Sec. 2.5. We then present a design (Sec. 2.6) and user evaluation (Sec. 2.7) of user interface built upon the algebra, before concluding the chapter with related work.

2.2 The Spreadsheet Data Model

A *data model* (as defined in [104]) contains a notation for describing data, and a set of operations used to manipulate that data. For example, in relational data model, we have "relation" and relational algebra operators. We seek to define the notation and structure of the data model in this section, and will consider specific operators in Sec. 2.3.

2.2.1 Intuition

The fact that the spreadsheet is used to present data to users for manipulation poses unique requirements not met by the relational data model. Key amongst these is that the data must be grouped and/or ordered after every operator. Furthermore, the user should not have to re-specify the grouping and ordering with each small operation performed. As such, grouping and ordering must be retained through operators, to the extent possible. For example, a selection condition applied to the data should not change its grouping or ordering.

The basic unit of the spreadsheet algebra is a *spreadsheet*. Unlike a relation, which is an unordered set of tuples, a spreadsheet must support both grouping and ordering. A spreadsheet that has no grouping (or ordering) specified is said to be

grouped (resp. ordered) by NULL.

Grouping is not in textbook relational algebra [83]. In SQL, it is always associated with aggregation. The aggregation and grouping operator, if treated as a single algebraic operator, is quite heavy weight. It can involve a GROUP BY clause, a HAVING clause, multiple aggregate functions, and constraints between the list of attributes in the SELECT clause and the GROUP BY clause. A single operator that does all this is not within the spirit of direct manipulation. Rather, we seek to define a separate operator for each logical component of this mega-operator – one for grouping, one for each aggregate, and one for each group selection predicate (in the HAVING clause). But this new grouping operator is not closed over relations (grouping is lost in a set), which creates a problem.

In relational algebra with grouping and aggregation, closure is achieved by insisting that attributes not in the grouping list be projected out (after aggregation and group qualification have been performed, if specified), leaving precisely one tuple in place of each group. In the spreadsheet model, we achieve closure by defining the algebra not over relations restricted to sets of tuples but over spreadsheets that are *recursively grouped set of tuples*. Simply put, a recursively grouped set of tuples is a set of tuples with grouping information. If no grouping is initially specified, the spreadsheet is grouped by NULL. When grouping contains multiple *levels* (e.g., group the cars first by Model, then by Year), a recursive grouping is formed. Each level of group is a relational group. We number the levels of group from the outermost – the root (first) is the spreadsheet itself, cars of the same Model form the second level, and the highest (or *finest*) level groups are cars with same Model and Year. The *basis* of a level of grouping is the set of attributes whose values are the same for all tuples inside any group at this level but not in the parent level. Following the example, we say the "basis" for each level of grouping, from outermost, are {NULL}, {Model}, and {Model, Year}. We will often find it convenient to speak

of the *relative grouping basis* as the difference between the basis for one level of grouping and level below it. Thus, the innermost level has a relative grouping basis of Year.

Order can be specified inside each level of group. In the finest level of group, tuples can be ordered by any attribute that is not in any grouping basis. For example, we can order cars with the same Model and Year by Price. For other level of groups, the ordering attribute is already specified by the grouping, and thus only "descending" or "ascending" is allowed. Specifically, the ordering attributes are those in the grouping basis of the immediate higher level but not this level. For example, for the second level groups (cars with the same Model but different year), cars are automatically grouped by Year.

Note that any recursive grouping can be emulated by a single ordering specification in that all tuples can be placed in the same order. To accomplish this, specify order by the lowest level group first, then the next level group, and so on, until finally the order within the highest group, with each order being the same as in the recursive grouping to be emulated. Mere ordering does not provide group identification, though, and leaves all tuples in a single set rather than in a set of (set of...) sets. We would then be unable to specify operators that compute any function of groups.

Whereas a spreadsheet could be used to represent data stored with various organizations, relational databases are prevalent today, and so using a spreadsheet to represent a relation is the natural thing to do. In this chapter, we restrict ourselves to the use of spreadsheets to represent relational data. Specifically, a single spreadsheet is used to represent a single relation. Each column in the relation corresponds to a column in the spreadsheet. In addition, the spreadsheet may have some *computed columns*. (We will describe the use and importance of these computed columns in Sec. 2.3).

2.2.2 Definition of the Spreadsheet Model

Based on previous discussion, we now formally define the spreadsheet data model. Throughout the chapter, we use subscripted lowercase letters to denote elements in sets or lists represented by their corresponding capital form. Superscript denotes the version of an object. For example, g_i^0 denotes the i-th element in G^0 ($1 \leq i \leq |G^0|$). If G^0 is changed, we create a new version, G^1, and its elements are now denoted as g_i^1.

Definition 1 (**Spreadsheet**). *A* spreadsheet *S is a multi-set of tuples specified by a quadruple (R, C, G, O), where*

1. *R is a reference to the relation it represents, known as the* base relation *of S.*
2. *C is a superset of columns in R,*
3. *G is a list for grouping specification. Element g_i is a set of attributes that forms the basis of the i-th level of grouping (i starts from 1). $g_1 = \{NULL\}$.*
4. *O is a list for order specification, where o_i is for group level i. For $1 \leq i < |O|$, o_i takes a value of either "ASC" or "DESC", and ordering attributes are those in g_{i+1} but not in g_i. For $i = |O|$, o_i contains two ordered lists:* attributes *and* orders *("ASC" or "DESC"). Elements in the same position of the two lists form an ordering, and* attributes *may contain any attribute not in G.*

A spreadsheet presents only its base relation R, and multiple spreadsheets can present the same relation. Given R, it is straightforward to construct an initial spreadsheet by directly inheriting the columns, and leaving grouping and ordering to be empty.

Definition 2 (**Base Spreadsheet**). *For relation R, its* base spreadsheet *is $S^0(R, C^0, G^0, O^0)$, where C^0 is the set of columns in R, and G^0 and O^0 are both empty lists.*

In this chapter, tuples in R can be changed anytime, and the spreadsheet always retrieves the latest data. However, we require that the columns of the base relation R remain unchanged in the *life–time* of a spreadsheet S, which starts from the creation of S_0 to the destruction of the latest version of S. If R does change, we create a new base spreadsheet.

2.3 The Spreadsheet Algebra

In this section, we present operators on spreadsheets and define their semantics.

2.3.1 Basic Data Organization Operators

Ordering and grouping are crucial operators in a spreadsheet even though they do not change the actual "content". We begin by studying these data organization operators first.

Grouping (τ). The grouping operator τ takes as parameters *grouping-basis* (a set of attributes) and *order* ("DESC" or "ASC"). It groups tuples with equal values in all elements of the *grouping-basis* and order the groups in the *order* specified. A new level of grouping is created when and only when *grouping-basis* contains a superset of attributes of any existing grouping basis. Tuples in previously finest level of groups are further grouped according to attributes newly specified. Those new groups (not the tuples in each group) are ordered according to *order*. Denote as L the ordering attributes at the finest level before applying τ. Tuples inside each new group are grouped by a new ordered list o_L, which contains all elements that are in L but not in *grouping-basis*. We use the subtraction sign to denote this "list subtraction" operation: $o_L = L - \mathit{grouping\text{-}basis}$. Note that L is unchanged after the subtraction.

We now give the formal definition for grouping. We assume we start from version j of a spreadsheet – $S^j(R^j, C^j, G^j, O^j)$. As in Section 2.2, we use subscripted lowercase letter to denote elements in a set or list represented by its corresponding capital form (e.g., o_1^j means the first element in O^j). Following standard convention, we use $o_1^j.\mathit{attributes}$ and $o_1^j.\mathit{orders}$ to access the components of ordering specification o_1^j.

Definition 3 (Grouping). $\tau_{\mathit{grouping\text{-}basis},\mathit{order}}(S^j(R^j, C^j, G^j, O^j)) = S^{j+1}(R^j, C^j, G^{j+1}, O^{j+1})$, *where*

1. $g_i^{j+1} = g_i^j$ *for* $1 \le i \le |G^j|$, *and* $g_i^{j+1} = \mathit{grouping\text{-}basis}$ *for* $i = |G^j| + 1$;

Table 2.2: Car Database After Grouping by Condition

ID	Model	Price	Year	Mileage	Condition
872	Jetta	$15,000	2005	50,000	Excellent
901	Jetta	$16,000	2005	40,000	Excellent
304	Jetta	$14,500	2005	76,000	Good
723	Jetta	$17,500	2006	39,000	Excellent
725	Jetta	$18,000	2006	30,000	Excellent
423	Jetta	$17,000	2006	42,000	Good
132	Civic	$13,500	2005	86,000	Good
879	Civic	$15,000	2006	68,000	Good
322	Civic	$16,000	2006	73,000	Good

2. $o_i^{j+1} = o_i^j$ *for* $1 \leq i \leq |O^j|$*; for* $i = |O^j| + 1$, $o_i^{j+1}.attributes = o_i^j.attributes - grouping\text{-}basis$, *and* $o_i^{j+1}.orders$ *is the corresponding sub-list of* $o_i^j.orders$.

Example 2.3.1. *We start from spreadsheet* S^j *in Table 2.1, where cars are grouped by Model (DESC) and then Year (ASC), and ordered in the finest groups by Price (ASC). After operation* $\tau_{\{Year,Model,Condition\},ASC}(S^j)$*, we create the fourth level of grouping with relative grouping basis of Condition. The result table is shown in Table 2.2.*

Ordering (λ). It takes as parameters (*attribute, order, l*) and orders tuples in the l-th level groups by *attribute* (ASC or DESC, as specified in *order*). Denote the number of grouping levels as n. As explained in Sec. 2.2, ordering attributes in group levels other than the n-th are dictated by the grouping. If an ordering attribute is specified in the i-th level groups ($i < n$), and it is different from the current ordering attribute imposed by grouping, all groupings from level-$(i+1)$ and beyond are destroyed (and so are any computed values, such as aggregations, based on these groupings).

While destruction of groups creates no algebraic or semantic difficulty, it can be confusing for a user. As such, order specifications that destroy grouping is permitted in our implementation only if there are no aggregates present on the grouping that will be lost (the aggregates have to be projected out before such operations are allowed).

Definition 4 (Ordering). $\lambda_{attribute,order,l}(S^j(R^j, C^j, G^j, O^j)) = S^{j+1}(R^j, C^j, G^j, O^{j+1})$, *and*

1. *If* $1 \leq l < |G^j|$ *and attribute* $\notin (g^j_{l+1} - g^j_l)$*: for* $i < l$, $g^{j+1}_i = g^j_i$, $o^{j+1}_i = o^j_i$*; for* $i = l$, $g^{j+1}_i = g^j_i$, $o^{j+1}_i = order$*; for* $i > l$, $g^{j+1}_i = NULL$, $o^{j+1}_i = NULL$.
2. *If* $1 \leq l < |G^j|$ *and attribute* $\in (g^j_{l+1} - g^j_l)$*:* $G^{j+1} = G^j$, $o^{j+1}_l = order$, *and* $o^{j+1}_i = o^j_i$ *for* $i \neq l$.
3. *If* $l = |G^j|$*, then* $G^{j+1} = G^j$*. And, if* $\exists i$ *such that* attribute $\in g^j_i$, $O^{j+1} = O^j$*. Let* $|G^j| = k$*. If attribute* $\in o^j_k.attributes$*, change corresponding entry in* $o^j_k.attributes$ *to* order *to obtain* o^{j+1}_k*; else, add* attribute *and* order *to the end of respective list in* o^j_k *to obtain* o^{j+1}_k.

Example 2.3.2. *We start from spreadsheet* S^j *in Table 2.1 as in Example 2.3.1. If we apply* $\lambda_{Mileage,ASC,3}(S^j)$*, we further order cars by Mileage in the finest level of groups. If we apply* $\lambda_{Mileage,ASC,2}(S^j)$*, in level-2 groups, we destroy the grouping at level 3 (relative grouping basis of Year).*

2.3.2 Basic Data Manipulation Operators

A desirable property of a query language is relational completeness [15, 83], meaning it can express all queries expressible with relational algebra. Among all relational operators, selection (σ), projection (π), Cartesian product ($\times$), set union ($\cup$), and set difference (-) form a *complete set*, since any other operators can be expressed as a sequence of operators from this set [104] (theorem 2.1). For this reason, we adopt this *complete set* of relational operators as the foundation of our direct manipulation interface. We also use the same set of abbreviations as relational algebra. For their relational counterparts, we subscript the abbreviations with "r" (for example, $\times_r$ for relational product). Since operators are confined within the spreadsheet model, there are crucial differences between our operators and their relational counterparts, as we will describe shortly. Other important operators include join, aggregation, formula computation, and duplicate elimination. The interface design of all operators will be presented in Section 2.6. Before detailing the operators, however, we introduce two related concepts: *stored spreadsheet* and *computed column*.

The spreadsheet is designed such that it should be sufficient to present only one spreadsheet to the user at any time. Not having to manipulate multiple spreadsheets simultaneously makes the spreadsheet model more user-friendly. This does create

a problem for binary operators like Cartesian product, where two spreadsheets are involved. This is where *stored spreadsheet* comes in. We allow a spreadsheet to be stored and later re-loaded, regardless of the number of operations it went through. Binary operations can now be performed between a stored spreadsheet and the *current spreadsheet* (the one currently presented to the user).

Another design consideration is to provide users the power of analytical processing in the spreadsheet. General spreadsheet applications like Microsoft Excel are essential to modern business, partly because users can define formulas on the spreadsheet and do calculations and analysis [108]. We define a *computed column* for the result column of computing a formula over the existing spreadsheet. The essential property of computed columns is automatic updates. Once a user has defined such a column, the user expects it to reflect the value correctly even as the database or spreadsheet is updated.

We now introduce formal definition for each operator.

Selection (σ). Let F be a condition that may involve:

1. Atomic predicates in the form of $A\ OP\ B$, where A and B can be column names or constants (but not both being constants), with optional arithmetic or string operators (for example, $2 \times a$ or $a + b$), and OP is any comparison operator (e.g., ">"),

2. Expressions built from connecting items in (1) with "AND", "OR", or "NOT".

Definition 5 (Selection). $\delta_F(S^j(R^j, C^j, G^j, O^j)) = S^{j+1}(R^{j+1}, C^j, G^j, O^j)$, *where for all tuples* t *in* R^j, $t \in R^{j+1}$ *if and only if* F *applied to* t *is true.*

Projection (π). Projection takes a single parameter, *column*, for the column to be removed from the spreadsheet.

Notice the differences between π and its relational counterpart π_r: 1) π only removes one column at a time, while π_r can remove multiple columns in one go; 2) Therefore it is more natural to specify as parameter to π the column to be removed,

whereas the parameter to π_r is the set of columns to be retained. No duplicate elimination is performed after projection, since a spreadsheet is defined as a recursive *multi-set*. Grouping and ordering is retained through projection.

Definition 6 (**Projection**). $\pi_{column}(S^j(R^j, C^j, G^j, O^j))$
$= S^{j+1}(R^j, C^{j+1}, G^j, O^j)$, *where* $C^{j+1} = C^j - column$.

Cartesian Product ($\times$). It is possible to compute the Cartesian product of the current spreadsheet S^j with a stored spreadsheet $S^k_s(R^k_s, C^k_s, G^k_s, O^k_s)$ (we use subscript s to denote "stored spreadsheet"). S^k_s and S^j can present different base relations. To compute $S^k_s \times S^j$, we perform relational product on R^j and R^k_s and apply grouping and ordering of S^j on the result to maintain coherence of presentation. This means that product is not symmetric: $S^j \times S^k_s \neq S^k_s \times S^j$ even after reordering columns, since the grouping and ordering would be different. All computed columns are updated such that computation is based on the product.

Definition 7 (**Cartesian Product**). $S^j(R^j, C^j, G^j, O^j)) \times S^k_s(R^k_s, C^k_s, G^k_s, O^k_s)) =$
$S^{j+1}(R^j \times_r R^k_s, C^j \cup_r C^k_s, G^j, O^j)$.

Set operators. Like product, set union ($\cup$) and set difference (-) operate on two spreadsheets each time. Grouping and ordering of the current spreadsheet S^j are used for the result. Note that union, like product, is asymmetric, because of this notion of current spreadsheet versus the second stored spreadsheet. The two spreadsheets must be compatible (having the same set of columns, excluding computed attributes). No duplicate elimination is performed. The union (resp. difference) is computed using standard multi-set semantics, so that the union of a tuple and its duplicate are two identical tuples, and the difference $\{t, t\} - \{t\}$ is $\{t\}$. Also, computed attributes do not participate in set operators. Instead, computed attribute columns in the base spreadsheet are retained, and recomputed based on the new set membership.

Definition 8 (**Set Union**). $S^j(R^j, C^j, G^j, O^j)) \cup S^k_s(R^k_s,\ C^k_s,\ G^k_s, O^k_s)) =$
$S^{j+1}(R^j \cup_r R^k_s, C^j, G^j, O^j)$.

Definition 9 (Set Difference). $S^j(R^j, C^j, G^j, O^j)) -_r S^k_s(R^k_s,\ C^k_s,\ G^k_s, O^k_s)) = S^{j+1}(R^j -_r R^k_s, C^j, G^j, O^j)$.

Join ($\bowtie$). Assume $S^j(R^j, C^j, G^j, O^j)$ is the current spreadsheet, and $S^k_s(R^k_s, C^k_s, G^k_s, O^k_s)$ is a stored spreadsheet. We can join the two on any condition F that is supported by SQL. To compute the join, we perform relational join with F as the join condition on R^j and R^k_s and inherit grouping and ordering from S^j. The resultant spreadsheet is used as the current spreadsheet, and all computed columns are updated accordingly. Join can be emulated by product followed by selection.

Definition 10 (Join). $S^j(R^j, C^j, G^j, O^j)) \bowtie S^k_s(R^k_s,\ C^k_s,\ G^k_s, O^k_s)) = S^{j+1}(R^j \bowtie_{r,F} R^k_s, C^{j+1}, G^j, O^j)$, *where*

1. $\bowtie_{r,F}$ *means a relational join with condition* F,
2. C^{j+1} *contains all computed columns, as well as all columns in* $R^j \bowtie_{r,F} R^k_s$.

Aggregation (η). Any standard aggregation operator (*e.g., sum, avg*) can be computed on a selected column. The operator computes the aggregate over all attribute values in that column within a group, at any level. Thus η takes parameters f(function), c (column over which aggregation is computed), and l (level of groups). Recall that a spreadsheet is always at least grouped by NULL. So if grouping has not been applied, the aggregation is computed over all values in the column in the entire spreadsheet. In general, grouping may have been applied, recursively. In such a case, the aggregation is computed over a group at the specified level, not necessarily the innermost one. In particular, aggregates may still be computed over the entire spreadsheet, across all groups. Note that all operators apply to individual tuples and not to intermediate groups. Thus the result of COUNT is the number of tuples in the group being counted, and not the number of sub-groups in the group, even if sub-groups are present.

The next question is where to store and display the result of the aggregation. There is only one result value per aggregation group. We could store the aggregation

Table 2.3: Car Database – Average Price by Model and Year

ID	Model	Price	Year	Mileage	Avg_Price
304	Jetta	$14,500	2005	76,000	$15,167
872	Jetta	$15,000	2005	50,000	$15,167
901	Jetta	$16,000	2005	40,000	$15,167
423	Jetta	$17,000	2006	42,000	$17,500
723	Jetta	$17,500	2006	39,000	$17,500
725	Jetta	$18,000	2006	30,000	$17,500
132	Civic	$13,500	2005	86,000	$13,500
879	Civic	$15,000	2006	68,000	$15,500
322	Civic	$16,000	2006	73,000	$15,500

results in a separate table and join this with the base table as needed. However, this can be confusing for the user and makes many subsequent queries hard to specify. Therefore, we choose to forgo normalization and store the result of aggregation in an additional computed column (which is automatically added by the aggregation operator) with the value in this column repeated for all rows in each aggregation group. Table 2.3 shows an example, where an extra column "Avg_Price" is computed for cars of the same Model and Year.

Definition 11 (**Aggregation**). $\eta_{f,c,l}(S^j(R^j, C^j, G^j,\ O^j)) = S^{j+1}(R^j, C^j \cup \{column\}, G^j, O^j)$, *where column is the aggregation result column computed from* $f(c)$ *at group level* l.

Formula Computation (FC) (θ). This operator provides facility for users to perform mathematical operations on columns and create a computed column from the result. It takes parameter f (formula to compute). Each record in the result column is computed from an arithmetic expression involving values in the same row (in one or more columns). For example, from a sales table, a user wants a formula *revenue* for each product, computed as "$price * quantity$". As a computed column, the result column is automatically updated when underlying data is changed. This new column can be used in the same way as any other columns in the database.

Definition 12 (**FC**). $\theta_f(S^j(R^j, C^j, G^j,\ O^j)) = S^{j+1}(R^j,\ C^j \cup \{column\}, G^j, O^j)$, *where column is the result column computed from* R^j *based on the current grouping.*

Duplicate Elimination (DE) (δ). Since the spreadsheet model operates on multi-sets, duplicates are allowed in projection, grouping, and set union/difference. This is similar to relational implementations, but different from pure relational algebra. Where duplicate elimination is required, it must be invoked explicitly through the DE operator, which removes all duplicates from the current spreadsheet, similar to "distinct" in SQL. As a result of DE, computed columns (including both aggregation and FC) need to be re-computed. Ordering and grouping are not affected, since duplicates are already placed together where it matters.

Definition 13 (DE). $\delta(S^j(R^j, C^j, G^j, O^j)) = S^{j+1}(R^{j+1},\ C^j,\ G^j, O^j)$, *where* $\forall t \in R^j$, $t \in R^{j+1}$, *and* $\forall t_1, t_2 \in R^{j+1}$, $t_1 \neq t_2$.

2.3.3 Additional Housekeeping Operators

As introduced in Section 2.3.2, we can save the current spreadsheet anytime, and should also be able to load a saved spreadsheet, either for reading or for operations with the current spreadsheet. Thus we have **Save**, **Open**, and **Close** operators. In addition, we supply the **Renaming** operator for changing the name of a column.

2.4 Properties of the Spreadsheet Algebra

The spreadsheet algebra is for building an expressive and usable direct manipulation interface. In the first subsection below we show that the algebra can emulate *core single-block SQL queries*.

While complex expressions can be developed in the spreadsheet algebra, it is important to remember that the purpose of this algebra is to enable the manipulation of a spreadsheet visible to the user. Operator precedence and commutativity play an important role in this regard, as will become clear later. In the second subsection below we deepen our understanding of these two vital properties of spreadsheet algebra operators.

2.4.1 Expressive Power

We define a *core SQL single-block query expression* to be a statement of the form:

SELECT < projection-list > < aggregation-list >
FROM < relation-list >
WHERE < selection-predicate >
GROUP BY < grouping-list >
HAVING < group-selection-predicate >
ORDER BY < ordering-list >

with the projection-list being a subset of the grouping-list and the ordering-list a subset of the projection-list union aggregation-list, where all lists are comprised of column names.

Theorem 1. *For every core SQL single-block query expression there exists an equivalent expression in the spreadsheet algebra such that the result of evaluating either expression against any set of relations is identical.*

Proof. We prove the above theorem by providing a procedure for specifying a core SQL single-block query expression (denote it as s) with our algebra.

Step 1: One at a time, obtain the Cartesian product of each relation named in the relation-list, to obtain a single product working relation.

Step 2: Remove all join conditions from the where clause, if any; specify the remaining where clause using the selection operator.

Step 3. We specify each item in the grouping-list from left to right, using the grouping operator. The grouping operator takes as input a grouping column, and the level of this new grouping. We create a new level of grouping with each item.

Step 4. Specify aggregations, which could appear in both the SELECT and ORDER BY clause. In SQL, when there are multiple grouping levels (multiple items in the group-list), aggregation is computed over the finest level. We specify each aggregation using the aggregation operator accordingly.

Step 5. Specify the HAVING clause. Since we already created aggregations columns for the HAVING clause in step 4, we can apply selection operator on aggregation columns as required in the given group-selection-predicate.

Step 6. Specify ORDER BY clause using the ordering operator. As in step 4, the ordering is specified over the finest level of grouping.

Step 7: Project out all columns not included in the projection-list, one at a time.

We have thus completed the specification of a core single-block SQL query. □

2.4.2 Commutativity

If operator $\aleph_i$ commutes with $\aleph_j$, the order of evaluation of these two operators does not affect the result. Commutativity among the *complete set* of relational

operators has been studied in [105]. In relational algebra, under the condition that attributes in selection predicates are retained in projection, selection commutes with all operators. Under the same condition, projection commutes with every operator except set difference [105]. In spreadsheet algebra, all binary operators (set union, set difference, join, and product) involve a stored spreadsheet, and this impacts the applicability of commutativity laws for reasons that follow. When commuting selection with a set difference, we use the following formula [105]:

$$\sigma_F(E_1 - E_2) \equiv \sigma_F(E_1) - \sigma_F(E_2) \tag{2.1}$$

As we can see, this requires selection to be applied to a stored spreadsheet first. Since the other spreadsheet already occupies the data view, this can only be done in the background. This is against our direct manipulation principle, which dictates all changes to be seen directly by the user. A *point of non-commutativity* is created whenever a binary operator (set union, difference, join, product) is applied, meaning any operator instance after this point does not commute with instances before that. As explained above, since these binary operators manipulate both the current spreadsheet and a stored spreadsheet, distribution is not possible in the spreadsheet model. There is no way to apply operators to a spreadsheet being read in prior to its being read in for, say, a set union.

We say that a spreadsheet operator instance p *precedes* operator instance q if q requires columns created by p or q removes a column that p requires. For example, column-creating operators (e.g., aggregation) precedes selection that uses the aggregation result column. In order for two operator instances to commute, neither of them can precede the other.

For commutativity in spreadsheet algebra, we have the following observation:

Theorem 2. *In the spreadsheet algebra, selection, projection, FC, DE, and aggregation commute with one another, with themselves, and with grouping and ordering, provided that all precedence relations are satisfied.*

Proof sketch: We first verify the pair-wise commutativity for the five unary data manipulation operators – selection, projection, aggregation, formula computation (FC), and duplicate elimination (DE). We observed that they all commute with each other in spreadsheet algebra. Some pairs appear surprising, for example, aggregation and selection, which do not commute in relational algebra. They commute in spreadsheet algebra for two reasons: 1) aggregation result is stored as a separate column with repeated values instead of a single value, and 2) result values are updated once underlying data is changed (for example, by selection). A similar argument applies to DE and aggregation. We next examine whether the five operators commute with grouping and ordering operator. Obviously grouping and ordering do not commute with each other (for example, some ordering can destroy grouping), but they commute with data manipulation operators. The general reason for this is that grouping and ordering are maintained by data manipulation operators.

2.5 Query Modification

In an interactive query environment, users often find it useful to modify their query to obtain the desired query results. Frequently, these modifications are small, such as changing a threshold parameter in a selection condition. Suppose the user has performed n operations $\{\aleph_1, \aleph_2, ...\aleph_n\}$ in sequence (that is, $\aleph_i$ is executed earlier than $\aleph_j$ when $i < j$) on the data, and she intends to modify a condition specified in the i-th operation. In the naive case, the user has to begin from scratch, and repeat all n operations, making the desired change in the i-th operation. If the interface provides an UNDO facility, the user may not have to start over from scratch. But she still has to back up to the i-th operation, and re-specify everything from there onwards.

What we would like instead is for the user to be able to specify a change only to

the one affected i-th operation, and have the system take care of the rest. To be able to accomplish this, we need a notion of *query state*. The system keeps track of the history of operators specified by the user. The system could undo all operations back to the i-th and then re-do from there again. However, this is likely to take too long. The commutativity property of spreadsheet algebra, as stated in Theorem 2, can be used to reduce this cost substantially.

2.5.1 Query State

Instead of keeping every user action from the beginning, we keep query history until only as far back as we expect to be able to permit rewriting efficiently, which is the most recent point of non-commutativity. From a user perspective what this will lead to is that queries specified on a single sheet can be changed as needed, but where data from other sheets has been pulled in we cannot go back beyond.

For each selection or FC, we associate the predicate applied with the column(s) referenced in the predicate. With each column, we store the associated selection/FC predicates. This includes columns created by aggregation and FC.

For each projection, we retain a list of columns projected out.

For each aggregation, we retain, associated with the corresponding aggregate column, a definition of the aggregate function applied, and the grouping it is applied to.

For each grouping and ordering, we retain the corresponding grouping-list or ordering-list.

Notice that we did not store the query state as an ordered list of manipulations, but rather as individual operators associated with objects they affected. On account of operator commutativity, we can generate a history that is equivalent to the actual history of the spreadsheet.

Theorem 3. *In a direct manipulation interface, modifying an operation in a sequence of operations without point of non-commutativity through query state*

change is the same as re-writing query history.

Proof sketch: The proof of this theorem follows commutativity properties we established in Sec. 2.4.2. We store all operations, without ordering, in association with either the related columns or the operations themselves, so we are able to list all previous operations without order. Since operations in a sequence commute, in the absence of any point of non-commutativity, the order in which they are applied is immaterial. Hence changing any one operation from the query state has the same effect as changing it in the complete ordered list of operations.

2.5.2 Query Specification

Having introduced query state, we now show how to modify a previously specified query through query state. We could show the user the entire query state and let them specify what they wish to modify. However, non-technical users may have difficulty understanding the query state. Moreover, manipulating query state goes against the notion of direct manipulation of data, which is our objective. Instead, we selectively present history to the user as she attempts to redefine the invocation parameters of an operator, in a manner that we make precise next.

When a user begins to specify a selection predicate on a column, the user is given a list of selection predicates currently applied to that column, from the query state. The user then has an option of replacing a previously applied predicate with the one now being specified, or even of deleting the previously applied predicate altogether, without specifying a new predicate at all. History is rewritten, with the previously applied predicate removed, and replaced with the new one, if one is specified. Of course, the user also has the option of simply specifying the new predicate in addition to those previously specified. In this case, the new predicate is added "now", without rewriting history.

We can remove an existing selection predicate on any column, just as we can

Table 2.4: Results before Query Modification

ID	Model	Price	Year	Mileage	Condition
872	Jetta	$15,000	2005	50,000	Excellent
901	Jetta	$16,000	2005	40,000	Excellent
304	Jetta	$14,500	2005	76,000	Good

add a new one to any column. Putting these two together, we can also modify the selection predicate on any column.

We use an "inverse" projection operator to "reinstate" a column that has been projected out. We write this as $\Pi_{\bar{i}}(R)$. Since a column is either included or excluded in a projection, there is really no additional history to show. The semantics of the reinstatement are to rewrite history, and make it as if the projection never took place.

We can remove an aggregate column, provided that no operator depends on it. If a column that serves dependencies needs to be removed, all dependent columns must be removed first. Of course, we can always add a new aggregate column.

We can modify an existing grouping or ordering, provided that there is no operator that depends on it. Otherwise, those that depend on the grouping or the ordering should be removed first.

We now turn to the used car example. Suppose Sam initially started by searching for cars in year 2005, model "Jetta", and mileage lower than 80k. Results should be grouped by condition and ordered in ascending order of price. After seeing the results (as shown in Table 2.4), he discovers that his budget allows him to purchase a newer car. Sam can now simply choose the "Year" column, and change previous condition of "Year = 2005" to "Year = 2006". If he prefers, he can also modify the mileage condition in a similar fashion. All results are updated to meet the new condition(s), and the specification of model, grouping and ordering remains effective, as shown in Table 2.5.

Table 2.5: Results after Query Modification

ID	Model	Price	Year	Mileage	Condition
723	Jetta	$17,500	2006	39,000	Excellent
725	Jetta	$18,000	2006	30,000	Excellent
423	Jetta	$17,000	2006	42,000	Good

2.6 Interface Design

The reason to develop the spreadsheet algebra described above is to implement an effective and easy-to-use spreadsheet interface to a database system. We built a prototype named *SheetMusiq* to validate the ideas we presented, as part of the **Musiq** (**M**odel-driven **U**sable **S**ystem for **I**nformation **Q**uerying) effort at the University of Michigan [2].

SheetMusiq reflects all three principles of direct manipulation. First, users specify queries in SheetMusiq by mouse-clicks, with minimal keyboard input (e.g., for inputting constants to compare with). Most query operations are accessible with a *contextual* menu, which pops up when the user right-clicks a cell or column-header. It is *contextual* because it shows only options that are available for the current cell value type under current grouping and ordering. Second, SheetMusiq provides immediate and intuitive result presentation for users to easily specify conceptually difficult queries. It continuously presents the resultant spreadsheet after each manipulation, helping users to better adjust the next query step. Third, all user actions are reversible. Users can access query history (all historical manipulations) through a "History" menu, and the complete list of operations is shown as a numbered list, each with meaningful names. Users can do one-step or multi-step undo/redo of data manipulation. They can also do query modification to modify an operation in a sequence of operations, without having to repeat the effort to re-specify unchanged operations.

2.6.1 Design of Operators

We first introduce user interface design for each operator, then we re-visit the motivating problem in Section 2.1.1 using related operators.

Grouping. Grouping is accessed through a context menu. If the spreadsheet is already grouped by other column(s), the user is asked whether to add to the existing grouping (as the inner most level of grouping) or destroy the current grouping and use this new one instead. However, if there are aggregation columns that depend on the current grouping, user clicking on the latter option will trigger a reminder to first remove columns that depend on the grouping.

Ordering. Clicking a column header sorts the table according to the column in ascending order, and another click changes the sorting to descending order. The header of such a column has an up/down arrow, to show ascending/descending order. In the presence of grouping, the user is asked explicitly for the level of grouping to which the order should be applied. If the new ordering can destroy some grouping, the user is asked for confirmation to do so. If there are aggregates that depend on that grouping, this operation is not allowed, and user is suggested to project out the aggregates, if necessary.

Selection. Selection is accessed with a right-click of the mouse. If the click is on a cell instead of a column header, the user can choose to filter the results based on current cell value with another click, and the result is immediately shown.

Projection. We present a checkbox to the left of each column header. By default all checkboxes are checked. Users can remove a column conveniently by unchecking the checkbox (Figure 2.1). Columns that are projected out can be restored from a drop down menu.

Cartesian Product and **Set operators**. These binary operators involve an additional stored spreadsheet (which is saved previously by a clicking on the "Save" button). Once an operator is called through a contextual menu, the user is presented

with all stored-relations listed in a pop-up menu. The result is then presented as the current spreadsheet.

Join. Join also involves a stored spreadsheet. In addition to choosing a spreadsheet to join with, the user is prompted to graphically choose join conditions. Validity of join condition is checked and any invalid condition is reported to the user immediately. The result spreadsheet is shown in the screen as the current one.

Aggregation. A user can perform an aggregation function on an attribute by right clicking a cell and choosing "aggregation". She is then given a choice of aggregate function, and possibly the option to specify the grouping level on which the aggregation should be computed (when data is grouped by some column). Result column appears next to rightmost column.

Formula Computation. A dialog is shown, allowing the user to choose related columns and mathematical operators. The user can optionally give a name for the result column. Otherwise, the system automatically generates a name for it and reminds the user of the new column. The new column is added to the right of all existing columns.

Duplicate Elimination. DE is accessible by a single right-click, and all duplicates are removed.

Having introduced all operators, we show how Sam could use the operators to explore the car database. Recall that Sam likes cars of Year 2005 or later in good or excellent condition, and the rest is open-ended. Since he cares about Model and Price the most, he first groups the data by "Model" and "Year", and selects "Good" and "Excellent" condition. He also selects his Models of interest: "Jetta" and "Civic". Now he wants to know the average price for the Model and Year so that he does not overpay. To accomplish that, he computes the average price using Aggregation, where he is asked to compute average over all the cars or just cars of the same Model and Year (show in Figure 2.1). Choosing the latter leads the spreadsheet in Table

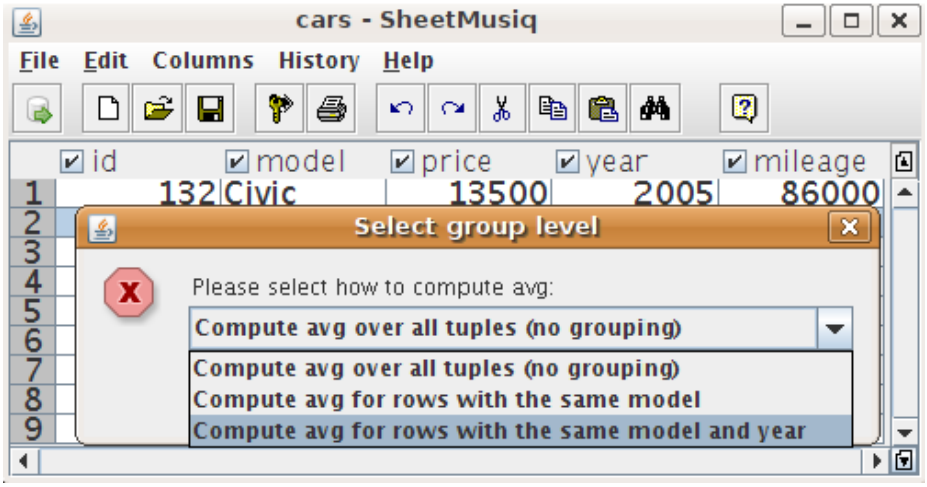


Figure 2.1: Aggregation under Grouping

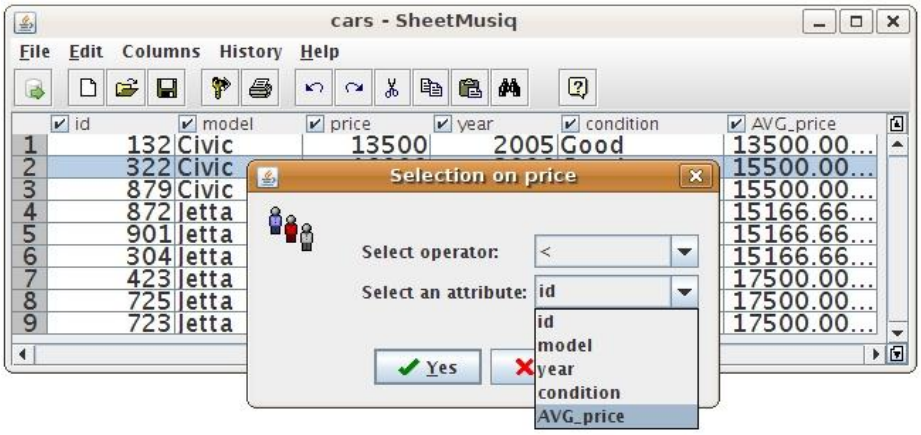


Figure 2.2: Compare Price with Avg_Price

2.3. Now he can filter out all cars more expensive than the average, as shown in Figure 2.2, where he chooses to compare "Price" with "AVG_price".

2.7 Evaluation

In this section we experimentally evaluate the algebra with our prototype, SheetMusiq. The database server was PostgreSQL 8.3 (beta 2).

2.7.1 User Study

We now measure the usability of SheetMusiq with user studies. Since the interface is built specifically to help people without knowledge of any database query language (which rules out any direct SQL query interface), we want to compare SheetMusiq with a tool that meets the same requirement and with similar expressive power. Graphical query builders are the closest existing tools that satisfy the requirement. We found an abundance of such packages, and they are mostly similar. We chose Navicat for PostgreSQL [3] as a representative to compare with. All experiments were performed on a laptop (Intel Core 2 Duo 2.16GHz CPU, 2GB of RAM, and running Windows XP).

Methods

We recruited ten volunteers with no background in database query languages. Their ages ranged from 24 to 30, and they all have at least a bachelor's degree (in various fields).

The data and queries are from the TPC-H benchmark [6]. We used the demonstration dataset in the benchmark, which was 31MB in size. Since TPC-H queries are quite complex, some of them contain features that SheetMusiq does not yet support. Specifically, SheetMusiq does not support nested queries and queries with keyword "exist" and "case". This leaves us 10 queries out of the original 22 in the benchmark. In addition, we predefined views for queries involving many joins so that users always query a single table.

A brief tutorial (introduction with examples for both SheetMusiq and Navicat) was given to each subject prior to the study. At the end of the tutorial, subjects completed a sample query, with help available upon request from the examiner. Each participant then completed all queries in the query set, using both SheetMusiq and Navicat, separately. For each user, we gave her/him time to understand the

query definition. We started measuring time when the user decided the query was fully understood and he/she was ready to specify the query. Since the software that is used first has a potential disadvantage, we alternate the order of which software was used first for the queries. In the end, each package was used first half the time.

For each subject, we measure speed (time for each task) and correctness. During the experiment, if a user did not finish the query in 900 seconds, the task was considered finished with wrong results, and the time was counted as 900 seconds.

Speed Results

We measure the time taken by the subjects for completing each query. Figure 2.3 shows the average time for each user to complete the 10 queries, using Navicat and SheetMusiq, respectively. Most queries were completed significantly faster with SheetMusiq than Navicat. Using the Mann-Whiteney test we found the speed result is statistically significant (with p-value < 0.002) for all queries except query 5, 7, and 10. For those three queries, the speed performance is comparable with both packages. We think the reason is that the three query tasks are relatively simple, and subjects can finish both in a short time. Furthermore, as shown in Figure 2.4, the standard deviation for SheetMusiq is much smaller on most queries, demonstrating the consistency of superior efficiency using SheetMusiq.

Correctness Results

We consider correctness of queries the subjects finished. Figure 2.5 shows the number of users that completed the queries correctly using the two approaches. SheetMusiq users correctly finished more queries (95 out of 100) than Navicat users (81 out of 100). If we consider each query alone, we can not establish a statistically significant conclusion. Hence we consider the total number of correctly answered queries. Using Fisher's exact test we conclude that SheetMusiq is statistically better than Navicat

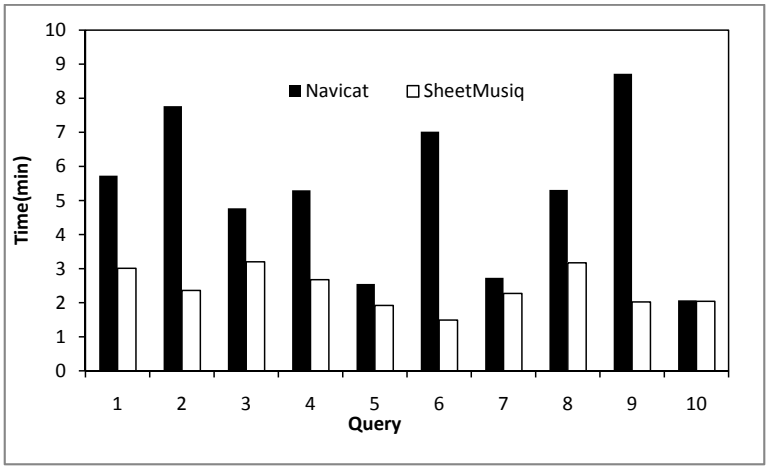


Figure 2.3: Speed Result

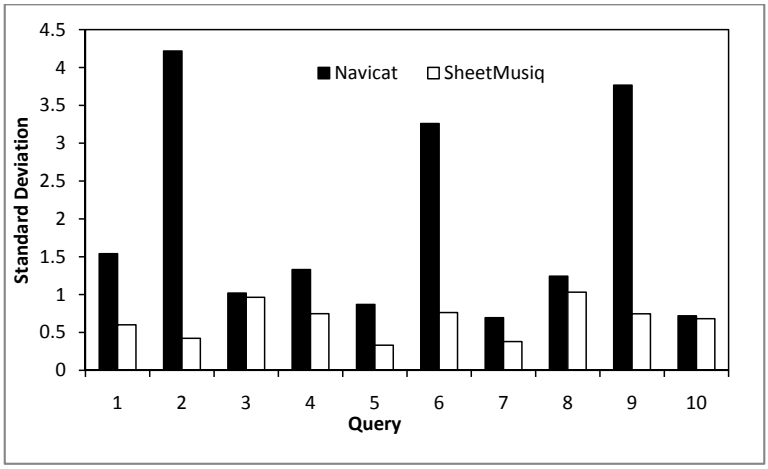


Figure 2.4: Standard Deviation of Speeds

(in leading to more correctly answered queries), with p value < 0.004.

Analysis

We now analyze why better speed and higher accuracy were observed for SheetMusiq.

Navicat, like most other graphical query builders, has two separate windows for building a query – a graphical window where users manipulate with mouse-clicks and a text window for SQL query expression. Usually, only queries with simple selection, sorting, and joins can be built graphically, while the vast majority of the

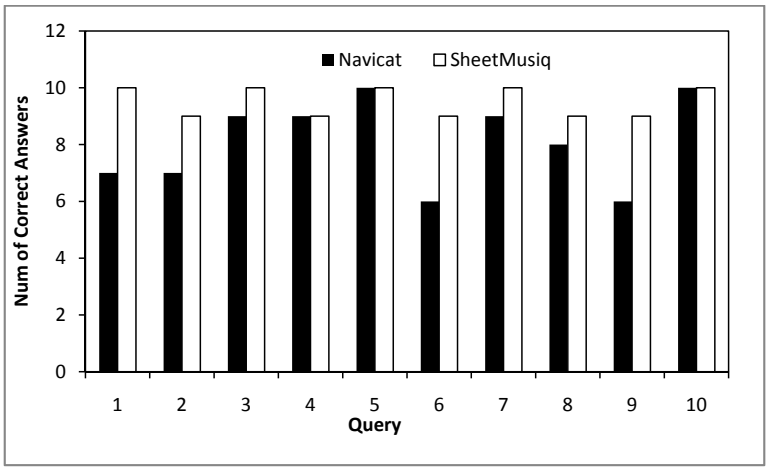


Figure 2.5: Correctness Result

queries need to be completed by adding to the SQL query. SheetMusiq never reveals or requires the user to know a SQL query. This leads to several benefits, which directly led to the superior performance of SheetMusiq. First, we found that most users picked up SheetMusiq much faster than Navicat (also shown by results of the first two queries). Second, users never stuck on syntactical errors in SheetMusiq, which often happen in Navicat.

Many users had difficulties accomplishing certain tasks in Navicat but did that effortlessly in SheetMusiq, largely because Navicat (and almost all graphical query builders we surveyed) does not have direct manipulation support for database concepts other than simple selection, sorting, and join and SheetMusiq does. We now give a few examples of such tasks. First, selection based on aggregation. In Navicat, users have to resort to a *sub-query*, which is a very difficult concept for non-expert users. In SheetMusiq, this can be accomplished by an aggregation followed by selection, both via mouse-clicks. Second, grouping is much easier in SheetMusiq. In Navicat, users have no choice but to understand the concept and syntax of grouping, as well as many related restrictions (e.g., aggregation). This is very challenging for almost every participant. In SheetMusiq, users do not need to

Table 2.6: Subjective Results

Question	Answers	Count
Which package do you prefer to use?	Navicat	0
	SheetMusiq	10
Seeing data helps formulate queries	Yes	10
	No	0
Progressive refinement is better than specifying a query all at once	Yes	8
	No	2
Database concepts are easier in SheetMusiq	Yes	10
	No	0

know the syntax (mouse right-click is sufficient), and they understand the concept faster with immediate visual feedback. Third, group-qualification. While many users struggled with the "having" clause in Navicat, they found it very intuitive to filter groups with mouse-clicks. This list could go much longer if space were permitted.

Subjective Results

Besides numerical evidence, each participant was asked for opinions on which program they would prefer and whether certain features are desirable. Results are presented in Table 2.6. All subjects preferred SheetMusiq over Navicat, and all agreed that being able to see the data helps formulate a query (second question). They also found that many concepts (e.g., group-qualification) are more intuitive and easier to understand in SheetMusiq (fourth question). Eight of ten subjects preferred progressive refinement of a query over specifying it all at once (third question).

2.8 Related Work

Direct manipulation [93], although a crucial concept in the user interface field, is seldom mentioned in database literature. Pasta-3 [62] is one of the earliest efforts attempting a direct manipulation interface for databases, but its support of direct manipulation is limited to allowing users to manipulate *a query expression*

with clicks and drags. Tioga-2 [16] (later developed into DataSplash [78]) is a direct manipulation database visualization tool, and its visual query language allows specification with a drag-and-drop interface. Its emphasis, however, is on visualization instead of querying.

Spreadsheets have proven to be one of the most user-friendly and popular interfaces for handling data, partially evidenced by the ubiquity of Microsoft Excel. FOCUS [99] provides an interface for manipulating local tables. Its query operations are quite simple (e.g., allowing only one level of grouping and being highly restrictive on the form of query conditions). Tableau [4], which is built on VizQL [45], specializes in interactive data visualization and is limited in querying capability. Spreadsheets have also been used for data cleaning [84], logic programming [98], visualization exploration [52], and photo management [56]. Witkowski et al [107] proposed SQL extensions supporting spreadsheet-like computations in RDBMS.

In addition to desktop tools, we have seen many online database query and management tools using spreadsheet interfaces. For example, Zoho DB [7] allows importing, creation, querying, and visualizing databases online. Zoho DB's querying capability, however, is still primitive (allowing only simple filtering and sorting, with the rest resorting to SQL). Dabble DB [1] is similar to Zoho DB, with the additional feature of supporting grouping.

Much effort has been spent on a visual querying interface for relational databases, starting with Query-by-Example [115]. Query-by-Diagram [25] allows users to query by manipulating ER-diagrams. T. Catarci et al. surveyed many early interfaces and visual query languages in [23]. [24] uses ontologies to help users with their vocabulary in formulating queries. VisTrails [88] provides a visual interface for users to keep track of incrementally specified workflows and to modify them incrementally. [60] proposes a visual query language that enables users to query a database by manipulating "schema trees".

2.9 Conclusion

A spreadsheet-like "direct manipulation" query interface is desirable for non-technical database users but challenging to build. In this chapter, we design a spreadsheet algebra that enables the design of an interface that: i) continuously presents the data to users, after each data manipulation, ii) divides query specification into progressive refinement steps and uses intermediate results to help users formulate the query, iii) provides incremental reversible data manipulation actions, iv) enables the user to modify an operation specified many steps earlier without redoing the steps afterwards, and v) allows the user to specify at least all single-block SQL queries while shielding her from complex database concepts. We built a prototype, SheetMusiq, with our algebra and evaluated it using user studies with non-technical subjects, in comparison with a commercial graphical query builder. Results show that the direct manipulation interface leads to easier and more accurate specification of queries, and it is welcomed by non-technical users.

CHAPTER III

RESULT REVIEW: GENERATING REPRESENTATIVES THROUGH *MUSIQLENS*

3.1 Introduction

3.1.1 Motivation

Database queries often return hundreds, even thousands, of tuples in the query result. In interactive use, only a small fraction of these will fit on one display screen. This chapter studies the problem of how best to present these results to the user.

The "Many-Answers Problem" has been well documented [27]: too many results are returned for a query that is not very selective. This problem arises because: i) it is very difficult for a user, without knowing the data, to specify a query that returns *enough* but not *excessive* results; and ii) often a user starts exploring a dataset without an exact goal, which becomes increasingly clear as she learns what is available. Consider Example 3.1.1 below, where a user searches a used car database for a Honda Civic.

Example 3.1.1. *Ann wants to buy a car, and visits a web site for used cars. The web site is backed by a database that we simplify for this example to have only one table "Cars" with attributes* ID, Model, Price, *and* Mileage. *Ann specifies her requirements through a form on the web site, resulting in the following query to the database:* `Select * from Cars where Model = 'Civic' and Price < 15,000 and Mileage < 80,000.` *The query she formulates may have thousands of results since it is on a popular model with unselective conditions. How should the web site show these results to Ann?*

A common approach to displaying many results is to batch them into "pages".

The user is shown the first page, and can navigate to additional pages as desired, and "browse" through the result set. For this process to make sense, the results must be organized in some manner that the user understands. One popular solution is to sort the results, say by Price or Mileage in our example. However, this sorting can be computationally expensive for large result sets. More important, similar results can be distributed many pages apart. For example, a car costing 8500 with 49000 miles may be very similar to another costing 8200 with 55000 miles, but there could be many intervening cars in the sort order, say by price, that are very different in other attributes (e.g. high mileage but recent model year, high mileage but more features, low mileage but in an accident, and so on).

Another possibility is to order results by what the system believes is likely to be of greatest interest to the user. Indeed, there is a stream of work [37] trying to develop ranking mechanisms such that the "best" results appear first. Such techniques can be successful when the system has a reasonable estimate of the user's preference function. However, determining this can be hard: in our example the system has no way to tell what Ann's tradeoff is for price versus mileage, let alone other attributes not even mentioned.

This "Many-Answer Problem" has also attracted much attention from the information retrieval community. The importance of the first page of results for a search interface has been well documented [10, 53]. It has been shown that over 85% of the users look at only the first page of results returned by a search engine. If there is no exact answer in the first page to meet users' information need, the first page needs to deliver a strong message that there are interesting results in the remaining pages.

In this chapter, we solve the "Many-Answer Problem" starting from a user's point of view. Psychological studies have long shown that human beings are very capable of learning from examples and generalizing from the examples to similar

objects [77, 96, 91]. In a database querying context, the first screen of data can be treated as examples of a large dataset. Since users can expect more items similar to the examples, we should make them as representative as possible.

To accomplish the above task, we propose a framework called *MusiqLens*, as part of the MUSIQ project [2] from the University of Michigan. MusiqLens is designed to: i) automatically displays the best *representatives* result tuples in the first screen of results when the result set is large, ii) at user's request, displays more representatives similar to a particular tuple, and iii) automatically adapt to user's subsequent query operations (selections and joins). This is exemplified in Fig. 3.1. Notice that each tuple represents many cars with similar Price and Mileage. The representatives naturally fragment the whole dataset into clusters such that cars of various price and mileage range are shown. The representatives themselves have a high probability of being what the users want. If they are not, they can lead to more similar items. On the right side of each representative tuple, the number of similar items is displayed. A hyper-link is provided for the user to browse those items. Suppose now the user chooses to see more cars like the first one. Since they cannot fit in one screen, MusiqLens shows representatives from the subset of cars (Fig. 3.2). We call this operation "zooming-in", in analogy to zooming into finer level of details when viewing an image. After seeing the first screen of results, if the user now has more confidence to further lower the price condition (since there are more than 100 cars with price around $10k), she could add a condition `price < 10,000`. The next screen of results are generated with the same spirit. By always showing the best representatives from the data, we enable users to quickly learn what is available in the data without actually seeing all the tuples. We have built a prototype of MusiqLens. See [68] for a demonstration. [1]

ID	Model	Price	Mileage	*Zoom-in*
643	Civic	14,500	35,000	311 more Cars like this
876	Civic	13,500	42,000	217 more Cars like this
321	Civic	12,100	53,000	156 more Cars like this
452	Civic	11,200	63,000	87 more Cars like this
765	Civic	10,200	71,000	65 more Cars like this
235	Civic	9,000	78,000	43 more Cars like this

Figure 3.1: MusiqLens Example

ID	Model	Price	Mileage	*Zoom-in*
643	Civic	14,500	35,000	71 more Cars like this
943	Civic	14,900	25,000	63 more Cars like this
987	Civic	14,700	28,000	55 more Cars like this
121	Civic	14,300	40,000	45 more Cars like this
993	Civic	14,100	43,000	40 more Cars like this
937	Civic	13,900	47,000	37 more Cars like this

Figure 3.2: After Zooming on First Tuple

3.1.2 Challenges

Several challenges must be addressed before one can construct an effective interface such as the one shown in Fig. 3.1. We discuss these below. Let the first page of results be limited to k tuples. We call these tuples on the first page *representatives* of the whole result set.

Representation Modeling Our first problem is to determine what it means for a small set of points to "represent" a much larger data set. How can we choose between two (or more) choices of possible representatives? Although it is generally accepted that humans can learn from examples, to our knowledge there is no gold standard for generating those examples.

A naive approach is to display results sorted by some attributes. This approach only presents to users a very small fraction of results at the boundary of the value

[1]Note that *MusiqLens* was named *DataLens* in the demonstration paper.

domain and makes it impossible to find other tuples (for example, a car that balances the price and mileage). Should we uniformly sample k tuples from the results? While this can reflect the density of data distribution, it misses small clusters that may interest the user. Should we sample the results using density biased sampling [79] instead? We need to answer these questions and find a metric that matches human information seeking behavior.

Representative Finding Challenge Once the representation model is decided, we need to efficiently find representatives for the result set that are "best" in this model. MusiqLens will impose some overhead, but the waiting time perceived by the user should not be significant relative to the time the database server needs to finish the query.

Query-Refinement Challenge In the application scenarios of interest to us, such as a used car purchase or a hotel booking, users are typically exploring available options. Queries will frequently be modified and reissued, based on results seen so far. For example, Ann may decide to restrict her search to cars with less than 60,000 miles (instead of the 80,000 originally specified). In addition, once we show representative data points we should permit users to ask for "more like this," an operation we call *zooming in*. See Fig. 3.2. Such operations must be fast, which means that we can probably not afford to recompute representatives from scratch.

3.1.3 Contributions

Our first contribution is the MusiqLens framework for solving the "Many-Answers Problem" in database search. We propose to generate best representatives from a result set to show on the first result page. Based on the representatives, users can obtain a global and diversified view of what is available in the data set. Users can drill down by choosing to view more items similar to any tuple in the screen.

Our second contribution is the development of a representation model and metric.

Since the ultimate purpose is for users to learn about the data, we compared several popular candidates with a user study. Results are reported in Sec. 3.2, and show that k-medoid clustering with minimum average distance to be the technique of choice.

The third contribution is a fast algorithm to find representative data points. Based on the cover-tree structure, we are able to generate high-quality k-medoids clusters, for metrics of average-distance or max-distance. This algorithm is presented in Sec. 5.3. Experiments show the distance cost and computational cost are both superior over the state-of-the-art.

The fourth major contribution is algorithms for maintaining representative samples under common query operations. When a query is applied, some of the original samples may still qualify to be in the answers and some are not. How to generate new representative samples without rebuilding the index from scratch? We devised algorithms for handling selection and projection operators such that we always have a valid cover tree index, and we can incrementally adjust the set of representatives in response to query refinement. These algorithms are presented in Sec. 3.4.

Our final contribution is a thorough experimental study of our algorithms, compared with the state-of-the-art competitor (R-tree based algorithm), presented in Sec. 3.5. Experiments show that: i) for generating initial representatives, we achieve better quality results (in terms of distance metric) in shorter time, and ii) our algorithms can adapt to selection and projection queries efficiently while R-tree based algorithms cannot.

3.2 What is a Good Set of Representatives

Given a large data set, our problem is to find a small number of tuples that best represent the whole data set. In this section, we evaluate various options. Note that statistical measures, such as mean, variance, skew, moments, and a myriad

of more sophisticated measures, can be used to characterize data sets. While such measures can be important in some situations, we believe they are not suitable for lay users interested in data exploration. Even for technically sophisticated people like members of our community, a few sample hotel choices convey a much better subjective impression than statistical measures of price and location distributions. As such, we only consider using selected tuples from the actual result set as a representative set.

3.2.1 Candidate Representative Choices

We consider the following methods for choosing representatives:

1. **Random selection**. Generate uniformly distributed random numbers in the range of [1, < Data Set Cardinality >] and use them as index to select cars as samples. This is a baseline against which to compare other techniques.

2. **Density biased sampling**. It is argued that uniform sampling favors large clusters in the data and may miss small clusters. We therefore use the algorithms by Palmer and Faloutsos [79] to probabilistically under-sample dense regions and over-sample sparse regions.

3. **Select *k*-medoids**. A *medoid* of a cluster of data points is the one whose average or maximum dissimilarity is the smallest to other points. We denote the two kinds of medoids as *avg-medoid* and *max-medoid*, respectively. Under the most commonly used Euclidean distance metric, we select k avg-medoids and max-medoids from the data. Note that k-means clustering is frequently used, and is very similar. We do not consider that since the mean values obtained may not represent actual data points, and so may mislead users.

4. **Sort by attributes**. Since sorting is the standard facility provided in systems today, we consider this choice as well. We note that sorting is one attribute at

a time in a multi-attribute scenario.

5. **Sort by typicality**. Hua et al. [48] proposed to generate the most "typical" examples. They view all data as independent identically distributed samples of a continuous random variable, and they select data points where the probability density function (estimated from the data) has highest values.

In the rest of the chapter we use the following abbreviations for each method: Random (random samples) , Density (density-biased sampling), Avg-Med (avg-medoids), Max-Med (max-medoids), Sort-<attr> (sorted by attribute <attr>), and Typical (sorted by typicality).

3.2.2 Data

We obtained information about cars of model Honda Civic from Yahoo! Autos. For each car, we have values for numerical attributes Mileage and Price. The site limits the total number of results for a particular type of car to 4100 items, some of which do not have mileage or price information and are removed. This leaves us with 3922 cars that we used in our study.

In Fig. 3.3 we show representatives generated using all above methods (note that all data have been normalized to range [0, 1]). The whole dataset is shown in the background of each figure. We can see visually that sorting does poorly, whether we sort first by price or by mileage. Even sorting by typicality does poorly, giving us a few points near the "center", but no sense of where else there may be data points. We also see that Avg-Medoids, Max-Medoids and Density-biased Samples all appear to do much better than random samples. We further see that Max-Medoids seems to choose points that delineate the boundary of the data set whereas Avg-Medoids gives us points in the "center" of it, with density-biased samples somewhere in between.

3.2.3 User Study

The goal of choosing representative points is to give users a good sense of what else to expect in the data. While each of us can form a subjective impression of which scheme is better by looking at Fig. 3.3, we would like to verify this through a careful user study. Towards this end, we recruited 10 subjects, and showed them the seven sets of representative points in random order, without showing them anything else about the data, and not telling them that these were all for the same distribution. For each set of representatives, we sought to elicit from the users what the rest of the data set may look like.

Eliciting information regarding an imagined distribution is very tricky. We cannot get into the head of the user! After considering many alternatives, we settled on asking the users to suggest a few additional points that they would expect to see in the data set. (We required these points not to be "too close" to the representatives provided or to one another – but all of our subjects naturally adopted this constraint without explicit direction from us). We require that the points suggested by the user can not be any existing point. Given a set of predicted data points, we can measure how far these predictions are from actual data set values. For each point in the dataset, we find the distance to the closest point in the predicted set. We call this the *prediction error distance* for that data point. If the minimum prediction error distance is small, that tells us that an individual predicted point is good, but says nothing about the overall data set. If the maximum prediction error distance is small, that tells us that there is no very poor prediction – the user has not been misled about the shape of the data set. Finally, if the average prediction error distance is small, that gives us a global metric of how well the set of predicted points as a whole match the actual data set. We refer to these three metrics as MinDist, MaxDist, and AvgDist, respectively. We computed values for all three, averaged across all participants. The results of AvgDist and MaxDist are shown in Fig. 3.4

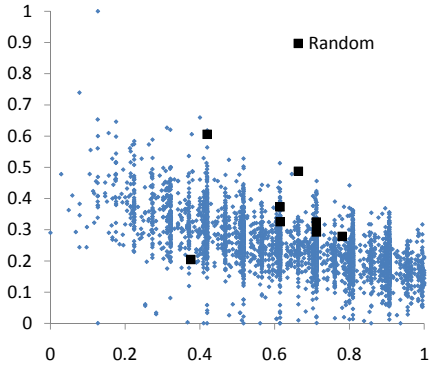
Table 3.1: p-value of Mann-Whitney Test

	Random	**Avg-Med**	**Sort-Mile**	**Density**	**Sort-Price**	**Max-Med**	**Typical**
AvgDist, Avg-Med vs.	<0.0001	NA	<0.0001	0.0087	<0.0001	<0.0001	<0.0001
MaxDist, Max-Med vs.	0.0228	<0.0001	<0.0001	<0.0001	<0.0001	NA	<0.0001
MinDist, Avg-Med vs.	0.0011	NA	0.0018	**0.1922**	<0.0001	0.0104	0.0446

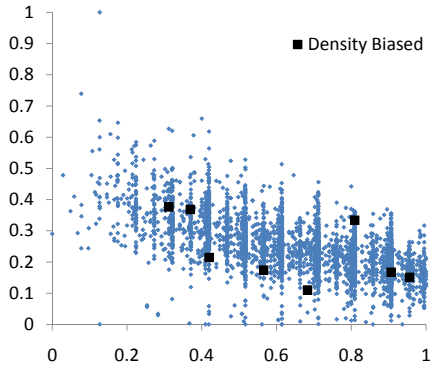
(a), and MinDist is shown in Fig. 3.4 (b) using a different scale in the y-axis.

In Fig. 3.4 (a), avg-medoids (Avg-Med) stands out as the best based on AvgDist measurement, while max-medoids (Max-Med) is the best in MaxDist measurement. For MinDist measurement (Fig. 3.4 (b)), the winner is not clear. Among the two best choices, avg-medoids has a smaller value than density-biased sampling (0.00161 vs. 0.00253). However, the values are too small to be statistically significant. We calculated the statistical significance using Mann-Whitney test to verify the above observation. p-values are shown in Table 3.1. The first row shows the p-values of Avg-Med against others under the AvgDist metric, second row shows that of Max-Med against others under MaxDist metric, and the third row shows Avg-Med against others under MinDist metric. All values are significant, except one – Avg-Med vs. Density under MinDist metric, meaning that the two are similar in performance. Since Avg-Med is clearly better than Density under AvgDist metric, it is overall more desirable. In summary, if we consider AvgDist and MinDist metric, avg-medoids is the choice; if we consider MaxDist, max-medoids is the best.

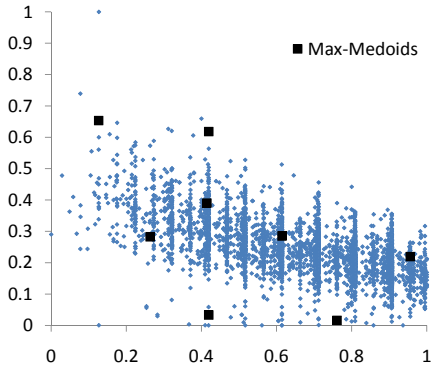
The conclusion from the investigation described above is that k-medoid (average) cluster centers constitute the representative set of choice. k-medoid (maximum) cluster centers may also make sense in a few scenarios. Even though the rest of the chapter will focus only on the former, computation of the latter is not that much different, and the requisite small changes are not hard to work out. For the rest of the chapter, we refer to average medoids when we use the term *medoid*. Formally, for a set of objects $\boldsymbol{O}$, *k-medoids* are a subset $\boldsymbol{M}$ from $\boldsymbol{O}$ with k objects, which minimize the average distance from each point in $\boldsymbol{O}$ to the closest point in $\boldsymbol{M}$.



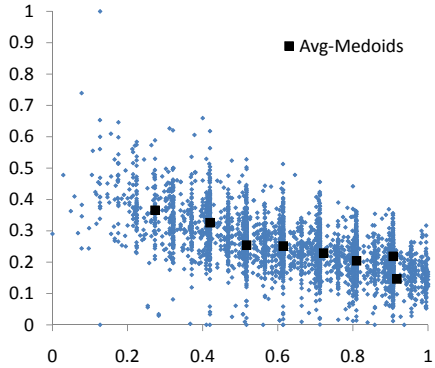
(a) Random Samples



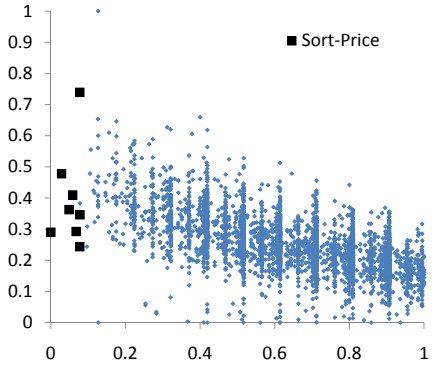
(b) Density-biased Samples



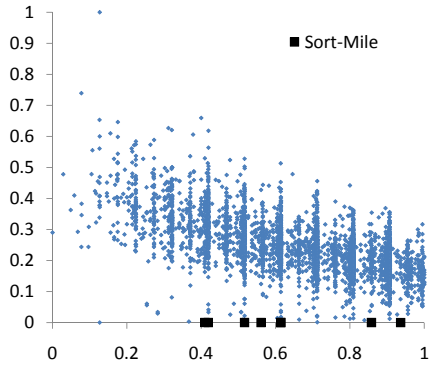
(c) Max-Medoids



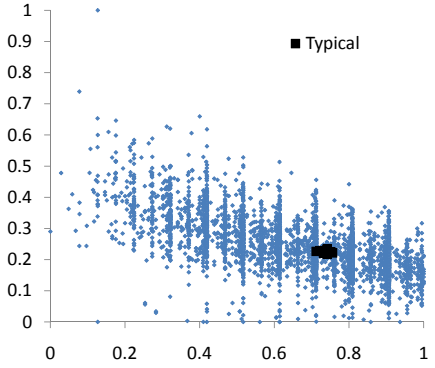
(d) Avg-Medoids



(e) Sort by Price

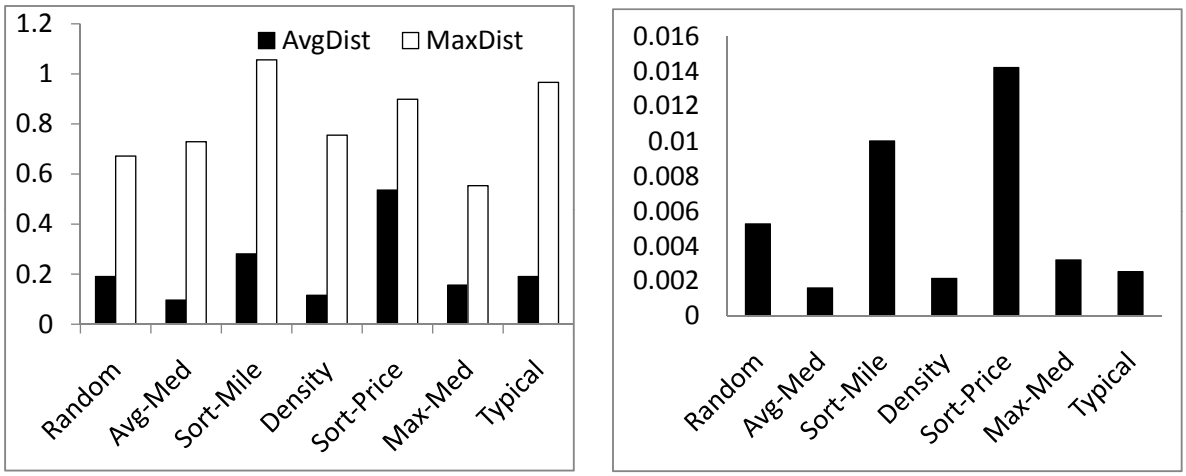


(f) Sort by Mileage



(g) Sort by Typicality

Figure 3.3: Samples Generated Using Different Methods. Light points are actual data, and dark points are generated samples.



(a) AvgDist and MaxDist (b) MinDist

Figure 3.4: Average Distance Results for the Seven Methods

3.3 Cover-tree Based Clustering Algorithm

Clustering has been studied extensively. Many clever techniques have been developed, both to cluster data sets from scratch and to cluster with the benefit of an index. See Sec. 3.6 for a short survey. Unfortunately, none of these techniques address the query-refinement challenge or even support incremental recomputation. As such, we must develop a new algorithm to meet our needs.

We propose using the cover-tree [21] data structure for clustering. The properties of cover-tree (which will be discussed shortly) make it a great structure for sampling. This immediately reduces the problem of finding medoids from the original data set to finding medoids in the sample. We then use statistics gathered during the tree construction phase to help find a good set of medoids. We begin by providing some brief background on the cover-tree in Sec. 3.3.1, followed by our novel contributions in the subsequent sub-sections.

3.3.1 Cover-tree

Cover-tree was proposed by Beygelzimer, Kakade, and Langford in 2006 [21]. It is so named because each level of the tree is a "cover" for the level beneath it. For convenience in explanation, we assume that the distance between any two data points is less than 1 (we will see later how this condition can be relaxed). Following convention, we number the levels of the tree from 0 (root level). For level i, we denote the value of $1/2^i$ as *D(i)*, which is a monotonically decreasing function of i. The condition that distance between any two points is less than 1 can be relaxed if we allow i to be negative integers. A cover-tree on a data set S has the following properties for all levels $i \geq 0$:

1. Each node of the tree is associated with one of the data points s_j.

2. If a node is associated with data point s_j, then one of its children must also be

associated with s_j (nesting).

3. All nodes at level i are at separated by at least $D(i)$ (separation).

4. Each node at level i is within distance $D(i)$ to its children in level $i+1$ (covering).

Fig. 3.5 shows a cover-tree of scale 2 for data points s_1 to s_7 in 2-dimensional space. Nesting property is satisfied by repeating every node in each lower level after it first appears. Covering property ensures that nodes are close enough to their children. Separation property means that nodes at higher levels are more separated (e.g., nodes s_1, s_4, s_5 are far away from root node s_7). Points s_1 and s_2 are at a larger distance from each other than are s_5 and s_6. Thus s_2 is a child of $s1$ at level 2 while $s6$ is a child of s_5 at level 3 (where inter-node distance is smaller). We can prove that the distance from any descendant to a node at level i is at most $2 \times D(i)$ (using the convergence property of $D(i)$). Cover-tree naturally provides a hierarchical view of the data based on the distance among nodes, which our algorithms will exploit.

The cover-tree shown in Fig. 3.5 is a theoretical *implicit* tree, where every node is shown. It is neither efficient nor necessary to repeat a node when it is the lone child of itself in intermediate levels (for example, s_7 at level 1 and 2). In practice, we use an *explicit* tree, where such nodes are pruned. So every explicit node either has a parent other than itself or or a child other than a self-child. We call the rest of the nodes *naive nodes*. We use the implicit cover-tree throughout this chapter for the ease of understanding. All algorithms in this chapter can be easily adapted to the explicit tree.

A very important property of cover-tree is that the subtree under a node spans to a distance of at most $2 \times D(i)$, where i is the level at which the node appears. We call this distance the *span* of the node. For example, point s_5 first appears in level 1. The actual span of s_5, however, is best obtained when it appears again at level 2,

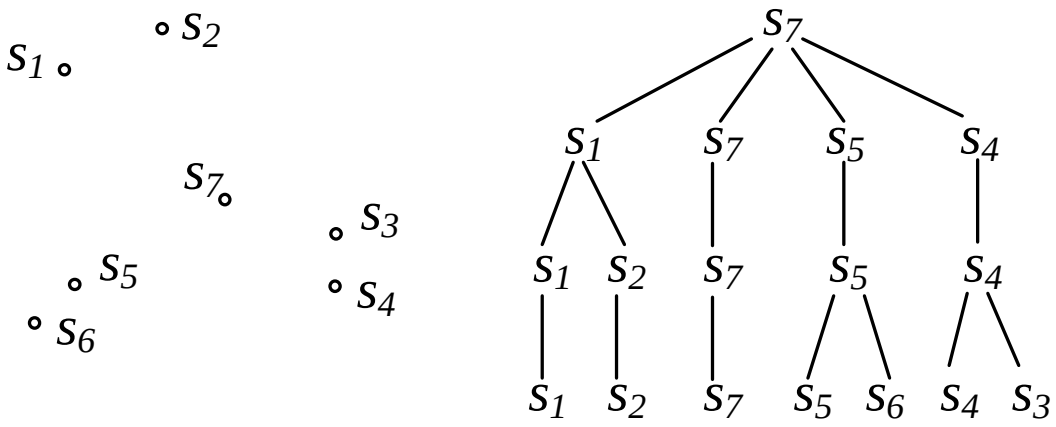


Figure 3.5: Cover Tree Example

where it has a non-self child. In both levels, we are in fact trying to get the range of the same subtree. The span obtained at level 2 is half of that obtained at level 1, and it gives more accurate information about the subtree. In the rest of the chapter, we always use *span* to refer to the least possible span that can be obtained for the subtree, which is achieved by descending from the root of the subtree to the node that has a non-self child.

The explicit cover-tree has a space cost to $O(n)$, and it can be constructed in $O(n^2)$ time. The tree is bounded in width (number of children of any node) and depth. For more details regarding properties and algorithms in cover tree, we refer the readers to the original paper [21] since they are out of the scope of this chapter.

3.3.2 Using the Cover Tree

Additional Statistics

In order to better grasp the distribution of data, we need to gather some additional statistics of the subtree rooted at each node s_i:

- **Density**. This is the total number of data points in the subtree rooted at node s_i. Note that this includes all descendants of the node. For all nodes at the

same level in the cover tree, a larger density indicates that the region covered by the node is more densely populated with data points.

- **Centroid**. This is the average value of all data points in the subtree. Assume that there are T points in total in the subtree. For node s_i, if we denote the N-dimensional points in the subtree as $\overrightarrow{X}_j$ where $j = 1, 2, ..., T$, then $Centroid(i) = \frac{\sum_{j=1}^{T} \overrightarrow{X}_j}{T}$.

We refer to the density and centroid for the subtree under node s as $DS(s)$ and $CT(s)$, respectively. The exact use of density and centroid of a node will become apparent in later sections. Both values can be collected when the tree is being built. As each point is inserted, we increase the density for all its ancestors. Assume the new data point inserted is $\overrightarrow{X}_j$, then for each node i along the insertion path of the new point, we update the density and centroid as follows:

$$\begin{aligned} DS(s)' &= DS(s) + 1 \\ CT(s)' &= \frac{CT(s) \times DS(s) + \overrightarrow{X}_j}{DS(s) + 1} \end{aligned}$$

Both operations can be accomplished with a minor change in the recursive insertion algorithm of the cover tree [21].

Distance Cost Estimation of Candidate k-medoid

Using density and centroid information, we can obtain an estimate of the average distance cost for a set of candidate k-medoids, using any level of nodes in the cover tree, without having to read the whole data set. We illustrate using the example in Fig. 3.6, where we have 8 nodes (s_1 to s_8) and two medoids (m_1 and m_2). Note that each node actually represents many other data points in the subtree. Also, these 8 nodes should form a cover of the tree - they should be *all* the nodes in a certain level of the cover tree. An arrow means the node is closest to the pointed medoid. The total number of data points can be found by summing up the density of each node.

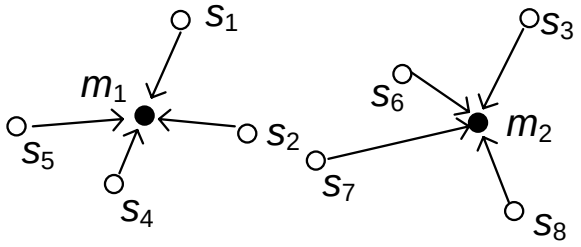


Figure 3.6: Distance Cost Estimation

Since we do not want to read all data points under a subtree, we use the centroid maintained at the root to stand for the actual data points when calculating the distance to the medoid. For example, to calculate the total distance from all data points under node s_1, we compute the distance from the centroid of s_1 to m_1, and multiply it by its density. We do the same for all other nodes and sum up the total distance. This value is then averaged over the total number of points, and we have obtained an estimate of the average distance cost.

3.3.3 Average-medoids Computation

We now introduce our algorithm for k average medoid queries. We start by traversing the cover tree from the root until we reach a level with more than k nodes. Assuming this is at level number m, and there are t nodes in this level of the tree. Following notations introduced earlier, we refer to the set of nodes at level m as C_m. For convenience we call this level of the tree the *working level*, since we find medoids by considering primarily nodes in this level. The separation property of the cover tree ensures that nodes in C_m spread out properly. We can view data under each subtree as a small cluster, whose centroid and density are maintained in the root of the subtree. In most cases, m does not equal k. The general approach in the literature is to group C_m into k groups first, and then find the medoid for each cluster. Usually, k *seeds* are first selected among the nodes, and the rest of the nodes are assigned to the respective closest seed. k-medoid clustering is NP-hard [40], so we usually try to

achieve a good local minimum in terms of distance cost from data points to their medoids. There are two existing seeding methods:

- **Random**. We can randomly choose k nodes to be the seeds. This is the simplest method with the lowest cost.
- **Space-filling curves**. Hilbert space-filling curve has been shown to preserve the locality of multidimensional objects when they are mapped to linear space [71], a property which Mouratidis et al. [74] exploited in their R-tree based clustering technique. We can apply the same idea in the cover tree. Nodes in C_m could be sorted by Hilbert values, and k seeds chosen evenly in the sorted list of nodes.

Seeds that are not properly chosen may lead the algorithms to a local minimum with high cost. In this chapter, we use information provided by the cover tree to choose seeds in a better way than the above techniques. Level $m-1$ of the cover tree, which contains less than k nodes, provides hints for seeds because of the clustering property of the tree. As usual, we denote nodes at level $m-1$ as set C_{m-1}. Intuitively, nodes in C_m that share a common parent in C_{m-1} form a small cluster themselves. When we choose seeds, we should avoid choosing two seeds in one such cluster. Since C_{m-1} contains fewer than k nodes, we will not have enough seeds if we simply choose one node from all that share a parent. As introduced in Sec. 3.3.1, nodes in C_{m-1} may have different maximum distance to their descendant. As a heuristic, we choose more seeds from children of a node whose descendants span a larger area. Meanwhile, nodes with a relatively small number of descendants should have low priority in becoming a seed, since the possible contribution to the distance cost is small. The contribution of a subtree to the distance cost is proportional to the product of the density and span. We denote this special value as the *weight* of a node. Based on this observation, we use a priority queue to choose seeds as follows.

The key of the priority queue is the weight of a node. Initially all non-naive nodes in C_{m-1} are pushed to the queue. We pop the head node from the queue and fetch all its children. We first make sure the queue has k nodes by adding children of the node with largest weight. Afterwards, if any child has a larger weight than the minimum weight of all nodes in the queue, we push it to the queue. We repeat this process until no more children can be pushed into the queue. The first k nodes in the queue are our seeds. This procedure, `CoverTreeSeeding()`, is shown in Algorithm 1.

Once the seeds are chosen, the rest of the nodes are assigned to their respective closest seed to form k initial clusters. We can obtain the centroids of each cluster by computing the geometric center of all nodes from their density and centroid. Using each centroid as input, we can find the corresponding medoid with a nearest neighbor query, which is efficiently supported by the cover tree. For each final medoid o, we call nodes in the working level that are closest to o as its *CloseSet*. In the future, if the user adds a selection condition and removes a large portion of the CloseSet, the corresponding medoid may have to be eliminated. More details on how the nodes in the CloseSet affects the existence of the medoid are in Sec. 3.4.2.

Optionally, we can try to improve the clusters before computing the final medoids. In the literature [72, 65], usually a repeated updating process is carried out: the centroid of each cluster is updated; nodes are re-assigned to the updated centroids. This process repeats until stable centroids are found. In this process, we can take into account the weight of each node, similar to [65]. This procedure is outline in Algorithm 2. As another refinement step, we can use cover-tree directed swaps. Literature [76] suggests that we can swap a medoid with other nodes and see if this leads to a lower cost. Usually it is the step that computes the cost that is expensive. We have at our disposal a formula that can estimate the distance cost, as described in Sec. 3.3.2. Instead of swapping with a random node, we can swap with nodes that was assigned to the closest neighbor medoid. Both measures significantly

Algorithm 1 Cover Tree-based Seeding Algorithm

Input: S: list of nodes in level m of the cover tree
Input: T: list of nodes in level m-1 of the cover tree
Input: k: number of medoids to compute
Input: Q: priority queue for nodes with key being the weight of a node
Output: O: list of seeds

```
minWeight = 0 {maintains the minimum weight of all nodes in Q}
for node t in T do
  if t is a naive node or leaf node then
    T = T − t
  else
    Insert(Q, t)
    if weight(t) < minWeight then
      minWeight = weight(t)
    end if
  end if
end for
repeat
  n = ExtractMax(Q)
  boolean STOP = TRUE
  if Size(Q) < k then
    add all children of n to Q
    update minWeight to smallest weight values seen
    STOP = FALSE
  else
    for each child node c of n do
      if weight(c) > minWeight then
        Insert(Q, c)
        STOP = FALSE
      end if
    end for
  end if
until STOP
O = Exact the first k nodes from Q
```

cut the computational cost. The details are omitted here due to limited available space.

Algorithm 2 Compute Medoids

Input: S: list of nodes in level m of the cover tree
Input: L: list of seeds
Input: k: number of medoids to compute
Output: M: list of medoids

```
for node s in list S do
  assign s to the seed in L whose centroid is closest {forming the initial clusters}
end for{denote the initial clusters as C}
repeat
  for c_i in C, i ∈ [1, k] do
    m_i = the node in c_i closest to geometric center of c_i
  end for
  for node s in list S do
    assign s to closest medoid in M
  end for
until no change in M
```

3.4 Query Refinement

In practical dataset browsing and searching scenarios, users often find it necessary to add additional filtering conditions or remove some attributes, often based on what they see from the data. In interactive querying, the time to deliver results to the user is most critical. Expensive re-computation that causes much delay (e.g., seconds) for the user severely damages the usability of the system. In this section, we show how our system can dynamically change the representatives according to the new conditions with minimal cost.

3.4.1 Zoom-in on Representative

When the user sees an interesting tuple from the list of representatives, she can click on the tuple to see more similar items. This operation can be efficiently supported using the cover tree structure. Recall that during the initial medoid generation

phase, every final medoid is associated with a CloseSet of nodes in the working level. Those nodes are the set of nodes that is closest to the medoid (relative to all other medoids). Once a medoid s is chosen by the user, we should generate more representatives around s. We proceed as follows. We fetch all nodes in the CloseSet of s, and descend the cover tree to fetch all their children and store them in a list L. This should give us a sample of the nodes/data around medoid s. We treat nodes in list L as the nodes in our new working level. We can run the same algorithm in Sec. 3.3.3 on nodes in L to obtain a new set of medoids.

3.4.2 Selection

When a user applies a selection condition, nodes in the working level are very likely to change. As mentioned in Sec. 3.3.3, existing representatives (medoids) will be eliminated if their CloseSet of nodes are removed by the selection condition. In this section, we detail how to effectively find new medoids when a selection is applied.

First we discuss the effect of a selection condition on each node in the working level. We start with this step because of the procedure we use to generate the medoids. Since the user queries a single table, we can consider a selection condition as a line (in the 2D case) or hyperplane (in 3D or higher dimensionality) in the data universe. For simplicity we now discuss only the 2D case, but high dimensional cases are easy to generalize to. For example, if we use Mileage as x-axis and Price as y-axis in 2D space, adding the selection condition "Price < 12000" removes all data points that are above the line $y = 12000$. Recall that for each node in the cover tree in level i, all its children fall in a circle with radius $D(i)$, and all its descendants are in the circle with radius $2 \times D(i)$ (span). Thus we can determine the impact of a new query condition on a node and its subtree by considering the relationship between the query line and the span of the node. For each node and its subtree can be classified into one of the following categories: 1) completely invalidated, 2)

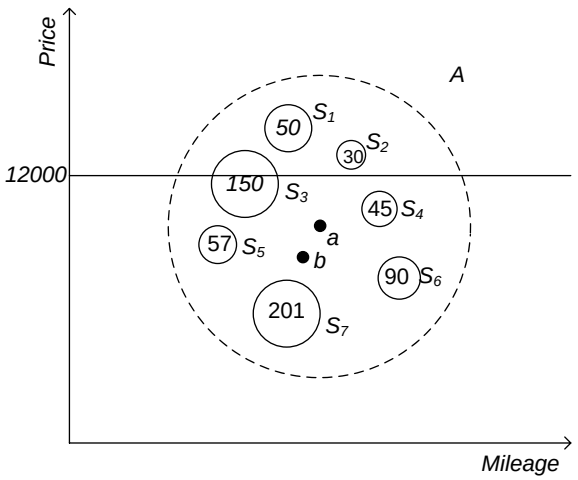


Figure 3.7: Effect of Selection on a Node

partially invalidated, or 3) completely valid. Category 1 nodes are removed from consideration and category 3 nodes stay intact.

For a category 2 node, once a selection condition is applied and a significant portion of the possible region that a node's descendants can span becomes invalid, the original data point is no longer a suitable approximation of the center of the subtree. The span value of the node is also inaccurate. The example in Fig. 3.7 shows a category 2 node, node A. The point associated with A is denoted as a, and it is located in the center of the largest circle, which is the span of A. Assume that we have a selection condition "price $<$ 12000 ". For each child s_1 to s_7, the radius of the respective circle denotes the span, and the numerical value denotes the density. After the selection, child nodes s_1 and s_2 are invalid, and s_3 is partially valid.

For partially valid nodes, we use their children to approximate the geometric center of the subtree. Specifically, we treat each child as a point with weight, and calculate the geometric center as the weighted average of all children. We also update the span using the child who is the farthest from the geometric center. Continue with the example in Fig. 3.7. By averaging over all valid children, we obtain point b as the estimated geometric center of all valid points of the subtree. Now node s_6 is

the farthest child from point b. Denote the span of s_6 as $s_6.span$, and the distance from s_6 to b is d. The new span is estimated as the sum of d and $s_6.span$. However, there is a recursion here, since the children can also be partially valid (for example, node s_3). When this happens, we estimate the *valid percentage* of the children as follows. For child node s, in 2D case, we calculate the area around s within distance $s.span$, and calculate the percentage that is still valid under the selection condition. This can be easily extended to higher dimensions. We take into this valid percentage by multiplying it with the node's weight.

After applying the selection condition, if there are less than k valid or partially valid nodes in the working level, we descend the cover tree until a level that has more than k nodes. On the other hand, if we still have more than k nodes in the working level, we can work on the nodes that remain or descend the tree to fetch new nodes. Next, we can re-run the medoid generation algorithm in Sect. 3.3.3 over the new set of nodes obtained from procedures detailed previously. This gives us a set of updated medoids.

3.4.3 Projection

We assume the user removes one attribute at a time, which is a reasonable assumption in interactive querying. There is usually some "think time" between two consecutive user actions. Our goal is to refresh the representative without incurring much additional waiting for the user.

Once an attribute is removed, the cover tree index is no longer a valid reflection of the actual distance among data points. Thus the brute-force approach is to re-construct a new cover tree without the attribute and re-compute the medoids. We want to do better than that. Our observations is that although the pair-wise distance between the samples may change dramatically after removing the attribute from consideration, the samples should still represent the data relatively well. Thus

we can still use the cover tree as a sampling tool - we sample the data at a level in the cover-tree regardless of the removed attribute.

Our approach is to use the same set of nodes in the working level we used to generate the previous medoids. We remove the dimension chosen by the user. These nodes serve as our primary samples of data. Since the cost is very low to re-run the medoid generation algorithm once we have the seeds, the key is to find a good set of seeds. Using the cover tree as direction is no longer viable: after removing a dimension, nodes that are previously far away can become very close. Also, the weight and span are less accurate in the new distance measure, which may severely affect the quality of generated seeds. So we use Hilbert sort re-order all nodes, and find seeds as outlined in Sec. 3.3.3. The rest is the same as described in Sec. 3.3.3.

3.5 Implementation and Experiments

3.5.1 System Architecture

The architecture of MusiqLens is shown in Fig. 3.8. When a query is initially sent from the client user interface to the DBMS, query results are fed to MusiqLens, which interacts with the client in this query session. MusiqLens then builds a cover tree index on the query results. This step can be done very efficiently through cover tree's construction algorithm. One of the features of cover tree is that it can be constructed efficiently in an online fashion. In our experiment, the index for a dataset comprising 130k points in 2D space is built in 0.7 seconds on an Intel Pentium Dual Core 2.8GHz machine with 4GB DDR2 memory. It takes PostgreSQL server 2.7 seconds to output the same set of data (we used a selection without any predicate).

Beside the indexer, the core of MusiqLens contains three other parts: the k-medoid generator, which generates the initial medoids after the user sends a new query to the database; the zooming operator, which is responsible for generating new

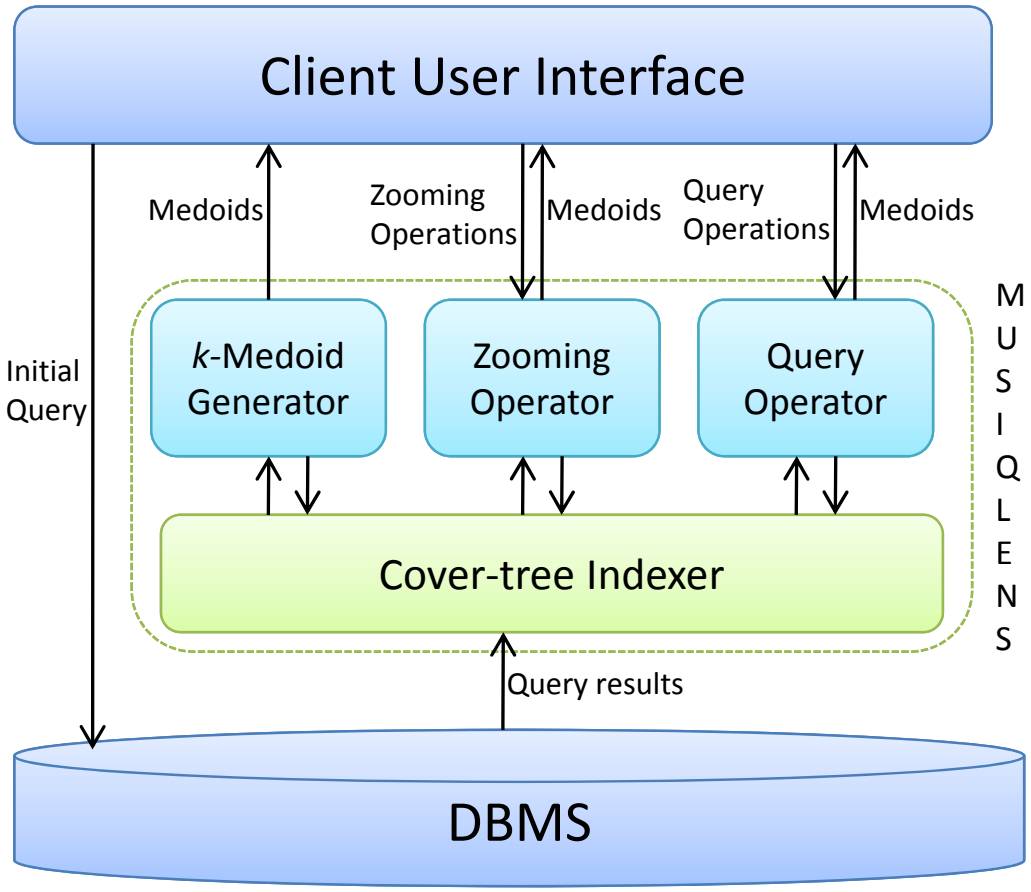


Figure 3.8: MusiqLens System Architecture

representatives after user performs a zooming operation; and the query operator, which dynamically adjusts the medoids according to user's new query conditions. MusiqLens can be implemented as a module in a DBMS or a layer between the client and the DBMS. The client interface we built is based on SheetMusiq [67], which is a spreadsheet direct manipulation interface for querying a database.

3.5.2 Experimental Results

The experiments are divided into two parts. First, we want to show that the cover tree based clustering algorithm generates high quality medoids with low time cost. For this we compare our algorithm with R-tree based algorithm in [74], which is the most related work and state-of-the-art. Second, we show that our query-adaptation

algorithms work effectively at low time cost and yet do not compromise in quality. We compare algorithms for selection and projection with computing the medoids from scratch.

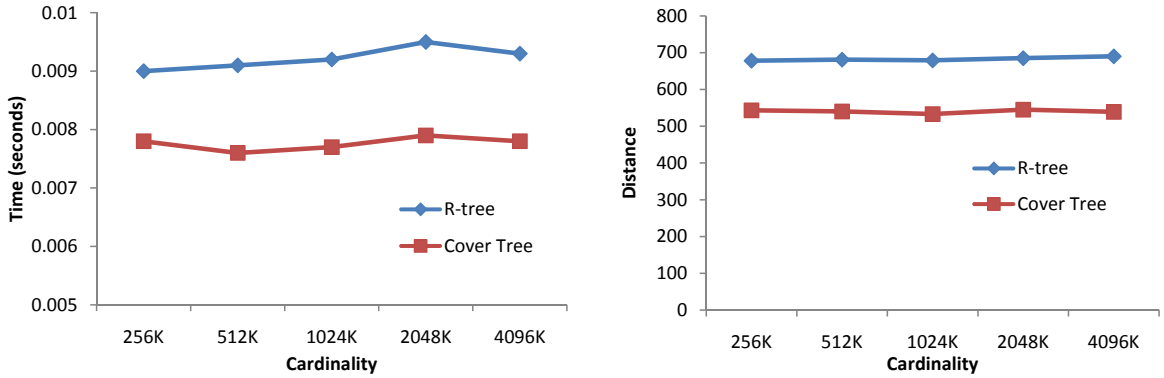
Comparison with R-tree Based Methods

We use both real and synthetic data sets for this comparison. We use the LA data set from the R-Tree Portal (http://www.rtreeportal.org). It contains $130k$ rectangles and we take the center of each rectangle as a data point. We generate synthetic data containing 2-dimensional data points that follow a Zipf distribution with parameter $\alpha = 0.8$. We use 5 sets of data of increasing cardinality: 256K, 512K, 1M, 2M, and 4M. For all data sets, we normalize each dimension of the data to the range of [0, 10000]. We also vary the value of k, the number of medoids to compute. Comparing with R-tree based algorithms, we measure two metrics:

- **Time**: time to compute the medoids
- **Distance**: the average Euclidean distance from data points to their respective medoid.

For convenience, we refer to the cover tree based method as *CTM*, and R-tree based method as *RTM*. In the figures below, legend for CTM is "Cover Tree", and that for RTM is "R-tree".

Fig. 3.9 shows the time and distance cost with synthetic data sets with growing cardinality, with a fixed value of k at 32. We can see that CTM (Cover Tree based Method) consistently outperforms RTM (R-Tree based Method) in both metrics. Fig. 3.9 (a) shows that both methods are stable and scalable in time with a growing size of the data set. The primary reason is that, once the R-tree or cover tree index is built, the cost depends more on the value of k (which we will see soon) than on the size of the data. Both algorithms fetch the upper levels of the data structure. For a



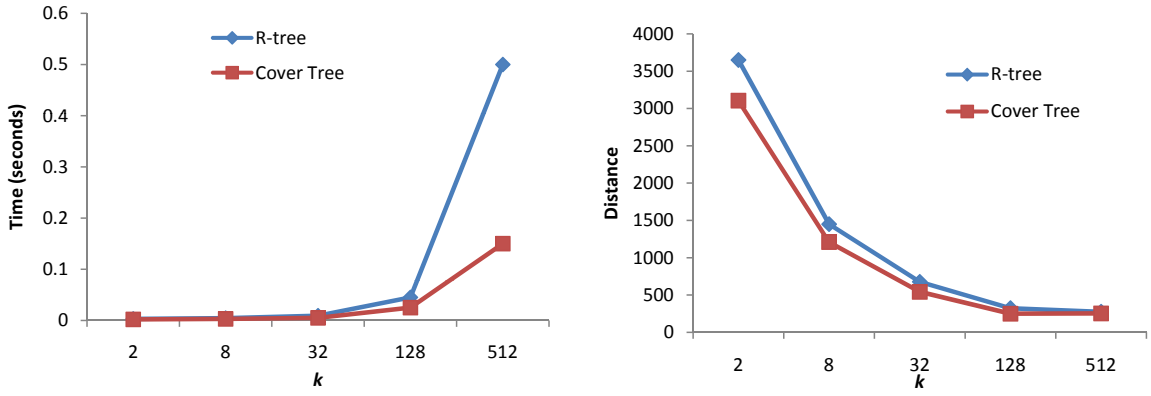
(a) Time (b) Distance

Figure 3.9: Synthetic Dataset of Various Sizes

fixed value of k, the number of nodes that need to be read from the disk does not vary significantly with the size of the data. The reason CTM is faster is it brings less data from the disk. Each node of the cover tree is also a data point, and it is smaller than an R-tree node. Fig. 3.9(b) shows that the distance cost stays stable as the data size varies. This is expected since the medoids are found mainly on the upper levels of the data structure. It also shows that cover tree method produces medoids with much smaller distance cost. In sum, cover tree based method generates better medoids at a lower cost, regardless of the size of the data.

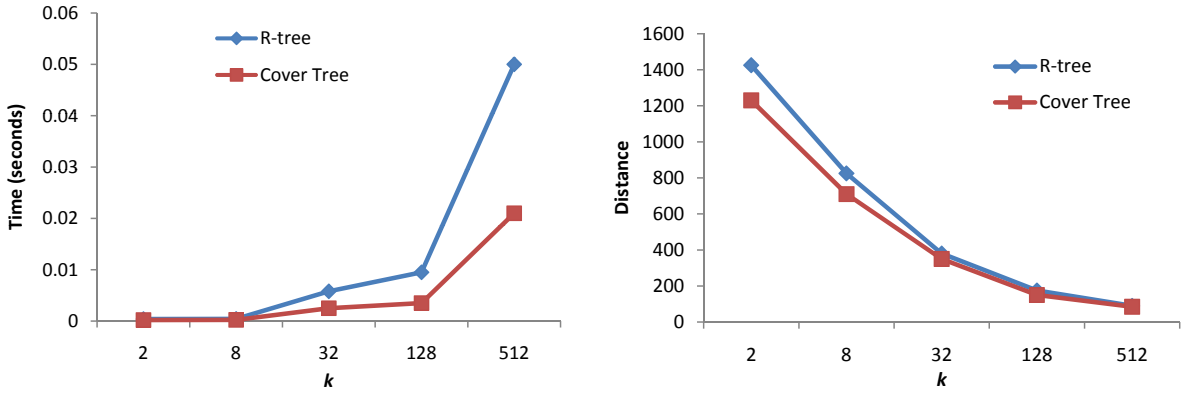
Fig. 3.10 shows the trend of time and distance cost with the growth of the number of medoids to compute (k), for a synthetic data set with cardinality of $1024k$. We can see that distance from CTM is consistently lower than RTM while using almost half of the time, affirming the conclusion before.

Fig. 3.11 shows results on the real data set, LA, with various values of k. The trend in time is the same as we observed in synthetic data, where CTM outperforms RTM by a large margin. CTM also produces better quality medoids than RTM.



(a) Time (b) Distance

Figure 3.10: Synthetic Dataset of Various k Values



(a) Time (b) Distance

Figure 3.11: Results for Real Dataset

Query Refinement

Having established that cover tree based method is superior than the R-tree based method in generating the initial representatives, we want to see if user issued refinement can be efficiently processed. Since we have no competitor in this incremental re-computation of medoids, we use the absolute running time as the measurement metric. For quality of results, we compare against re-computing the medoids from scratch. For the latter case, when a user issues a selection condition or removes an attribute, we re-build a new cover tree on the data after the query. Thus we are comparing the incremental re-computation algorithms with expensive

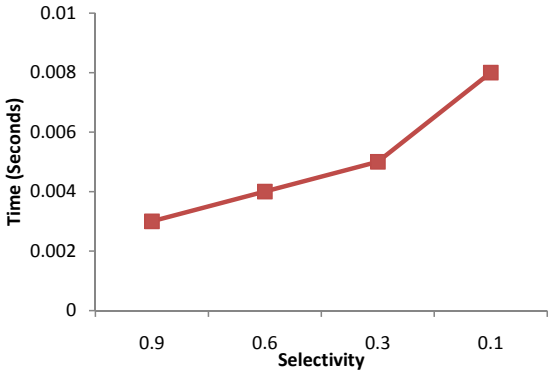
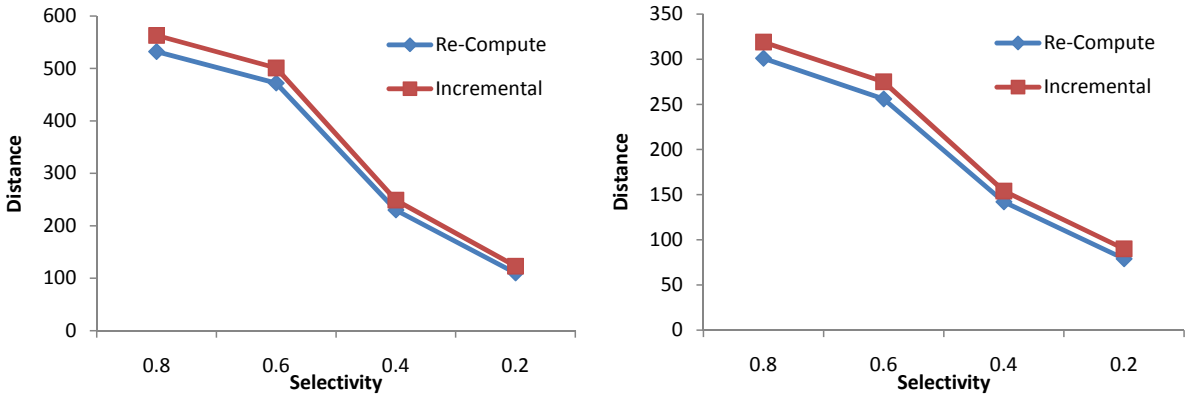


Figure 3.12: Time for Selection

fresh re-computation.

Selection. We apply selection conditions of various selectivity on a synthetic data set of cardinality $1024k$. The selectivity values are 0.8, 0.6, 0.4, and 0.2. We use selection conditions such as "$x < 4500$" to remove a portion of the data. Since re-constructing a cover tree takes much more time than computing the medoids, there is little meaning to show the time difference. The running time for our algorithm is show in Fig. 3.12. Since the nodes at the working level can be already cached in the memory when computing previous medoids, we do not need to fetch them from the disk. Possible I/O is still necessary if a large portion of the nodes are disqualified and we need to descend the tree to fetch lower level nodes. The time for selection is well below 0.01 seconds, which is orders of magnitude smaller than re-building the index. In Fig. 3.13 we show the comparison in distance cost, for both synthetic and real data. The synthetic data is of cardinality $1024k$. We also use the LA data set. We can see that incremental re-computation of medoids using the proposed algorithm (legend "Incremental") provides comparable quality of medoids. There is little, if noticeable at all, difference in terms of distance cost. The time saved is with orders of magnitude, with no compromise of result quality.

Projection. We take the same approach as for selection - compare the



(a) Synthetic Data (b) Real Data

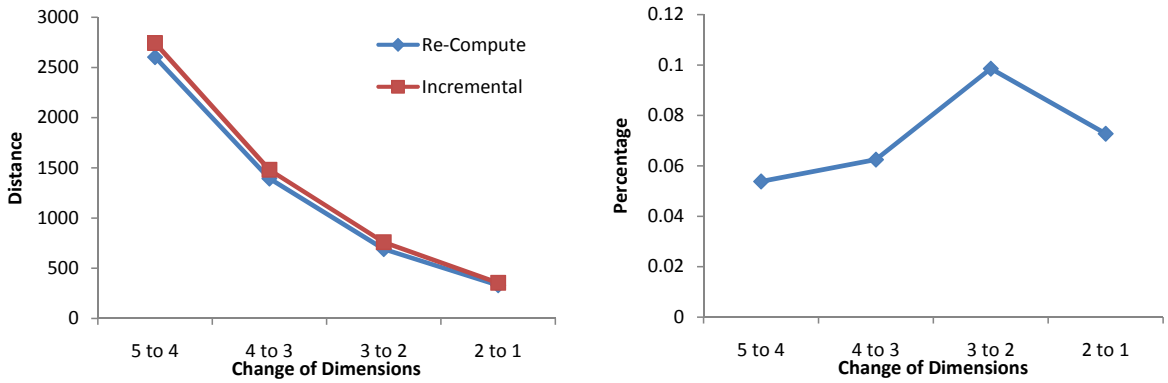
Figure 3.13: Selection Performance

incremental algorithm with re-computing from scratch. The number of medoids to compute is 32.

We assume that the user projects one attribute at a time. We start with 4 different artificial data sets, each of dimension 5, 4, 3, and 2, respectively. We then remove one dimension from each data set and compare incremental approach with re-computing from scratch. Fig. 3.14 shows the result. The left figure shows the comparison of absolute distance cost, while the right shows the percentage of increased distance cost using the incremental approach. We can see that the percentage of result compromise is consistently below 10%. In interactive querying, users may not notice the 10% of difference in distance cost, but they will surely notice the difference in time between milliseconds and seconds. Thus we think it is still valuable to save the time of re-building the index and re-computing the medoids at the cost of small deterioration of result quality. Re-computation is the last resort, when the user removes a significant number of dimensions.

3.5.3 Fast Representative Choice

Cover tree construction is moderately fast – under one second for a moderate size data set (LA) with $131k$ tuples. Fig. 3.15 shows the time to build a cover tree index



(a) Absolute Distance Cost (b) Percentage of Increase

Figure 3.14: Projection Performance on Single Dimension

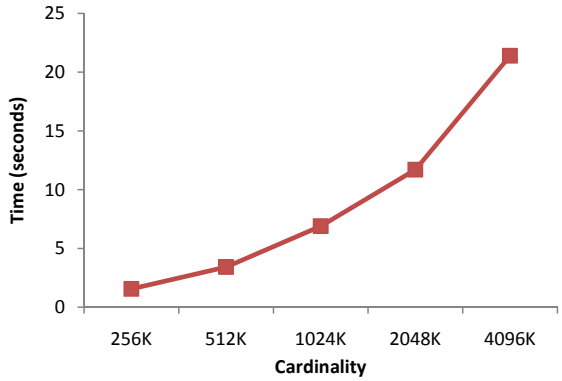


Figure 3.15: Cover Tree Building Time on Synthetic Data Sets

for synthetic data sets, with cardinality from 256k up to 4M. We can see that the construction time scales up gracefully. While it is not too expensive from an absolute time perspective, even one second may be too much time to add to how long a user waits to see results.

The encouraging results presented above for incremental computation offer a simple way around this. We pre-compute the cover tree for the data set – maintenance of this structure is comparable to the cost of incremental index maintenance. Then *every* query against the data set, including the very first, can be treated as a "refinement" of a base query that returns the whole data set. With this, we have an overhead of only 10s of milliseconds per query, a level that is quite

affordable.

3.6 Related Work

Various methods have been proposed for K-medoid clustering. PAM (Partitioning Around Medoids) [58] starts with k randomly selected objects and iteratively improves upon them until the quality of clustering (measured by the average distance to medoids) converges. In each iteration, PAM considers exchanging any of the current k-medoids with any other data point and choosess the swap that leads to the best improvement. This is prohibitively expensive to run on large data sets. CLARA (Clustering LARge Applications) [58] attempts to reduce the cost by first randomly sampling the data set and then performing PAM on the samples. In order to ensure the samples are sufficiently random, CLARA draws multiple (e.g., 5) sets of samples and uses the best output as the final result. However, in order to estimate the quality of the result, CLARA still needs to compute the distance from all data points to the candidate medoids. This would require scanning the whole data set at least once in each iteration, which is again inefficient for large data sets. Ng and Han's CLARANS (Clustering Large Applications based on RANdomized Search) [76], instead of considering all possible swaps like PAM, randomizes the search and greatly reduces the cost. CLARANS is a main-memory clustering technique, and it also requires scanning the whole data set. For MusiqLens framework, main-memory methods will not suffice since we aim at large data sets.

Some other work uses disk-based indices to speed up the clustering. FOR (focusing on representatives) [34, 36] and TPAQ (tree-based partitioning querying) [73, 74] both assumes that the data set is indexed by an R-tree. FOR takes the most centrally located object out of each leaf node and runs CLARANS on the samples. This means that FOR has to read the entire data set to obtain the samples. TPAQ starts from the root of the R-tree until it reaches a level where there are more than k

nodes. It then uses Hilbert curve to sort the nodes and evenly chooses k *seed* nodes out of all nodes in that level. Other nodes are signed to their closest seeds and thus forming small clusters. The geometric center of each cluster is estimated and used to perform a nearest-neighbor (NN) query to fetch the closest point in the data set. The result of each NN query is the medoid of the corresponding cluster. Experiments in [74] shows that TPAQ improves both result quality and computational cost over FOR. FOR and TPAQ are advantageous compared to main-memory methods for the ability to handle large data sets, which is the first requirement of MusiqLens. The second requirement, query adaptation, however, remains unsatisfied by either FOR or TPAQ. For interactive browsing, users may not be so critical on the quality of the medoids, but the lack of interactive refinement and navigation would make the system unusable.

Pan et al [80] proposed an information-theoretic approach for finding representatives from large set of categorical data. They treat each data element as a set of features and obtain a data distribution from the presence of features. It is unclear how to extend the proposed techniques to numerical data, which is the focus of this chapter.

Recent work by Li et al [65] proposed generalized group-by and order-by for SQL. Their grid based method is for clustering only, without actually finding the medoid for each cluster. They use a separate ordering function to choose which data point to output for each cluster. To apply techniques in [65] to MusiqLens framework, we would have to find a center for each cluster. One of the methods is to find the point that is closest to the geometric center of the cluster. This would require an additional scan of data or an additional index structure. In addition, [65] does not support zooming, which is an essential feature of MusiqLens. DataScope [109] provides an interface for zooming in and out of a data set by ranking. Common built-in ranking functions are provided (e.g., ranking by the number of publications of authors).

The system supports browsing but no searching nor adaption to multiple searching criteria.

Computing k-medoid is related to the problem of clustering. The goal of clustering is to find the naturally close groups in a set of data, where the number of clusters is not known or given a priori. Many efficient techniques have been developed for clustering, for example, BIRCH [113], DBSCAN [35], and CURE [44] (see [110] for a comprehensive survey). The difference between the two problems is, one is to find the natural groupings in the data, and the other is to optimize a distance cost. The cluster centers that naturally exist in the data may not be the best k-medoids, which is shown in [74]. Projection adaptation (Sec. 3.4.3) is related to the problem of *subspace clustering*, which has been extensively studied [12, 81]. Subspace clustering attempts to find clusters in a data set by selection the most relevant dimensions for each cluster separately. In our case, the set of dimensions to consider are dynamically determined by the user.

Another related problem that has attracted increasing attention is query result diversification [106]. Both [106] and this chapter attempt to provide better usability when the number of tuples that can be shown are limited. We believe the two are different solutions under different situations. Diverse results needs ordering of attributes from experts while we do not.

3.7 Conclusion

In this chapter, we propose the MusiqLens framework for interactive data querying and browsing, where we solve the "Many-Answers Problem" by showing users representatives of a data set. Our goals are: 1) to find the representatives efficiently, and 2) adapt efficiently when users refine the query. We start by identifying what is a good set of representatives by conducting a user study. Results show consistently that k-medoids are the best amongst seven options. Towards the first

goal, we devised cover tree based algorithms for efficiently computing the medoids. Experiments on both real and synthetic data sets shows that our algorithm is superior over our competitor in both time and distance cost. Towards the second goal, we proposed algorithms to efficiently re-generate the representatives when users add selection condition, remove attributes, or zoom-in on frequentatives.

A larger context for the work presented in this chapter is the concept of direct manipulation [67, 57, 94], where a user always has in hand data representing some partial result of a search. Thus, someone looking for used cars is shown a few representative cars upon entering the web site, and incrementally refines the result set in multiple steps. The work presented in this chapter provides a means for databases to show meaningful results to users without requiring additional effort on their part.

CHAPTER IV

QUERY REFINEMENT: A PROVENANCE-BASED FRAMEWORK

4.1 Introduction

Information extraction — the process of deriving structured information from unstructured text — is an important aspect of many enterprise applications, including semantic search, business intelligence over unstructured data, and data mashups. The structured data that information extraction systems produce often feed directly into important business processes. For example, an application that extracts person names from email messages might load this name information into a search index for electronic legal discovery; or it may use the name to retrieve employee data for help desk problem determination. Because the outputs of information extraction are so closely tied to these processes, it is essential that the extracted information have very high precision and recall; that is, the system must produce very few false positive or false negative results.

Most information extraction systems use rules to define important patterns in the text. For example, a system to identify person names in unstructured text would typically contain a number of rules like the rule in Figure 4.1. The example in the figure is written in English for clarity; an information extraction would typically use a rule language like JAPE [31], AQL [61], or XLog [90, 18].

In some systems, the outputs of these rules may feed directly into applications [33,

If a match of a dictionary of common first names occurs in the text, followed immediately by a capitalized word, mark the two words as a "candidate person name".

Figure 4.1: An example information extraction rule, in English.

64, 47, 31]. Other systems use rules as the feature extraction stage of various machine learning algorithms (as in [39, 63, 82]). In either case, it is important for the rules to produce very accurate output, as downstream processing tends to be highly sensitive to the quality of the results that the rules produce.

Developing a highly accurate set of extraction rules is difficult. Standard practice is for the developer to go through a complex iterative process: First, build an initial set of rules; then run the rules over a set of test documents and identify incorrect results; then examine the rules and determine refinements that can be made to the rule sets to remove incorrect results; and finally repeat the process. Of these steps, the task of identifying rule refinements is by far the most time-consuming. An extractor can easily have hundreds of rules, and the interactions between these rules can be very complex. When changing rules to remove a given incorrect result, the developer must be careful to minimize the effects on existing correct results. In our experience building information extraction rules for multiple enterprise software products, we found that identifying possible changes for a single false positive result can take hours.

In the field of data provenance, techniques have been developed to trace the lineage of a tuple in a database through a sequence of operators. This lineage also encodes the relationships between source and intermediate result tuples and the final result. In this chapter, we bring these techniques to bear on the problem of information extraction rule refinement. Intuitively, given a false positive result of information extraction, we can trace its lineage back to the source to understand exactly why it is in the result. Based on this information, we can determine what possible changes can be made to one or more operators along the way to eliminate

the false positive, without eliminating true positives. Actually realizing this vision, the central contribution of this chapter, requires addressing some challenges, as outlined in Sec. 4.4.

Most information extraction rules can be translated into relational algebra operations. Over such an operator graph, provenance-based analysis, developed in Sec. 4.5, produces a set of proposed rule changes in the form of "remove tuple t from the output of operator O". We refer to this class of rule changes as *high-level changes*. To remove a "problem" tuple from the output of a rule, the rule developer needs to know how to modify the extraction primitives that make up the rule. We call such changes *low-level changes*. (Extraction primitives include regular expressions and filtering predicates like "is followed by"). These modifications may in turn result in the removal of additional tuples besides the "problem" tuple, and the developer needs to consider these side-effects in evaluating potential rule changes, while simultaneously keeping the rules as simple and easy to maintain as possible.

In Sec. 4.6, we develop a framework for enumerating the low-level changes that correspond to a given set of high-level changes. We also develop efficient algorithms for computing the additional side-effects of each proposed low-level change. Using this information, we then rank low-level changes according to how well they remove false positives without affecting existing correct answers or complicating the rule set. This ranked list of low-level changes is then presented to the rule developer.

We have embodied these ideas in a software system that automates the rule refinement process and implemented it in the SystemT information extraction system[1] [30, 61, 85]. Given a set of rules, a set of false positive results that the rules produce, and a set of correct results, our system automatically identifies candidate rule changes that would eliminate the false positives. The system then computes

[1]Available for download at http://alphaworks.ibm.com/tech/systemt.

the overall effects of these changes on result quality and produces a ranked list of suggested changes that are presented to the user.

The system can be also used in fully automated mode, where the highest ranked change is automatically applied in each iteration.

We have extensively evaluated the system, and present representative results to demonstrate its effectiveness in Sec. 4.7.

We begin with a discussion of related work in Sec. 4.2, and set up the problem formally in Sec. 4.3.

4.2 Related Work

The field of **data provenance** studies the problem of explaining the existence of a tuple in the output of a query. A recent survey [29] overviews various provenance notions for explaining *why* a tuple is in the result, *where* it was copied from in the source database, and *how* it was generated by the query. It is the latter type of provenance, *how-provenance* [43], that is leveraged in our system to generate the set of high-level changes: place-holders in the rule set where a carefully crafted modification may result in eliminating one false positive from the output. However, this is only the first step of our approach. In a significant departure from previous work on data provenance, our system generates a ranked list of concrete rule modifications that remove false positives, while minimizing the effects on the rest of the results and the structure of the rule set.

Early work in **information extraction** produced a number of rule-based information extraction systems based on the formalism of cascading regular expression grammars. Examples include FRUMP [33], CIRCUS [64], and FASTUS [47]. The Common Pattern Specification Language [17] provided a standard way to express these grammars and served as the basis for other rule-based systems like JAPE [31] and AFsT [22]. In recent years the database community

has developed other rule languages with syntaxes based on SQL [100, 61, 51] and Datalog [90, 18]. The techniques that we describe in this chapter can be used to automate the rule refinement process across all these different classes of rule languages.

Other work has used machine learning to perform information extraction, and a variety of systems of different flavors have been developed, ranging from entity relation detection ([114]) to iterative IE (e.g., Snowball [11]) and open IE (e.g., TextRunner [111]). Researchers have employed a variety of techniques, including covering algorithms [97], conditional random fields [63, 82], support-vector machines [114], and mixtures of multiple learning models [39, 111]. The work that we describe in this chapter is complementary to this previous work. Our system employs a semi-automatic iterative process with a human in the loop, which represents a new area of the design space for information extraction systems. This design choice allows our system to handle highly complex rule structures and to leverage expert input. Whereas machine learning models are generally opaque to the user, the rules that our system produces can be understood and "debugged" by the rule developer.

Recently, [32] has shown how introducing transparency in a machine learning-based iterative IE system by recording each step of the execution enables the automatic refinement of the machine learning model via adjusting weights and removing problematic seed evidence. Our work differs from [32] in that we consider automatic refinement in the context of rule-based systems, and therefore our space of refinements is completely different.

In practice, information extraction systems that employ machine learning generally use rules to extract basic features that serve as the input, and our techniques can be used to assist in the process of developing these rules. Additional previous work has used machine learning for extraction subtasks like creating dictionaries [86] and character-level regular expressions [66]. These techniques

are complementary to the work we describe in this chapter. In particular, our work provides a mechanism for "plugging in" these algorithms as low-level change generation modules.

Finally, [89] describes an approach for refining an extraction program by posing a series of questions to the user. Each question asks for additional information about a specific feature of the desired extracted data. The features considered are pre-defined. For each question, the corresponding selection predicate is added to the extraction program. Our work differs fundamentally from the approach of [89] in that it automatically suggests fully-specified rule refinements based on labeled extracted data, as opposed to asking the user to fill in the blanks in template questions. Furthermore, consider a much broader space of refinements ranging from adding/modifying selection/join predicates and dictionary extraction specifications, to adding subtraction sub-queries. To the best of our knowledge, ours is the first system for suggesting concrete rule refinements based on labeled extracted data.

4.3 Preliminaries

Different information extraction systems have different rule languages [31, 61, 90, 18]. However, most rule languages in common use share a large set of core functionality.

In this chapter, we use SQL for expressing information extraction rules in order to describe the theory behind our system in a way that is consistent with previous work on data provenance. Specifically, we use the SELECT - PROJECT - JOIN - UNION ALL - EXCEPT ALL subset of SQL. Note that UNION ALL and EXCEPT ALL are not duplicate-removing, as per the SQL standard [38].

Our use of SQL does not in any way preclude the application of our work to other rule languages. Figure 4.3 shows some examples of some rule languages referenced in recent work. The figure shows three different implementations of the

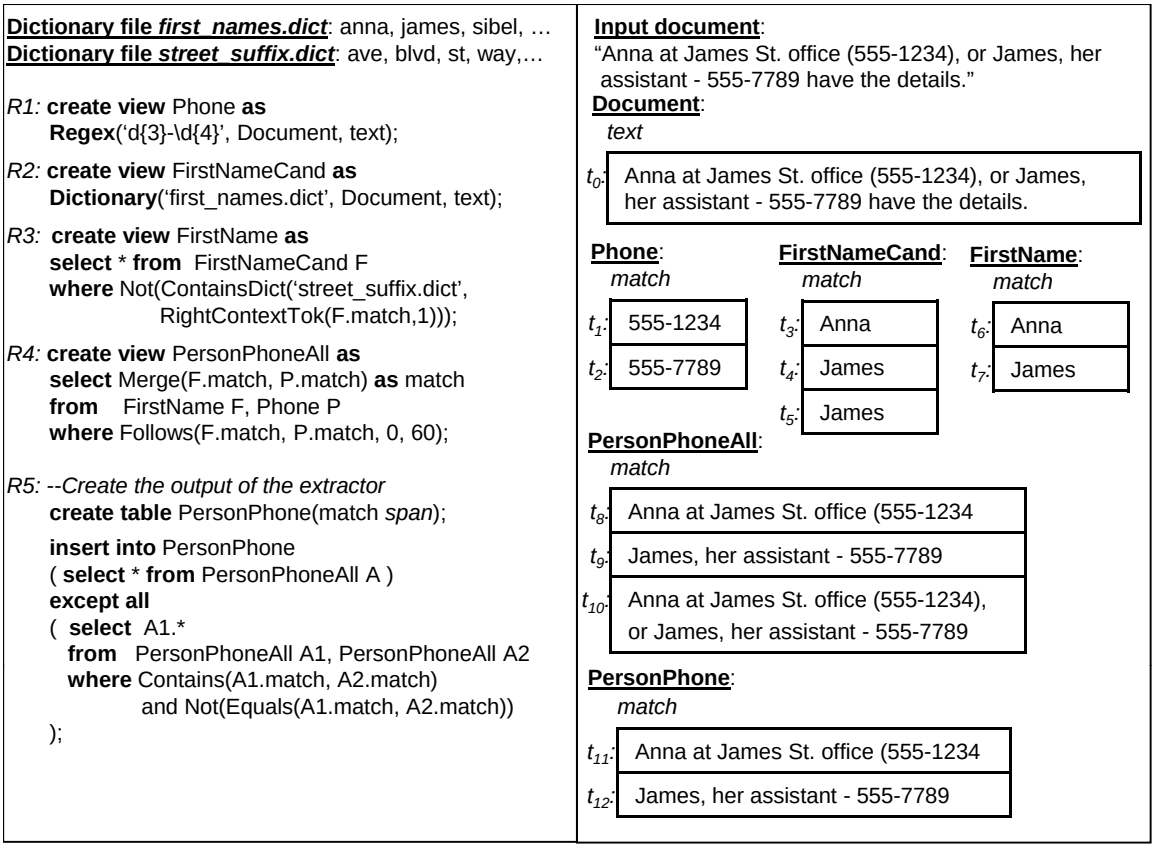


Figure 4.2: Example extraction program, input document D, and view instances created by the extraction program on D.

rule that we had described earlier in in Figure 4.1. Each implementation uses a different rule language, but all three generate the same output, except in certain corner cases. In general, information extraction rule languages often have very different syntaxes. There are also some differences between languages in terms of their overall expressive power [85]. However, most rule languages in common use share a large set of core functionality. These languages define the extractor as a set of rules with dependency relationships that can be used to construct a provenance graph for computing high-level changes. Rules are made up of atomic operations that can be modified, added, or deleted to create low-level changes. As such, the high-level/low-level change framework that we define in this chapter carries over easily to the rule languages in common use today.

JAPE [31] AQL [85] XLog [90, 18]

```
Rule: CandidatePersonName
 Priority: 1
   (
     { Lookup.kind == firstName }
     { Token.orthography == initialCaps }
   ):match
--> :match.kind = "CandidateName";
```

```
create view CandidatePersonName as
select CombineSpans(F.name, L.name) as name
from (extract dictionary FirstNameDict
        on D.text as name from Document D) F,
     (extract regex /[A-Z][a-z]+/
        on D.text as name from Document D) L
where FollowsTok(F.name, L.name, 0, 0)
consolidate on name;
```

```
CandidatePersonName(d, f, l) :-
  docs(d),
  firstNamesDict(fn),
  match(d, fn, f),
  match(d, "[A-Z][a-z]+", l),
  immBefore(f, l);
```

Figure 4.3: The rule from Figure 4.1, expressed in three different information extraction rule languages

4.3.1 Extensions to SQL

To make our examples easier to read, we augment the SQL language with shorthands for some basic information extraction primitives.

We add a new atomic data type called *span* for modeling data values extracted from the input document. A *span* is an ordered pair $\langle begin, end \rangle$ that identifies the region of an input document between the *begin* and *end* offsets. For clarity, we may sometimes identify a *span* using its string value in addition to the begin and end offsets, or we may simply drop the offsets when they are clear from the context. For example, to identify the region starting at offset 0 and ending at offset 5 in the document from Figure 4.2, we may use the notations $\langle 0, 5 \rangle$, or $\langle 0, 5 \rangle$: "Anna", or simply, "Anna".

We model the input document as a table called Document with a single attribute of type *span* named *text*. We also add several predicates, scalar functions, and table functions to SQL's standard set of built-in functions.

In the examples in this chapter, we augment the standard set of SQL functions with the following text-specific functions:

1. Predicates and scalar functions for manipulating spans, used for expressing *join* and *selection predicates*, and creating new values in the SELECT clause of a rule; and

2. Table functions for performing three crucial IE tasks: *regular expression matching*, *dictionary matching*, and *deduplication of overlapping spans*.

Figure 4.4 lists these text-specific additions, along with a brief description of each.

The ability to perform character-level *regular expression matching* is fundamental in any IE system, as many basic extraction tasks such as identifying phone numbers or IP addresses can be achieved using regular expressions. For our example rule in Figure 4.3, regular expression matching is appropiate for identifying capitalized words in the document, and is expressed, for instance, in AQL lines 5 – 6, and xLog line 5 in Figure 4.3.

For this purpose, we have added to our language the *Regex* table function (refer to Figure 4.4), which takes as input a regular expression, a relation name R, and an attribute of type span A of R, and computes an instance with a single span-typed attribute called *match* containing all matches of the given regular expression on the A values of all tuples in R.

A second fundamental IE functionality is *dictionary matching*: the ability to identify in an input document all occurrences of a given set of terms specified as entries in a dictionary file. Dictionary matching is useful in performing many basic extraction tasks such as identifying person salutations (e.g., "Mr", "Ms", "Dr"), or identifying occurrences of known first names (e.g., refer to Figure 4.3, line 4 of JAPE, lines 3–4 of AQL, and line 3 of xLog). The *Dictionary* table function serves this purpose in our language: it takes as input the name of a dictionary file, a relation name R, and an attribute of type span A of R, and computes an instance with a single span-typed field called *match* containing all occurrences of dictionary entries on the A values of all tuples in R.

A third component of information extraction rules is a toolkit of span operations. Table 4.4 lists the text-based scalar functions that our system uses to implement various operations over the *span* type. Note the distinction between scalar functions

Type	Format	Description
Predicate function	*Follows/FollowsTok*($span_1$,$span_2$,n_1,n_2)	Tests if $span_2$ follows $span_1$ within n_1 to n_2 characters, or tokens
	Contains/Contained/Equals($span_1$,$span_2$)	Tests if $span_1$ contains,is contained within, or is equal to $span_2$
	MatchesRegex/ContainsRegex(r, *span*)	Tests if *span* matches (contains a match for, resp.) regular expression r
	MatchesDict/ContainsDict(*dict*, *span*)	Tests if *span* matches (contains a match for, resp.) an entry of dictionary d
Scalar function	*Merge*($span_1$, $span_2$)	Returns the shortest span that completely covers both input spans
	Between($span_1$, $span_2$)	Returns the span between $span_1$ and $span_2$
	LeftContext/LeftContextTok(*span*, n)	Returns the span containing n chars/tokens immediately to the left of *span*
	RightContext/RightContextTok(*span*, n)	Returns the span containing n chars/tokens immediately to the right of *span*
Table function	*Regex*(r, R, A)	Returns all matches of regular expression r in all $R.A$ values.
	Dictionary(d, R, A)	Returns all matches of entries in dictionary d in all $R.A$ values.

Figure 4.4: Text-specific predicate, scalar, and table functions that we add to SQL for expressing the rules in this chapter.

that return a boolean value (e.g., *Follows*) and can be used as join predicates, and scalar functions that return non-boolean values (e.g., Merge), used as selection predicates, and to create new values in the SELECT clause of rules.

4.3.2 Example Rules

Figure 4.2 shows an example rule program, expressed in SQL, which extracts occurrences of person names and their phone numbers. We have divided the SQL into individual rules, labeled R_1 through R_5. Rules R_1 through R_4 define logical views, while rule R_5 materializes a table of extraction results.

Rule R_1 illustrates one of the shorthands that we add to SQL: the *Regex* table function. This function evaluates a regular expression over the text of one or

more input spans and returns a set of output spans that mark all matches of the expression. In the case of rule R_1, the regular expression phone numbers of the form $xxx - xxxx$.

Rule R_2 shows another addition to SQL: the *Dictionary* table function. Similar to the *Regex* table function, *Dictionary* identifies all occurrences of a given set of terms specified as entries in a dictionary file. For R_2, the dictionary file contains a list of common first names. The rule defines a single-column view *FirstNameDict* containing a *span* for each dictionary match in the document.

Rule R_3 uses a filtering dictionary that matches abbreviations for street names on the right context of names, to filter out first names that are street names, e.g., "James St.". The view definition uses two of the scalar functions that we add to SQL: *RightContextTok* and *ContainsDict*. *RightContextTok* takes a span and a positive integer n as input and returns the span consisting of the first n tokens to the right of the input span. The *ContainsDict* function, used here as a selection predicate, takes a dictionary file and a span and returns *true* if the span contains an entry from the dictionary file.

Rule R_4 identifies pairs of names and phone numbers that are between 0 and 60 characters apart in the input document. The view definition uses two of the scalar functions that we add to SQL: *Follows* and *Merge*. The *Follows* function, used here as a join predicate, takes two spans as arguments, along with a minimum and maximum character distance. This function returns *true* if the spans are within the specified distance of each other in the text. The *Merge* function takes a pair of spans as input and returns a span that exactly contains both input spans. The `select` clause in R_5 uses *Merge* to define a span that runs from the beginning of each name to the end of the corresponding phone number.

Finally, R_5 materializes the table PersonPhone, which constitutes the output of our extractor. It uses an EXCEPT ALL clause to filter out candidate name–phone spans

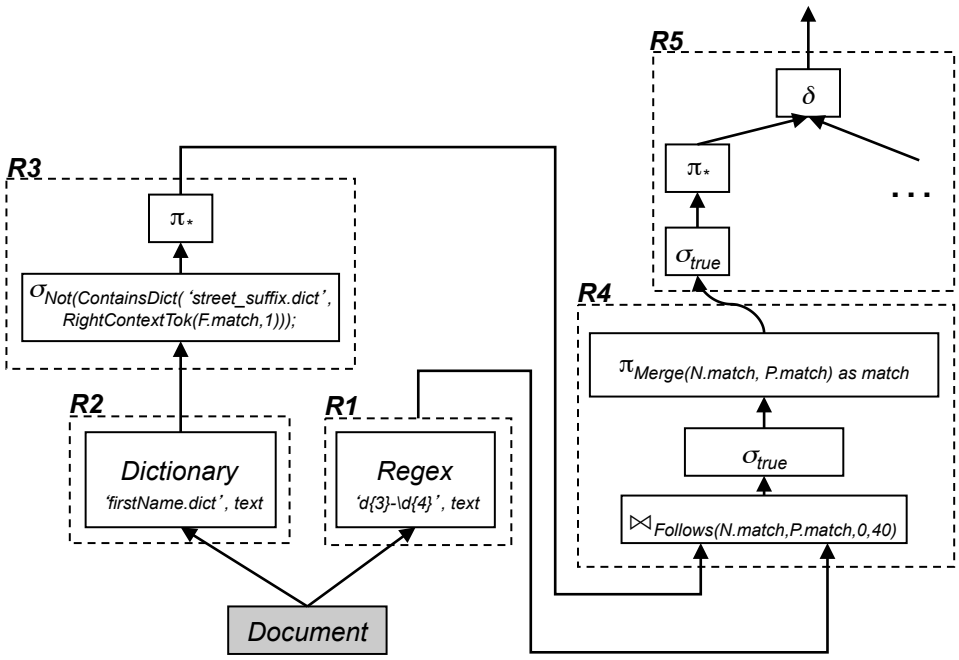


Figure 4.5: Canonical representation of rules in Figure 4.2.

strictly containing another candidate name–phone span. The join predicate of the second operand of the EXCEPT ALL clause illustrates two other text-based scalar functions: *Equals*, which checks if two spans are equal, and *Contains*, which tests span containment. Note that the false positive t_{10} in PersonPhoneAll that associates Anna with James' phone number is filtered out by R_5, since its span strictly contains other candidate name-phone spans (i.e., from t_8 and t_9).

4.3.3 Canonical Rule Representation

To simplify our subsequent discussions, we shall assume a canonical algebraic representation of extraction rules as trees of operators, such that for each rule, there is a direct one-to-one translation to this canonical representation and back. The canonical representation is very similar, if not identical for the SELECT - FROM - WHERE - UNION ALL - EXCEPT ALL subset of the language, to the representation of SQL statements in terms of relational operators. A rule in

the form "SELECT *attributes* FROM R_1, ..., R_n WHERE *join_predicates* AND *selection_predicates*" is represented in the usual way as the sequence of project – select – join operators shown below:

$$\pi_{attributes}(\sigma_{selection\_preds}(\bowtie_{join\_preds}(R_1, \ldots, R_n)))$$

When table functions like *Dictionary* and *Regex* appear in the FROM clause of a SELECT statement, we translate these table functions to operators by the same names.

Figure 4.5 illustrates the canonical representation of our example extractor from Figure 4.2, where the dashed rectangles indicate the correspondence between parts of the operator tree and rule statements. (The part corresponding to the second operand of the EXCEPT ALL clause in rule R_5 is omitted.) Note that when the WHERE clause of a rule does not contain any selection predicates (e.g., R_4), the condition in the select operator of the corresponding canonical representation is simply *true*.

4.4 Overall Framework

Given a set of examples in the output of an extractor, each labeled correct or incorrect by the user, our goal is to generate a ranked list of possible changes to the rules that result in eliminating the incorrect examples from the output, while minimizing the effects on the rest of the results, as well as the rules themselves. Our solution operates in two stages: High-level change generation (See Section 4.5) and low-level change generation (Section 4.6).

The high-level change generation step generates a set of *high-level changes* of the form "*remove tuple t from the output of operator Op in the canonical representation of the extractor*". Intuitively, removing a tuple t from the output of rule R translates to removing certain tuples involved in the *provenance of t according to the canonical*

operator tree of R. Our solution leverages previous work in the *data provenance* [29] in generating the list of high-level changes. These high-level changes have the potential to remove all incorrect examples from the output. For example, high-level changes for removing the tuple t_{10} from the output of rule R_4 would be "*remove tuple* $(Anna, 555-7789)$ *from the output of the join operator in rule* R_4", or "*remove tuple* t_3 *from the output of the* Dictionary *operator in rule* R_2".

A high-level change indicates *what* operator to modify to remove a given tuple from the final output. However, a high-level change does not tell *how* to modify the operator in order to remove the offending tuple. High-level changes are only the first step towards automating the rule refinement process.

If a rule developer were presented with a set of high-level changes, he or she would need to overcome two major problems in order to translate these high-level changes into usable modifications of the information extraction rule set.

The first problem is one of *feasibility*: The rule writer cannot directly remove tuples in the middle of an operator graph; she is restricted to modifying the rules themselves. It may not be possible to implement a given high-level change through rule modifications, or there may be multiple possible ways to implement the change. Suppose that the Dictionary operator in our example has two parameters: The set of dictionary entries and a flag that controls case-sensitive dictionary matching. There are at least two possible implementations of the second high-level change described above: Either remove the entry `James` from the dictionary, or enable case-sensitive matching. It is not immediately obvious which of these possibilities is preferable.

The second problem is one of *side-effects*. A single change to a rule can remove multiple tuples from the output of the rule. If the rule developer chooses to remove the dictionary entry for `James`, then every false positive that matches that entry will disappear from the output of the Dictionary operator. Likewise, if he or she enables case-sensitive matching, then every false positive match that is not in the

proper case will disappear. In order to determine the dependencies among different high-level changes, the rule developer needs to determine how each high-level change could be implemented and what are the effects of each possible implementation on other high-level changes.

Just as modifying a rule to remove one false positive result can simultaneously remove another false positive result, this action can also remove one or more *correct* results. There may be instances in the document set where the the current set of rules correctly identifies the string "James" as a name. In that case, removing the entry `James` from the dictionary would eliminate these correct results. A given implementation of a high-level change may actually make the results of the rules worse than before.

In the second step of our solution, we go beyond the work done in data provenance and show how to address the issues of feasibility and side-effects. We introduce the concept of a *low-level change*, a specific change to a rule that implements one or more high-level changes. Example low-level changes implementing the two high-level changes above are "*Modify the maximum character distance of the* Follows *join predicate in the join operator of rule* R_4 *from* 60 *to* 50", and "*Modify the* Dictionary *operator of rule* R_2 *by removing entry* james *from dictionary file first_names.dict*", respectively.

Rather than presenting the user with a large and rather unhelpful list of high-level changes, our system produces a ranked list of low-level changes, along with detailed information about the effects and side-effects of each one. Logically speaking, our approach works by generating all low-level changes that implement at least one high-level change; then computing, for each low-level change, the corresponding set of high-level changes. This high-level change information is then used to rank the low-level changes.

A naive implementation of this approach would be prohibitively expensive,

generating huge numbers of possible changes and making a complete pass over the corpus for each one. We keep the computation tractable with a combination of two techniques: pruning individual low-level changes using information available at the operator level and computing side-effects efficiently using cached provenance information.

Since low-level changes are expressed in terms of our internal representation as canonical operator trees, we translate them back to the level of rule statements (we shall show that there is a direct one-to-one translation), prior to showing them to the user. For instance, our two example low-level changes would be presented to the user as '*Modify the maximum character distance of the* Follows *join predicate in the WHERE clause of rule* R_5 *from* 60 *to* 50", and respectively, "*Modify the input of the* Dictionary *table function of rule* R_2 *by removing entry* 'james' *from input dictionary file first_names.dict.*" The user chooses one change to apply, and the entire process is then repeated until the user is satisfied with the resulting rule set.

4.5 Generating High-Level Changes

Definition 14 (High-level change)**.** *Let t be a tuple in an output table V. A* high-level change for t *is a pair (t', Op), where Op is an operator in the canonical operator graph of V and t' is a tuple in the output of Op such that eliminating t' from the output of Op by modifying Op results in eliminating t from V.*

Intuitively, for the removal of t' from the output of Op to result in eliminating t from the final output, it must be that t' *contributes* to generating t. In other words, t' is involved in the *provenance of* t according to the rule set. Hence, to generate all possible high-level changes for t, we first need to compute the provenance of t. Next, we shall first discuss how provenance is computed in our system, and then describe our algorithm for generating high-level changes.

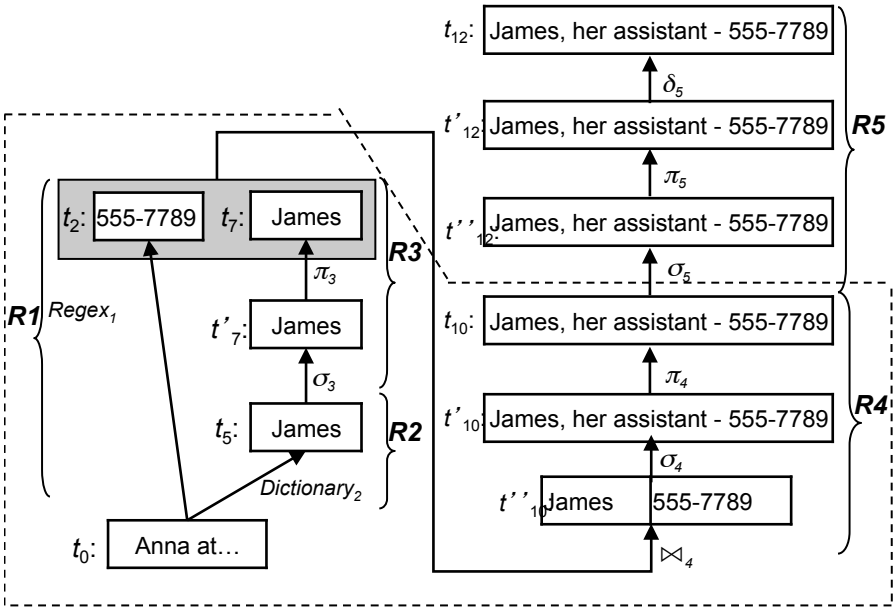


Figure 4.6: Provenance of tuple t_{12} from Figure 4.2.

4.5.1 Computing Provenance

Various definitions have been proposed for describing the *provenance of a tuple t in the result of a query Q*: *why-provenance*: the set of source tuples that contribute to the existence of t in the result, *where-provenance*: the locations in the source database where each field of t has been copied from, and *how-provenance*: the source tuples, and how they were combined by operators of Q to produce t. Among these, how-provenance is the more complete version, since it generalizes why-provenance, and "contains" where-provenance in a certain sense [29]. It is also the most suitable in our context, since knowing which source tuples and how they have been combined by Q to generate an undesirable output tuple t is a pre-requisite to modifying Q in order to remove t from the result. Therefore, in this chapter we shall rely on how-provenance extended to handle text-specific operators (e.g., *Regex*, *Dictionary*).

Given a set of rules Q and input document collection D, a conceptual procedure for computing how-provenance at the level of the operator graph of Q is as follows.

Each tuple passing through the operator graph (i.e., source , intermediate, or output tuple) is assigned a unique identifier. Furthermore, each operator "remembers", for each of its output tuples t, precisely those tuples in its input responsible for producing t. This procedure can be thought of as constructing a *provenance graph for Q on D* that contains an edge $\{t_1, \ldots, t_n\} \xrightarrow{Op} t$ for each combination $\{t_1, \ldots, t_n\}$ of input tuples to operator Op, and their corresponding output tuple t. This provenance graph essentially embeds the provenance of each tuple t in the output of Q on D. As an example, Figure 4.6 shows the portion of the provenance graph for our example in Figure 4.2 that embedds the provenance of tuple t_{12}. Next, we present a procedural definition for the notion of provenance graph.

Definition 15 formalizes the notion of *provenance graph* used in this chapter. Note that the intention of the formalism below is not to propose yet another definition for provenance. In fact, when restricted to the SPJU fragment of SQL, Definition 15 corresponds to the original definition of how-provenance of [43][2]. Rather, our goal is to provide a pictorial representation of provenance that we can use in discussing the algorithm for computing high-level changes.

Definition 15. *[Provenance graph] Let Q be a set of rules and D be a document collection. The* data flow graph of Q and D*, or in short,* the data flow graph of Q *when D is understood from the context, is a hypergraph $G(V, E)$, where V is a set of hypervertices, and E is a set of hyperedges, constructed as follows. For every operator Op in the canonical representation of Q:*

- *If $Op = Regex_{(regex,A)}(R)$, or $Op = Dictionary_{(dict\_file,A)}(R)$, then for every $t \in R$ and every output tuple $t' \in Op(t)$, V contains vertices $v_t, v_{t'}$ and E contains edge $v_t \xrightarrow{Op} v_{t'}$. We say that* the provenance of t' according to Op is t.
- *If $Op = \pi_A(R)$, where A is a set of attributes, then for every $t \in R$ and corresponding output tuple $t' = \pi_A(t)$, V contains vertices $v_t, v_{t'}$ and E contains edge $v_t \xrightarrow{\pi_A} v_{t'}$. We say that* the provenance of t' according to π_A is t.
- *If $Op = \sigma_C(R)$, where C is a conjunction of selection predicates, then for every $t \in R$ and corresponding output tuple $t' = \sigma_C(t)$ (if any), V contains vertices*

[2]Note that since each tuple is assigned a unique identifier, we are essentially in the realm of set semantics.

$v_t, v_{t'}$ *and* E *contains edge* $v_t \xrightarrow{\sigma_C} v_{t'}$. *We say that* the provenance of t' according to σ_C is t.

- *If* $Op = \bowtie_C (R_1, \ldots, R_n)$, *where* C *is a conjunction of join predicates, then for every* $t_1 \in R_1, \ldots, t_n \in R_n$ *and corresponding output tuple* $t' = \bowtie (t_1, \ldots, t_n)$ *(if any),* V *contains vertices* $v_{t_1}, \ldots, v_{t_n}$ *and hypervertex* $\{v_{t_1}, \ldots, v_{t_n}\}$, *and* E *contains hyperedge* $\{v_{t_1}, \ldots, v_{t_n}\} \xrightarrow{\bowtie_C} v_{t'}$. *We say that* the provenance of t' according to $\bowtie_C$ is $\{t_1, \ldots, t_n\}$.
- *If* $Op = \cup(R_1, R_2)$, *then for every* $t_1 \in R_1$ *(or* $t_2 \in R_2$*) and corresponding output tuple* $t' \in \cup(\{t_1\}, \emptyset)$ *(or respectively,* $t' \in \cup(\emptyset, \{t_2\})$*),* V *contains vertices* v_{t_1} *(or* v_{t_2}*) and* $v_{t'}$, *and* E *contains edge* $v_{t_1} \xrightarrow{\cup} v_{t'}$ *(respectively,* $v_{t_2} \xrightarrow{\cup} v_{t'}$*). We say that* the provenance of t' according to $\cup$ is t_1 (or respectively, t_2).
- *If* $Op = \delta(R_1, R_2)$, *then for every* $t \in R_1$ *such that* $t \notin R_2$ *and corresponding output tuple* $t' \in \{t\} - R_2$, V *contains vertices* $v_t, v_{t'}$ *and* E *contains edge* $v_t \xrightarrow{\delta} v_{t'}$. *We say that* the provenance of t' according to δ is t.

In computing the provenance graph, we use a query rewrite approach similar to [41]. The approach of [41] is to rewrite an SQL query Q into a *provenance query* Q^p by recursively rewriting each operator Op in the relational algebra representation of Q. The rewritten version preserves the result of the original operator Op, but adds additional *provenance attributes* through which information about the input tuples to Op that contributed to the creation of an output tuple is propagated. Given Op and a tuple t in its output, the additional information is sufficient to reconstruct exactly those tuples in the input of Op that generated t. Conceptually, the provenance query Q^p records the flow of data from input to output of Q, thus essentially computing the provenance graph of Q for the input document collection. The implementation of our system extends the rewrite approach of [41] to handle text-specific operators. Extensions are straightforward and omitted.

4.5.2 Generating High-level Changes

Figure 4.7 lists our algorithm GenerateHLCs for computing a set of high-level changes, given a set of rules Q, an input document collection D and a set of false positives in the output of Q on D. First, the provenance graph of Q and D is recorded using the rewrite approach outlined in Section 4.5.1. Second, for each false positive t, the

GenerateHLCs(G, X, D)
Input: Operator graph G of a set of rules Q, set X of false positives in the output of G (i.e., Q) applied to input document collection D.
Output: Set H of high-level changes.
Let $H = \emptyset$.

1. Compute the provenance graph $G_p^{Q,D}$ of Q and D;
2. For every $t \in X$ do CollectHLCs($G_p^{Q,D}$, t, H);
3. Return H.

Procedure CollectHLCs(G_p, t', H)
Input: Provenance graph G_p, node t' in G_p, set of high-level changes H.
If t' is a tuple of the Document instance, return.

Otherwise, let e: $T \xrightarrow{Op} t'$ be the incoming edge of t' in G_p. Do:

1. Add (t', Op) to H;
2. If e is of type $t'' \xrightarrow{Op} t'$, where $Op \in \{\pi, \sigma, \cup, \delta, Regex, Dictionary\}$, do CollectHLCs($G_p, t'', H$).
   Otherwise, e is of type $\{t_1, \ldots, t_n\} \xrightarrow{\bowtie} t'$. Do CollectHLCs($G_p$, t_i, H), for all t_i, $1 \leq i \leq n$.

Figure 4.7: Algorithm for computing high-level changes.

algorithm traverses the provenance graph starting from the node corresponding to t in depth-first order, following edges in reverse direction. For every edge $\{\ldots\} \xrightarrow{Op} t'$ encountered during the traversal, one high-level change "remove t' from the output of Op" is being generated.

Suppose the algorithm is invoked on rules R_1 to R_4, with negative output tuple t_{10} and input document from Figure 4.2. The algorithm traverses the provenance graph starting from t_{10} thus visiting each node in the provenance of t_{10} (refer to the dashed rectangle in Figure 4.6), and outputs the following high-level changes: (t_{10}, π_4), (t'_{10}, σ_4), $(t''_{10}, \bowtie_4)$, $(t_2, Regex_1)$, (t_7, π_3), (t'_7, σ_3), $(t_5, Dict_2)$.

4.6 Generating Low-Level Changes

In terms of the relational algebra, a *low-level change* is defined as the change to the configuration of a single operator, or insertion of a new operator subtree in between

two existing operators. Examples include changing the numerical values used in a join condition or a WHERE clause. Notice that the space of all low level changes is unlimited. In order to make the problem tractable, we limit the discussion in this chapter to low-level changes that *restrict* the set of results returned by the query. This is in the same philosophy as [66] – users generally start with a query with high recall and progressively refine it to improve the precision.

4.6.1 Producing Low-Level Changes

Given a set of high-level changes, our goal is to produce a corresponding set of low-level changes, along with enough information about the effects of these changes to rank them. One semi-naive way to compute these low-level changes is to iterate over the operators in the canonical relational algebra representation of the annotator, performing the following three steps:

1. For each operator, consider all the high-level changes that could be applied at that operator.
2. For each such high-level change, enumerate all low-level changes that cause the high-level change.
3. For each such low-level change, compute the set of tuples that the change removes from the operator's output.
4. Propagate these removals up through the provenance graph to compute the end-to-end effects of each change.

This approach computes the correct answer, but it would be extremely slow. This intractability stems directly from the two challenges discussed in Section 4.4: *feasibility* and *side-effects*.

First, the feasibility problem makes step 2 intractable. Just as there could be no feasible low-level change that implements a given high-level change, there could easily be a nearly infinite number of them. For example, consider a high-level change

to remove one of the output tuples of a dictionary operator. Suppose that the dictionary has 1000 entries, one of which produces the tuple. By choosing different subsets of the other 999 entries, one can generate $2^{999} - 1$ distinct low-level changes, any of which removes the desired tuple!

We address this aspect of feasibility by limiting the changes our system considers to a set that is of tractable size, while still considering all feasible combinations of high-level changes at a given operator. In particular, we generate, for each operator, a single low-level change for each of the k best possible combinations of high-level changes; where k is the total number of changes that the system will present to the user. We enforce these constraints through careful design of the algorithms for generating individual types of low-level changes, as we describe in Section 4.6.2.

The side-effects problem causes problems at step 4 of the above approach. Traversing the provenance graph is clearly better than rerunning the annotator to compute the effects of each change. However, even if it generates only one low-level change per operator, the overall cost of this approach is still $O(n^2)$, where n is the size of the operator tree. Such a computation rapidly becomes intractable, as moderately complex annotators can have thousands of operators.

We can reduce this complexity from quadratic to linear time by leveraging our algorithm for enumerating high-level changes. The algorithm in Section 4.5.2 starts with a set of undesirable output tuples and produces, for each input tuple, a set of high-level changes that would remove the tuple. We can easily modify this algorithm to remember the mapping from each high-level change back to the specific output tuple that the change removes.

By running this modified algorithm over every output tuple, including the correct outputs, we can precompute the end-to-end effects of any possible side-effect of a low-level change. With a hash table of precomputed dependencies, we can compute the end-to-end effects of a given low-level change in time proportional to the number

of tuples the change removes from the local operator.

Applying the optimizations described above to the semi-naive algorithm yields the following steps for generating low-level changes.

1. Precompute the mapping from intermediate tuples to the final output tuples they generate.
2. For each operator and each category of low-level change, compute a top-k set of low-level changes.
3. Compute the local effects of each low-level change.
4. Use the table from step 1 to propagate these local effects to the outputs of the annotator.

In the next section, we explain in detail how we perform step 2 efficiently for several different types of low-level change.

4.6.2 Specific Classes of Low-Level Change

We now introduce the specific types of low-level changes that our system currently implements, along with the techniques we use to generate these low-level changes efficiently.

Modify numerical join parameters. This type of change targets the join operator. We use predicate function *Follows* as an example for all joins based on numerical values. Recall that $Follows(span_1, span_2, n_1, n_2)$ returns true if $span_1$ is followed by $span_2$ by a distance value in the range of $[n_1, n_2]$. Low-level changes to a *Follows* predicate involve shrinking the range of character distances by moving one or both of the endpoints.

Our approach to generate low-level changes for numerical join predicates involves interleaving the computation of side-effects with the process of iterating over possible numerical values. Recall that the end goal of our system is to produce a ranked list of low-level changes, where the higher-ranked changes produce a greater improvement

in result quality according to an error metric. We use this ranking function to compute a *utility* value for each value in the range and remove those with low utility. In particular, we compute utility by probing each value in the range: remove it, propagate the change to the output, and compute the change in result quality as the utility of the value in consideration.

We now need to find the top-k sub-sequences in $[n_1, n_2]$ that corresponds to maximum summation of utility values. This problem can be solved with Kadane's algorithm [20] in $O(nk)$ time, where n is the number of values, and k is the number of ranges to find.

Remove dictionary entries. Another important class of low-level change involves removing entries from a dictionary file so as to remove the corresponding dictionary matches from the annotator's input features. Our approach to this type of change takes advantage of the fact that each dictionary entry produces a disjoint set of tuples at the output of the *Dictionary* operator.

As with numerical join parameters, we interleave the computation of low-level changes with the process of computing the effects of each change and the resulting improvement in utility. We start by grouping the outputs of the *Dictionary* operator by dictionary entry. For each dictionary entry that matches at least one high-level change, we compute the tuples that would disappear from the final query result if the entry was removed. We then rank the entries according to the effect that removing that entry would have on result quality. We then generate a low-level change for the top 1 entry, the top 2 entries, and so on, up to k entries. In addition to the dictionary operator, this class of changes also applies, analoguously, to select operators having a dictionary predicate such as `MatchesDict()`.

Add filtering dictionary. This class of changes targets the select operator. In addition to modifying, our system also generates new dictionaries and uses them to filter spans based on the presence of dictionary matches in close proximity. We

produce filtering predicates by composing a span operation like *LeftContextTok* with a dictionary predicate like *Not(ContainsDict())* as in rule R_3 (Fig. 4.2).

To generate filtering predicates our system considers the tokens to the left or right of each span in a tuple affected by a high-level change. The union of these token values forms a set of potential dictionary entries. We rank the effects of filtering with these dictionary entries the same way that we rank changes involving removal of dictionary entries: we group together tuples according to which dictionary entries occur in the vicinity of their spans, and compute the effect of each potential entry on end-to-end result quality.

Add filtering view. Unlike all low-level changes discussed above, which apply to an individual operator, this last type of changes applies to an entire view. Specifically, it involves using subtraction to add a filter view on top of an existing view V. It removes spans from V that overlap with, contain, or are contained in some span of the filtering view. As an example, rule R_5 in Figure 4.2 implements a filtering view on top of *PersonPhoneAll*. To generate filtering views, our algorithm considers every pair of views V_1 and V_2 such that V_1 and V_2 are not descendants of one another in the canonical representation of the ruleset. For each filter policy (OVERLAP, CONTAINS, or *CONTAINED*) the algorithm identifies the tuples of V_1 that are in relationships with at least one V_2 span according to the policy, and ranks the resulting filters according to their effects on the overall end-to-end result quality.

4.7 Experiments

We developed our refinement approach on top of the *SystemT* [30, 85, 61] information extraction system v0.3.6 enhanced with a provenance rewrite engine as described in Section 4.5.1. In this section we present an experimental study of our system in terms of performance, and quality of generated refinements.

4.7.1 Extraction Tasks and Rule Sets

We use two extraction tasks in our evaluation: Person (person entity extraction) and PersonPhone (extraction of relationships between persons and their phone numbers). We chose Person because it is a classic named-entity extraction task and there are standard evaluation datasets available. We chose PersonPhone as an example of a relationship extraction task.

The Person extraction rule set consists of 14 complex rules for identifying person names by combining basic features such as capitalized words and dictionaries of first and last names. Example rules include "*CapitalizedWord* followed by *FirstName*", or "*LastName* followed by a comma, followed by *CapitalizedWord*". We have also included rules for identifying other named-entities such as *Organization*, *Address*, *EmailAddress*, that can be only used as filtering views, in order to enable refinements commonly needed in practice, where person, organizations and locations interact with each other in various ways (e.g., "Morgan Stanley" may be an organization, or a person, "Georgia" may be a person, or a U.S. state).

The PersonPhone extraction rule set consists of 11 complex rules for identifying phone/extension numbers, and a single rule "*Person* followed within 0 to 60 chars by *Phone*" for identifying candidate person–phone relationships (as in rule R_4 from Figure 4.2). To evaluate the system on the relationship task, we use a high-quality Person extractor to identify person names in the PersonPhone task. Note that the system is evaluated separately on the Person task, and we focus on the relationship extractor for the PersonPhone task.

4.7.2 Evaluation Settings

Data Sets

We used the following data sets in our experiments:

- *ACE*: collection of newswire reports, broadcast news and conversations with

Person labeled data from the ACE05 Dataset [8].

- *CoNLL*: collection of news articles with Person labeled data from the CoNLL 2003 Shared Task [102].
- *Enron*, *EnronPP*: collections of emails from the Enron corpus [5] annotated with Person and respectively PersonPhone labels.

The characteristics of the datasets used in our experiments in terms of number of documents and labels in the train and test sets are listed below.

Dataset	Train set		Test set	
	#docs	#labels	#docs	#labels
ACE	273	5201	69	1220
CoNLL	946	6560	216	1842
Enron	434	4500	218	1969
EnronPP	322	157	161	46

We note that these datasets are realistic in practical scenarios[3], as extractor developers are not likely to examine a large number of documents, and obtaining labeled data is known to be a labor intensive and time consuming task. (Machine learning techniques such as active learning [101] have been used to facilitate the latter task.)

Set Up

We developed our refinement approach on top of the *SystemT* [30, 85, 61] information extraction system v0.3.6, enhanced with a provenance rewrite engine as described in Section 4.5.1. The experiments were run on a Ubuntu Linux version 9.10 with 2.26GHz Intel Xeon CPU and 8GB of RAM. Unless otherwise stated, all experiments are from a 10-fold cross-validation.

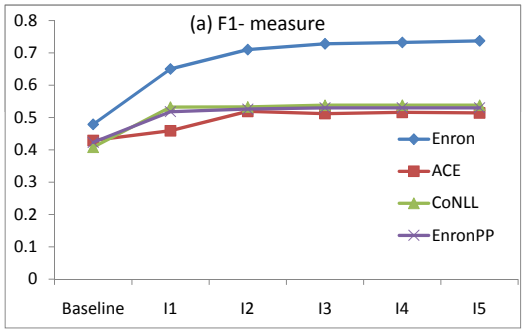
[3]Both *ACE* and *CoNLL* datasets have been used in official Named Entity Recognition competitions [8, 102]

4.7.3 Quality Evaluation

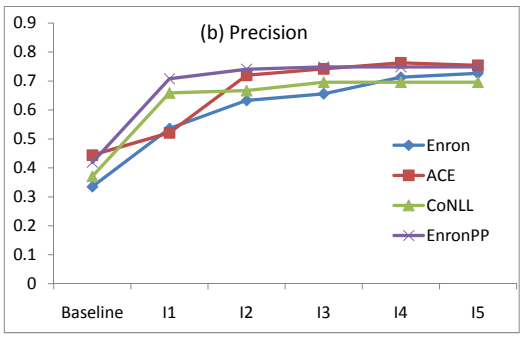
The goal of the quality evaluation is to validate that our system generates high quality refinements in that: 1) they improve the precision of the original rules, while keeping the recall fairly stable, and 2) they are comparable to refinements that a human expert would identify. To this end, we evaluate the quality of refinements produced by our system on a variety of datasets, and we perform a user study where a rule refinement task is presented to human experts and their actions are compared with those suggested by our system. In our evaluation, we use the classical measures of *precision* (percentage of true positives among all extracted answers), *recall* (percentage of true positives extracted among all actual answers), and $F1$-*measure* (harmonic mean of precision and recall).

Experiment 1. We use 4 workloads in this experiment: the Person task on *ACE*, *CoNLL* and *Enron* datasets, and the PersonPhone task on the *EnronPP* dataset. For each workload, we run the system for k iterations starting from the baseline rule set. After each iteration, the refinement with the highest improvement in $F1$-measure on the training set is automatically applied.

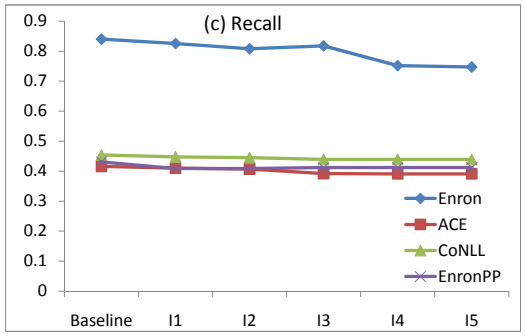
Figure 4.8 shows the quality of k refined rule sets on the test set of each workload, when k is varied from 1 to 5. Note that the quality of the baseline rule sets is as expected in practice, where developers usually start with a query with reasonable recall and progressively refine it to improve precision. As can be seen, our system achieves significant improvements in $F1$-measure between 6% and 26% after only a few iterations. This improvement in $F1$-measure does not arise at the expense of recall. Indeed, as shown in Figures 4.8(b-c), the precision after 5 iterations improves greatly when compared to the baseline rule set, while the recall decreases only marginally. The $F1$-measure and precision plateau after a few refinements for two reasons. First, many false positives are removed by the first few high ranked



(a) $F1$-measure



(b) Precision



(c)Recall

Figure 4.8: Result Quality After Each Iteration of Refinement

refinements, therefore substantially decreasing the number of examples available in subsequent iterations. Second, removing some of the other false positives requires low-level changes that are not yet implemented in our system (e.g., modify a regular expression).

Experiment 2. In this experiment we compare the top refinements generated by our system with those devised by human experts. To this end, we conducted a user study in which two experts A and B were given one hour to improve the rule set for the Person task using the *Enron* train set. Both experts are IBM researchers (not involved in this project) who have written over 20 information extraction rule sets for important IBM products over the past 3 years. To ensure a fair comparison, the experts were restricted to types of refinements supported in our current implementation (Section 4.6.2).

ID	Description	P	R	$F1$	I_1	I_2
	Baseline	35.2	85.0	49.8		
A_1,B_1	Filter Person by Person (CONTAINED)	57.3	83.7	68.0	1	n/a
A_2	Dictionary filter on CapsPerson	70.3	83.9	76.5	4	4
A_3,B_4	Dictionary filter on Person	71.8	83.8	77.4		
A_4	Filter PersonFirstLast by DblNewLine (OVERLAP)	72.6	84.0	77.9	9	5
A_5	Filter PersonLastFirst by DblNewLine (OVERLAP)	72.7	84.1	78.0	9	5
A_6,B_2	Filter PersonLastFirst by PersFirstLast (OVERLAP)	73.5	84.1	78.4	5	3
A_7,B_3	Filter Person by Org (OVERLAP)	74.1	82.5	78.0	3	1
A_8	Filter Person by Address (OVERLAP)	74.3	82.4	78.1	11	9
A_9	Filter Person by EmailAddress (OVERLAP)	77.3	81.7	79.4	12	6

Table 4.1: Expert refinements and their ranks in the list of generated refinements after iterations 1 and 2 (I_1, I_2).

Table 4.1 shows the refinements of both experts and the $F1$-measure improvements achieved on the test set for expert A. (Expert B's refinements are a subset of A's.) The table also shows the rank of each expert refinement in the list automatically generated by our system in the first iteration, and the second iteration after applying the top-most refinement. We observed that the top refinement suggested by the system (remove person candidates strictly contained within other person candidates) coincides with the first refinement applied by both experts (i.e., A_1 and B_1). Furthermore, with a single exception, all expert refinements appear among the top 12 results generated by our system in the first iteration. The dictionary filter generated in iteration 1 consisted of 12 high-quality entries incorrectly identified as part of a person name (e.g., "Thanks", "Subject"). It contains 27% of all entries in corresponding refinement A_2, and all entries in the filter dictionary on person candidates of B_4. Furthermore, in both iterations, the system generated a slightly better refinement compared to A_4 and A_5 that filters all person candidates overlapping with a double new line. This achieves the combined effect of A_4 and A_5, while producing a refined rule set with a slightly simpler structure (a single filter, instead of two). Based on the observations above, we believe it is reasonable to conclude that our system is capable of generating rule refinements that are comparable in quality to those generated by human experts.

4.7.4 Performance Evaluation

The goal of our performance evaluation is two-fold: to validate that our algorithm for generating low-level changes is tractable, since it should be clear that without the optimizations in section 6, CPU cost would be prohibitive, and to show that the system can automatically generate refinements faster than human experts.

The table below shows the running time of our system in the first 3 iterations with the Person rule set on the *Enron* dataset, when the size of the training data is varied between 100 and 400 documents.

Train set #docs	I_1 (sec)	I_2 (sec)	I_3 (sec)	$F1$ after I_3 (%)
100	35.3	1.8	1.1	74.9
200	44.5	6.0	4.2	70.2
300	72.9	9.9	6.3	72.1
400	116.4	21.3	13.6	70.0

As shown above, the system takes between 0.5 and 2 minutes for the first iteration, which includes the initialization time required for loading the rule operators in memory, running the extractor on the training set, and computing the provenance graph, operations performed exactly once. Once initialized, the system takes under 20 seconds for subsequent iterations. As expected, the running time in each iteration decreases, since less data is being processed by the system after each refinement. Also note that the $F1$-measure of the refined rule set after iteration 3 (refer to last column of the table) varies only slightly with the size of the training set.

We note that in each iteration the system sifts through hundreds of documents, identifies and evaluates thousands of low-level changes, and finally presents to the user a ranked list of possible refinements, along with a summary of their effects and side-effects. When done manually, these tasks require a large amount of human effort. Recall from Experiment 2 that the experts took one hour to devise, implement and test their refinements, and reported taking between 3 and 15 minutes per refinement. In contrast, our system generates almost *all* expert's refinements in

iteration 1, in about 2 minutes!

4.8 Conclusions

As we seek to leverage database technology to manage the growing tide of poorly structured information in the world, information extraction has gained growing importance. Most information extraction is based on painstakingly defined extraction rules that are error-prone, often brittle, and subject to continuous refinement. This chapter takes a significant step towards simplifying IE rule development through the use of database provenance techniques.

Specifically, this chapter showed how to modify extraction rules to eliminate false positives in the extraction result. Standard provenance techniques only consider the provenance of tuples in the result set, and hence are not useful for addressing false negatives. However, recent provenance work [26, 46, 49] has begun to develop tools to reason about expected tuples not present in the result set. We believe these techniques can be adapted to our framework to address false negatives in information extraction rules as well. However, this is the subject of future work.

CHAPTER V

ASSISTED QUERYING BY BROWSING

5.1 Introduction

5.1.1 Motivation

Relational databases remain a indispensable resources for users, three decades after their advent. In many situations users without formal database training are required to use SQL to interact with a database. For example, Sloan Digital Sky Survey [9], which hosts the largest publicly available astronomical database, employees SQL as the primary querying means. Researchers and practitioners in astronomy are forced to learn SQL in order to make extensive use of the database. Such scenarios exist in other disciplines such as biology (biologists query multiple databases to investigate biological phenomenon) and even machine learning (researchers specifies features in SQL before submitting the workload for long-time execution).

Querying through SQL is intrinsically difficult for non-experts. It is easily conceivable that learning and writing code in a formal language is difficult for users who may have never written a single line of code. Two major barriers are faced by users. First, structural uncertainty. Users first have to decide the right structure of the results by forming joins across multiple tables, which is difficult if the database schema is complex. Second, value uncertainty. Once the structure of the data is determined, users have to specify complex selection conditions to find the best set of results. This is difficult when the user is in a browsing mode and does not have a

clear idea of the criteria to specify. This is often the case, as evidenced by a recent report by Google [42], according to which, "49% of consumers did not know exactly which type of device they would purchase when the started the shopping process for a new device". Users browse, compare, and then decide. Both uncertainties need to be addressed in order to help users search a database with much less effort.

In this chapter, we propose to address both structural and value uncertainty through a new querying paradigm, Assisted Querying by Browsing (AQB). Users never have to write a SQL query. Instead, they are presented with an initial set of results to start browsing. To address the structural uncertainty, we present to the user the top-k most popular join patterns. Based on the what they see, users choose a view to work on. To address the value uncertainty, we make suggestions on the selection predicates to apply on attributes in the view. Users provide input by simply clicking a button of "Like" or "Dislike" for the tuples that they see. Our system takes such input, refine the suggestions, and presents better choices of tuples. This process continues until users are completely satisfied with the results or suggestions are exhausted.

For ease of understanding, we use a shopping example. Consider our user Amy, who is looking for real estate to purchase through Realtor.com. There are just too many attributes to search in order to find the perfect house. For example, typical attributes include price, size, property status (new construction or second-hand), property types (house or condo), number of beds and baths, parking space, stories, and there are more. It is overwhelming for Amy to specify all these attributes at once - either she is uncertain about them in her own mind or she simple does not know the available data enough to do so.

So, instead of showing a simple table of results (even HTML listing is a table), we can show an additional column, where user can pick and choose her preference, as in Figure 5.1. Note that in the figure we only listed a few available attributes out of

ID	Price (k$)	Size (Sq Ft)	Property Status	Property Type	Beds	Bath	Like or Dislike
231	2,195	5,400	New	House	6	6	
13	1,999	5,200	Recently Sold	House	5	6	
342	3,212	6,500	New	House	7	7	
19	1,211	3,400	Second-hand	House	4	5	
231	569	2,500	Second-hand	Condo	3	3	
765	982	3,100	New	House	4	4	
334	500	2,300	Recently Sold	Condo	2	2	

Figure 5.1: Example Interface

around 20 attributes available on Realtor.com. (Results can be shown with graphics, which are omitted here for simplicity). Amy can pick and choose among the listed ones the one that matches her ideal home the best and label it as "Like" (through the right-most column). For those she clearly dislikes, she can label them as well. Once a few rounds of labeling is done, the system is able to drastically improve the suggestion based on what is learned from Amy. Under the hood, the system is using Amy's labeling to improve a SQL query recommendation engine to suggest the best match for her.

Note that we are not the first to take advantage of such user feedback. By collecting user preferences through "Like" buttons provided by social network sites, search engines such as Bing (through Facebook) and Google (through Google +1) provide personalized search results and improve their advertisement targeting. We are, however, the first to propose querying a structured database directly through browsing and labeling.

This paradigm relies to a certain degree on query logs. Logs are often available

and they provide valuable information on both user preference and characteristics of the data in the database being queried. We analyze the logs to discover the most frequently specified query operations and make query suggestions to the user based on the analysis.

5.1.2 Challenges

Realizing the querying by browsing paradigm requires us to overcome several challenges.

Querying Challenge The structural and value uncertainty of a query are easy to address using SQL through specifying joins and selection predicates. How to help non-experts achieve the same results without SQL is a major challenge.

Cold Start Challenge In the initial deployment period of a database, there is no query log. This makes the approach of completely relying on query logs infeasible. How do we circumvent such an obstacle and still provide users assistance?

Minimum User Effort Challenge Based on user preference on tuples, we need to quickly identify user's information need based on minimum user input. Users have a low threshold of tolerance for the amount of effort to complete a task. Specifically, users do not want to label too many tuples to get started. We need to ask the minimum user input to meet her information needs.

5.1.3 Contributions

In this chapter, we make the following contributions:

- We propose the assisted querying by browsing paradigm for relational databases that completely relieves the user from writing SQL. User interact with a database directly through data items, which is much more tangible and friendly. This addresses both structural and value uncertainty and hence the querying challenge.

- We address the cold start challenge by using a set of automatically generated queries to jump start the system. We use existing data mining techniques to identify clusters of data and generate descriptions for the clusters. These descriptions serve as the initial set of query logs, based on which we can make suggestions.

- We address the minimum user effort challenge by always displaying the most discriminating tuple to the user, which is the tuple that can maximally differentiate one query from the others in the logs. This helps quickly identify user's information need and converge the suggestions.

5.2 Related Work

How to make databases usable has been attracting much interest in both academia and industry. Jagadish et al. [50] overviewed the pain users faces using a relational databases and outlined some research directions. One point argued in the paper is we need to eliminate joins from user's work. Towards the usability goal, many querying paradigms have been proposed or improved, including form-based [54, 55] and spreadsheet-based [67, 69, 19] approaches.

To reduce user's effort in querying, much work has been done on autocompletion. SnipSuggest [59] uses query logs to automatically complete user's SQL query clauses. This approach is shown to be highly effective in predicting frequently used clauses such as popular joins and selection predicates. The application scenario is still SQL-based querying and the user is required to know SQL. Qunits [75] proposes to derive popular parametrized views from query logs, which can be matched with user's keyword queries. This is a keyword-based approach with limited expressive power, and it relies heavily on query logs and hence it suffers from the cold start problem.

Our work is related to Query by Output (QBO) [103], which induces the best

query from a given set of tuples. The purpose of QBO is to find a query that produces the identical result set. Their approach, if applied in our scenario, produces an over-fitted query that has little generalizability and hence it is not applicable.

5.3 Algorithms

5.3.1 Addressing Structural Uncertainty

Based on a query log, we follow these steps to help users identify the best structure of query results.

1. Enumerate all possible joins from the schema by following foreign key references. Each join is assigned to a bucket.

2. For each join that appeared in the log, assign it to the corresponding bucket and increment the count of the bucket. For each bucket, also count the appearances of all combinations of attribute appearances. For example, for all joins on parts and supplier (join on part_id), count the appearance of "part_name", "part_name, supplier_name", etc.

3. At query time, when a user starts to navigate a table, follow all outgoing and incoming foreign key references to this table in the schema. Thus we produce a list of candidate joins and pick the join with highest frequency counts.

In the third step, the results of different join patterns are shown graphically in parallel so users can choose based on actual query results rather than on schema.

Note that in the first two steps, sometimes a join involves more than one hop in the schema graph (multi-way join). Whether we present a one-hop join or multi-hop join depends on their presence in the query log.

Without query log, we can use clustering-based schema summarization techniques [112] to derive most closely related tables from schema alone.

5.3.2 Query Suggestion Based on Tuple Preference

In this section, we discuss how we suggest queries from the query repository to the user based on what we learn about her preference. Note that we only address value uncertainty, assuming that the user has decided on a structure of the result (e.g., all joins are already specified) so that structure uncertainty has been addressed. The goal now is to help users specify complex selection conditions.

Interaction Model

After the user decides on the joins, she is presented with an initial view V. We present a first screen of results for V, where the result tuples are ordered by their power to differentiate one query from another in the query repository using techniques we will specify in Sec. 5.3.3. The user picks tuples that she has a clear preference of like or dislike and assigns them the corresponding label. This input is processed by the system, and two cases may happen. First, the user input so far is not enough for the system to decide on one unique query in the query repository that matches the user's information need. In this case, the user is shown more tuples to label so that she can provide further hints. Second, the system is able to uniquely locate a query based on the input so far, and the user is informed the target query is found. Results of that query are presented to the user.

Differentiating Power of a Tuple

Once the user decides on the structure of the results, we need to help her specify selection conditions. Specifically, our goal is to help the user find a query that matches her preference out of all queries in the repository. Denote the initial view as V, we only need to consider all queries that refine V. We denote this set of queries in the repository as Q.

Note that the differentiating power (DP) of a tuple needs to be considered in two cases, namely, positive feedback and negative feedback. Each tuple t in V has

different power to distinguish one query q from others in Q. If t is a result of q, a user's positive feedback on t increases the likelihood of q being the right query, and a negative feedback on t decreases the likelihood. On the contrary, if t is not a result tuple of q, we have the exact opposite effect. If we do not know the prior likelihood of a tuple being liked or disliked, we assume an equal likelihood.

Consider the real estate example in Figure 5.2, where we have three tuples (t_1, t_2, and t_3), each contains price and size attribute of a house (for simplicity of explanation we picked two popular attributes). The values on the right upper corner show the exact price and size of each tuple. Assume there are previous range queries (q_1, q_2, and q_3) issued by others each represented by a rectangle. A tuple in the query range means that it satisfies the query. Our user, Amy, liking or disliking each tuple promotes or demotes each query differently:

- Likes t_1: both q_1 and q_2 receive a reward, but q_2 should receive a larger reward because it contains a smaller number of results. If Amy stops here, q_2 is her desired query. q_3 receives a penalty because it has less chance of being liked.
- Likes t_2: q_2 gets a penalty while q_1 receives a reward. q_3 receives a penalty.
- Likes t_3: q_3 receives reward and both q_1 and q_2 receive a penalty because neither matches t_3.
- Dislikes t_1: both q_1 and q_2 should be demoted, but q_2 receives a bigger penalty since Amy may still like t_2. q_3, on the other hand, receives a reward.
- Dislikes t_2: q_1 gets a penalty. Other queries receive a reward.
- Dislikes t_3: q_3 receives a penalty and other queries receive a reward.

From the above example, we can intuitively derive a formula for DP similar to TF/IDF (term frequency/inverse document frequency) in information retrieval. For

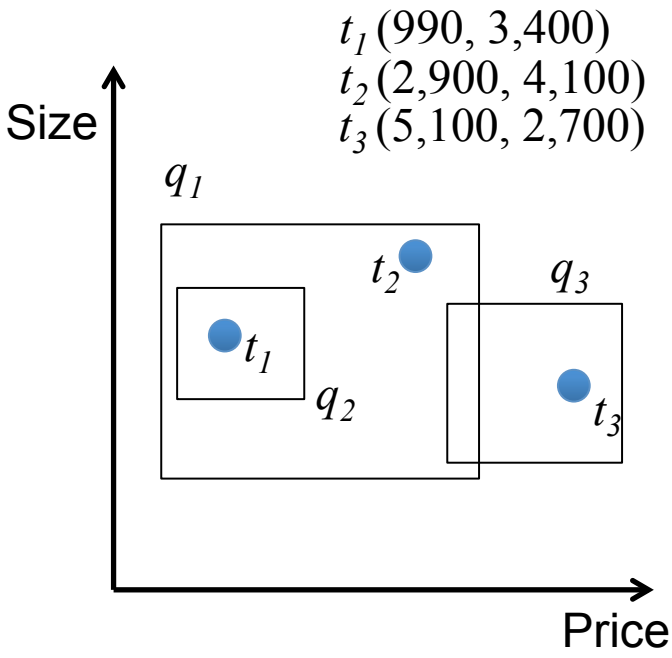


Figure 5.2: Differentiating Power Example on Price and Size Attribute of Real Estate

query q, if tuple t is in the result set of q, the DP is inversely proportional to: i) the number of tuples in the result set of q, and ii) the number of queries out of Q that has t as a result tuple. We denote the two quantities as ResultSize (RS) of q and QueryFrequency (QF) of t. DP of tuple t for query q is given by the following formula:

$$DP(t,q) = \frac{\alpha}{RS(q) \times QF(t)} \tag{5.1}$$

α is a tuning parameter (details in Sec. 5.5).

If t is not a result of q, if the user likes/dislikes t, we need to give q a penalty/reward. Intuitively, if q contains a large number of tuples, it has a higher chance of still matching user's preference. It does not matter to q, however, the number of queries that have t as part of the result. In this case, the DP is given by the following formula:

$$DP(t,q) = \frac{\beta}{RS(q)} \tag{5.2}$$

Similarly, β is also a tuning parameter.

Continuing with the example in Figure 5.2, the values of DP of reach tuple with respect to each query are shown in Table 5.1. We have left both α and β unspecified at the moment, but β is considerably smaller than α. As we can see in this table, tuple t_3 has the biggest maximum DP (with q_3), while t_1 has the smallest maximum DP (with q_1).

Table 5.1: DP of Tuples With Respect to Queries

Tuple \ Query	q_1	q_2	q_3
t_1	$\alpha/4$	$\beta/2$	β
t_2	$\alpha/2$	β	β
t_3	$\beta/2$	β	α

Ranking Queries with Differentiating Power

The definition of differentiating power (DP) allows dynamic ranking of queries in the repository. Initially, all queries in the log have the same score of zero. As the user starts to label tuples, each action from the user adjusts the score of queries, according to formulae in Section 5.3.2. The top k ranked queries are visualized to the user, through actual query results (k is a parameter that can be specified by the user or using a small default value such as 3).

After the user labels the first tuple, each subsequent user action triggers an adjustment of ranking. In order to converge on the suggestion, we add a decaying factor, γ (less than one), to the differentiating power of labeled tuples. Thus, the *i-th* tuple the user labels will have their DP multiplied by γ^{i-1}. The pseudo-code for this procedure is shown in Figure 3.

Algorithm 3 Procedure for Adjusting Query Ranking based on User Label

Input: Q: list of queries in the repository, each identified by its position in the sequence
Input: T: list of tuples labeled by the user
Input: L: label provided by the user for tuples in T, in the same order
Input: RS: result set for each query in Q; RS[q] gets the result set of query q
Input: QF: query frequency for tuples in T; QF[t] gets query frequency of tuple t
α, β, and γ are system parameters
Output: S: score of queries in the repository, in the same order as those in Q

```
for s in S do
    s = 0
end for
for i = 0 do
    if User stops or scores of all queries are below a pre-set threshold then
        Stop
    else
        for j = 0; j < |Q|; j++ do
            if User likes T[i] then
                if (T[i] is in RS[q]) then
                    S[j] += γ^i × α / (RS(q)×QF(T[i]));
                else
                    S[j] += γ^i × β / RS(q)
                end if
            end if
            if User dislikes T[i] then
                if (T[i] is in RS[q]) then
                    S[j] -= γ^i × α / (RS(q)×QF(T[i]));
                else
                    S[j] -= γ^i × β / RS(q)
                end if
            end if
        end for
    end if
end for
```

5.3.3 Serving Tuples for Labeling

In this section, we discuss different orders of serving tuples for users to label. This problem concerns which tuples are shown to the users first, and how to adapt to user's input to dynamically adjust the next batch of tuples to show. Tuples are presented to the user in a ranked list, where the top ranked is the one that is most likely labeled by the user.

Proposed Solution We can treat the decision of which tuple to label first as a decision-tree building process. Each tuple that the user labels can be considered as a decision attribute. This decision, in effect, clusters the queries in the repository as relevant or irrelevant, although the decision boundary is a soft rather than hard one. In addition, unlike the approach in Query by Output [103], we are not trying to devise a tight query that matches exactly tuples the user likes. Rather, we want our suggestion to be able to generalize. Thus we are not using any off-the-shelf decision-tree building algorithm. It is well-known that it is intractable to build the smallest decision tree [87]. Like decision tree building, it is intractable to find the best sequence of tuples to suggest to the user. Inspired by this, we devise a greedy algorithm to choose which tuples to serve first.

Our strategy is to always choose the most differentiating tuple. For query set Q, we denote each of its element by q_i, where $i \in [1, |Q|]$. Each time the user is asked to label a tuple, we call it an "iteration". For each query q_i, we keep a score of ranking, which is updated after each iteration. We denote the score of query q_i before iteration k as by $S_k(q_i)$. Consider the real estate example in Figure 5.2 and Table 5.1. Our approach is to always choose the tuple that leads to the highest DP change to *any* query. Continuing with the housing example, we start with tuple t_3 since it has the highest DP to q_3. If user likes it, we suggest q_3 after this step. Otherwise, we suggest query tuple t_2 in the next step. Since we always choose the tuple with the maximum DP on any query, we call this approach **Maximum Max-DP**.

5.3.4 Generating the Initial Set of Queries

While query logs serve as a great source of insights into users' query interest, they are not always possible to obtain. This is a severe problem when a database is just deployed. Therefore, if we can have some seed queries based on which we can make suggestions without any query logs, our system will be much more robust and applicable.

We solve this problem using a clustering approach. Many data sets contain naturally formed clusters and they can be identified in advance with existing clustering techniques. It is conceivable that those clusters can represent regions that are of different interests to users. For example, if we cluster a real estate database, we are likely to get clusters of high-end houses, mid-range and low-end ones. If we can describe these clusters by their ranges in each attribute, we can use these descriptions as the initial set of queries. For example, if the cluster of high-end houses are described as "price $>$ 1,500k and size $>$ 3,000", this description can be used as a potential query for users who are looking for a high-end house.

So our goal is to identify clusters and their descriptions, preferably with cluster shape being hyper-rectangles in parallel with attributes of data. The CLIQUE algorithm [13] does exactly this.

Introduction to CLIQUE

CLIQUE partitions each dimension into intervals of equivalent length and thus it is a grid-based clustering technique. It accomplishes two goals for us:

- Identifying subspace clusters of the highest dimensionality. It is conceivable that we may deal with data sets of many attributes. CLIQUE applies Apriori Property (first proposed in the seminal paper on association rule mining [14]) to find clusters in subspaces. This not only finds more meaningful clusters but also reduces the complexity of descriptions for the clusters.

- Generating *usable* descriptions for clusters. We stress the usability because we need the descriptions to be easy to interpret. Since GLIQUE merges connected rectangular clusters into bigger clusters, the result clusters have axis-parallel edges. As a result, the cluster edges can be described with simple selection predicates. This makes the cluster descriptions ideal as initial seeding queries.

Applying CLIQUE Offline to Generate Seed Queries

When a database is just set up, we run CLIQUE *offline* on its data. Before doing so, however, we need to analyze the database schema to find useful joins, because CLIQUE is going be to performed on the join results rather than on the initial data set. Those joins are also our best guess of what joins a user may be interested in performing, without any query log.

We apply clustering-based schema summarization technique proposed by Yu and Jagadish [112] to find tables that are most related. This gives us the join paths with maximum potential interests. Once the set of joins are obtained, we perform those joins one by one and store the results in a temporary storage space in separate tables. For each table, we run CLIQUE to cluster the data and generate descriptions. We accumulate all descriptions as seed queries and discard the temporary join results.

Note that this method does require quite some temporary storage and processing time. Storage space is usually not a concern because disk space is quite affordable. We do all processing offline, and CLIQUE is a very efficient algorithm. We do not need to re-do this process when new data comes in, although we may need to do so when database schema changes so significantly that existing joins are broken.

5.3.5 Dealing with Cold Items

A potential problem with our AQB paradigm is that, if a tuple is never mentioned in any previous query, it is not covered by any query and it will never show up in the query result. To the user, this tuple is essentially *lost*. We need to be sure that

every item in the database can be found with browsing, even if no query has ever included the item in the result.

Our initial set of queries, generated from clustering, can potentially deal with a portion of cold items. Note that clustering can leave outliers unclustered; however, those outliers may not be cold items. Some data points of extreme values, although likely to be outliers in clustering, may be frequently queried items. So it is not necessary that clustering leaves cold items uncovered. After the clustering step, only the outliers of clustering are uncovered. An straightforward method is to assign each outlier to the cluster whose centroid is closest. When the query corresponding to the cluster is suggested, we also suggest associated outliers. In this manner, all items can be found through browsing.

Once we have accumulated queries posed by users, we can improve the assignment by assigning outliers to the closest queries. We can expect that the probability of a tuple being missed by all queries decreases with the number of queries in the repository, and the distance from a tuple to its assigned query decreases. Thus the quality of suggestions improves over time.

5.4 Scalability

Scalability is a potential problem for the AQB approach, for several reasons. First, the number of composite tuples is large. Each join pattern corresponds to a set of composite tuples, which means the total cardinality of composite tuples can be a multiple of database size, depending on the database schema. To be able to dynamically serve those tuples to the user for labeling in an interactive setting is a challenge. Second, the set of queries may be large, and we need to quickly identify the best query to suggest; each time a new tuple is labeled, we need to refresh the ranking. Third, we need to efficiently maintain the relationship between a tuple and the query that includes the tuple as part of the result. We now discuss how to

address those issues separately.

Composite Tuple Index We build an index similar to a join index for composite tuples. For each join pattern, we maintain a separate index, where each entry correspond to a composite tuple. The index entry contains record IDs to base tuples. At run time, we can fetch the base tuples to assemble composite tuples on the fly. In this manner, we avoid duplicating the database multiple times.

Query Repository We maintain the set of query log in a table, sorted by the cardinality of query result. Together with the query string, we also maintain the cardinality of the query result.

Inverted Index from Composite Tuples to Queries When a tuple is labeled, we need to quickly identify all queries that have it as part of the query result in order to adjust the weight we assign to the queries. We build an inverted index, from composite tuple to queries for this purpose.

Using the query repository and inverted index, we can quickly compute the differentiating power of tuples. However, we still face the problem of having too many tuples to suggest, if the database is large. We take a uniform sample of all composite tuples if the number is too large to process interactively.

5.5 Experiments

5.5.1 Experimental Settings

We run the experiments on a Macbook Pro with 2.16GHz Intel Core 2 Duo CPU, 4GB of DRAM, and MacOS 10.5.8.

Data Set and Query Log

We use the same data set and query history used in the SIGMOD paper of Chen and Li [28]. The following two paragraphs of description are directly borrowed from [28]:

"The data set contains information on 18537 mutual funds downloaded from

www.scottrade.com. The dataset includes almost all available mutual funds in US market. There are 33 attributes, 28 numerical and 5 categorical. The total data size is 20 MB."

"Query history: We collected query history by asking 14 users (students in our department) to ask queries against the dataset. Each query had 0 to 6 range conditions. The history contained 124 queries with uniform weight. Each user asked about the same number of queries. There were six attributes "1 year return", "3 year return", "standard deviation", "minimal investment", "expenses", and "manager year" in search conditions of these queries. Each query had 4.4 conditions on average."

Experimental Procedure

We measure the absolute measurement of user effort. We randomly select a query q as our target query from the query log. We measure the total number of steps it takes to reach the query through labeling result tuples, assuming that user always labels the top suggested tuple correctly (meaning that the label increases the likelihood of correct queries being pushed up in ranking). We repeat this step for 10 times and report the average and variance. This gives us a measurement of the absolute steps it takes to reach arbitrary query through labeling.

Parameter Tuning

We have to tune the scaling factors we put on the differentiating power and the decaying factor. Like many parameters used in ranking algorithms, they are decided by experiments - we test the performance of different combinations and choose the one that produces the best results.

5.5.2 Results

User Effort Table 5.2 reports the number of steps it takes for each query to complete. We show results for 10 individual runs, as well as mean and variance across these runs.

Table 5.2: Absolute Measurement of User Effort

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_10
Step Count	8	3	5	6	5	3	6	7	5	4

The mean number of steps to reach a query is 5.2, and the variance is 2.62. Running the system is instantaneous, and the total time spent on this task solely depends on how long a user takes to label a tuple. With an average 5.2 tuples to label, the system can quickly converge to user's desired query (if it exists).

5.6 Conclusion

In this chapter, we propose assisted querying by browsing for relational databases. Compared to traditional approaches, our work completely eliminates the need of writing SQL, which has been a huge barrier for non-expert users. In the proposed paradigm, users are suggested some tuples to start up and through user's feedback on tuples we refine our suggestions until user's information need is met. This approach is particularly suited for situations where the user's information is not precisely specified in the beginning, which is often the case in a browsing scenario. Our work is based on database schema and query log analysis, and it is resilient to cold start of the database. Experiments show that we can quickly find results from a database and our query suggestions are of high quality based on a small number of training tuples.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

In this thesis we focus on bringing usability to database systems through the direct manipulation paradigm. We start with a spreadsheet algebra that enables the implementation of a direct manipulation spreadsheet query interface, through which users directly interact with the data. This is our solution to the query specification stage of an information seeking task. We also proposed the Assisted Querying by Browsing paradigm, where users query a database through labeling data tuples. Throughout query specification or after the query is completely specified, users can review results through our *MusiqLens* framework, which generates representatives from the data. This helps relieve users from having to flip through pages of results. When the user finds undesirable results and wants to refine the query, our work in query refinement can be applied.

The components build in the four chapters of this thesis integrate together to form a usable system for users to access a database. The following two scenarios demonstrate how users can better use a database through this system.

- Sam, a freshman in college, wishes to purchase a used car at a bargain price. He is interested in Ford Focus because his family has had pleasant experiences with the brand. Sam opens a used car database using *SheetMusiq* and starts playing with the data. He filters all cars by brand "Ford" and then model "Focus". This eliminates unwanted brands and models, but it still gives him too many results. He turns on the *MusiqLens* feature such that all cars

(all Ford Focus) are presented to him hierarchically. Sam is interested in finding a good balance between price and mileage so he chooses to generate representatives based on those two attributes. Sam can now pick cars of different characteristics, such as those with high mileage but low price, low mileage but high price, or somewhere in between. If this does not satisfy him, Sam can label some results as wanted or unwanted, and ask the system to refine the query for him. The *AutoRefine* project can suggest to narrow down the results further, by suggesting filtering conditions. This process continues until Sam's information needs are met or the system concludes that no result completely matches Sam's requirement.

- Alternatively, Sam can take a different path after specifying that he wants "Ford Focus" in *SheetMusiq*. He can use the assisted querying by browsing (*AQB*) feature and starts picking the cars that matches his interest. *AQB* suggests the most fitting query out of the query repository based on Sam's labeling. Sam may reach a query that completely captures his information needs, or he can settle on a query that is an close approximation. In the former case, he can stop; in the latter case, he can use the suggested query as a starting point, and then performs more manipulation using *SheetMusiq* or query refinement using *AutoRefine*.

In both scenarios, Sam is relieved from the difficulties involved with traditional database interfaces. As a result, the daunting task of finding information from a database becomes much easier and more rewarding.

This thesis strives to achieve a balance between intelligent agents and user-controlled directed manipulation [70]. *SheetMusiq* is a realization of direct manipulation on database querying interface, and it does not provide any additional intelligence other than providing a contextual menu. This allows users to freely express their queries to the database. *MusiqLens* shifts towards intelligent agents by

clustering the data first, although users still interact with it in a direct manipulation fashion. *AutoRefine* takes the intelligence to another level by automatically deriving best refinements while hiding the inner works from the user. *AQB* provides suggestions based on labeling, where much intelligence is involved. As a thesis, there is a healthy mixture of both intelligent agents and direct manipulation, which is exactly what we set out to accomplish.

A natural future directions following this thesis is database usability on the Web. There are millions of databases accessible from the Web. Getting things done over the Web usually involves querying and consuming data from multiple data sources. This means that improving usability of individual databases is just the first step; we need to improve the usability of multiple databases at the Web scale. Information integration approaches such as federated search and data warehousing provide solid solutions only when the number of databases involved is small because they require considerable manual work for each database. We cannot expect to write wrappers or transform data for thousands of databases. Achieving the ultimate goal of making databases over the Web usable requires a long-term effort due to the scale and difficulty of the problem.

A second direction is to drastically improve query refinement effort. Currently, we require users to label a large number of tuples in order to derive high-quality refinements. While this is very useful since it can dramatically cut down refinement time, it is not applicable in an interactive setting. We believe that active learning techniques can be applied to reduce the number of tuples to be labeled through selecting highly promising tuples for labeling first.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Dabble db - online database. http://dabbledb.com/.

[2] Database usability research at university of michigan. http://www.eecs.umich.edu/db/usable/.

[3] Navicat. http://pgsql.navicat.com/.

[4] Tableau software. http://www.tableausoftware.com/.

[5] The Enron corpus. www.cs.cmu.edu/enron/.

[6] Transaction processing performance council. TPC-H Benchmark Specification, Version 2.6.1.

[7] Zoho db & reports. http://db.zoho.com/.

[8] Automatic Content Extraction 2005 Evaluation Dataset, 2005.

[9] Sloan digital sky survey: http://www.sdss.org/, 2010.

[10] E. Agichtein, E. Brill, S. T. Dumais, and R. Ragno. Learning user interaction models for predicting web search result preferences. In *SIGIR*, pages 3–10, 2006.

[11] E. Agichtein and L. Gravano. *Snowball*: extracting relations from large plain-text collections. In *ACM DL*, pages 85–94, 2000.

[12] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *SIGMOD Conference*, pages 94–105, 1998.

[13] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *SIGMOD Conference*, pages 94–105, 1998.

[14] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 207–216. ACM Press, 1993.

[15] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *ACM POPL*, 1979.

[16] A. Aiken, J. Chen, M. Stonebraker, and A. Woodruff. Tioga-2: A direct manipulation database visualization environment. In *ICDE*, pages 208–217, 1996.

[17] D. E. Appelt and B. Onyshkevych. The common pattern specification language. In *TIPSTER workshop*, 1998.

[18] N. Ashish, S. Mehrotra, and P. Pirzadeh. Xar: An integrated framework for information extraction. In *WRI Wold Congress on Computer Science and Information Engineering*, 2009.

[19] E. Bakke and E. Benson. The schema-independent database ui: A proposed holy grail and some suggestions. In *CIDR*, 2011.

[20] J. L. Bentley. Programming pearls: algorithm design techniques. *Commun. ACM*, 27(9):865–873, 1984.

[21] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *ICML*, pages 97–104, 2006.

[22] B. Boguraev. Annotation-based finite state processing in a large-scale nlp architecture. In *RANLP*, pages 61–80, 2003.

[23] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *J. Vis. Lang. Comput.*, 8(2):215–260, 1997.

[24] T. Catarci, P. Dongilli, T. D. Mascio, E. Franconi, G. Santucci, and S. Tessaris. An ontology based visual tool for query formulation support. In *ECAI*, pages 308–312, 2004.

[25] T. Catarci and G. Santucci. Query by diagram: A graphical environment for querying databases. In *SIGMOD*, page 515, 1994.

[26] A. Chapman and H. V. Jagadish. Why not? In *sigmod*, pages 523–534, 2009.

[27] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic information retrieval approach for ranking of database query results. *ACM Trans. Database Syst.*, 31(3):1134–1168, 2006.

[28] Z. Chen and T. Li. Addressing diverse user preferences in sql-query-result navigation. In *SIGMOD Conference*, pages 641–652, 2007.

[29] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[30] L. Chiticariu, R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, and S. Vaithyanathan. SystemT: An Algebraic Approach to Declarative Information Extraction. In *ACL (to appear)*, 2010.

[31] H. Cunningham. JAPE: a Java Annotation Patterns Engine. Research Memorandum CS – 99 – 06, Department of Computer Science, University of Sheffield, May 1999.

[32] A. Das Sarma, A. Jain, and D. Srivastava. I4E: interactive investigation of iterative information extraction. In *Proceedings of the ACM SIGMOD Conference*, pages 795–806, 2010.

[33] D. DeJong. An overview of the frump system. In W. G. Lehnert and M. H. Ringle, editors, *Strategies for Natural Language Processing*, pages 149–176. Hillsdale: Erlbaum, 1982.

[34] M. Ester, H. Kriegel, and X. Xu. *A Database Interface for Clustering in Large Spatial Databases*. Inst. für Informatik, 1995.

[35] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[36] M. Ester, H.-P. Kriegel, and X. Xu. Knowledge discovery in large spatial databases: Focusing techniques for efficient class identification. In *SSD*, pages 67–82, 1995.

[37] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[38] I. O. for Standardization. Information technology – database languages – sql – part 1: Framework (sql/framework). Technical report, 2003. ISO/IEC 9075-1:2003.

[39] D. Freitag. Multistrategy learning for information extraction. In *ICML*, 1998.

[40] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman San Francisco, 1979.

[41] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, pages 174–185, 2009.

[42] Google. Wireless shopper 2.0. 2010.

[43] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *pods*, pages 31–40, 2007.

[44] S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. In *SIGMOD Conference*, pages 73–84, 1998.

[45] P. Hanrahan. Vizql: a language for query, analysis and visualization. In *SIGMOD*, page 721, 2006.

[46] M. Herschel and M. Hernandez. Explaining Missing Answers to SPJUA Queries. *PVLDB*, 2010.

[47] J. R. Hobbs, D. Appelt, J. Bear, D. Israel, M. Kameyama, and M. Tyson. Fastus: a system for extracting information from text. In *HLT '93: Proceedings of the workshop on Human Language Technology*, pages 133–137, Morristown, NJ, USA, 1993. Association for Computational Linguistics.

[48] M. Hua, J. Pei, A. W.-C. Fu, X. Lin, and H. fung Leung. Efficiently answering top-k typicality queries on large databases. In *VLDB*, pages 890–901, 2007.

[49] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1):736–747, 2008.

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

[51] A. Jain, P. G. Ipeirotis, A. Doan, and L. Gravano. Join optimization of information extraction output: Quality matters! In *icde*, pages 186–197, 2009.

[52] T. J. Jankun-Kelly and K.-L. Ma. A spreadsheet interface for visualization exploration. In *IEEE Visualization*, pages 69–76, 2000.

[53] B. J. Jansen and A. Spink. How are we searching the world wide web? a comparison of nine search engine transaction logs. *Inf. Process. Manage.*, 42(1):248–263, 2006.

[54] M. Jayapandian and H. V. Jagadish. Automated creation of a forms-based database query interface. *PVLDB*, 1(1):695–709, 2008.

[55] M. Jayapandian and H. V. Jagadish. Expressive query specification through form customization. In *EDBT*, pages 416–427, 2008.

[56] S. Kandel, A. Paepcke, M. Theobald, and H. Garcia-Molina. The photospread query language. Technical report, Stanford Univ., 2007.

[57] S. Kandel, A. Paepcke, M. Theobald, H. Garcia-Molina, and E. Abelson. Photospread: a spreadsheet for managing photos. In *CHI*, pages 1749–1758, 2008.

[58] L. Kaufman and P. Rousseeuw. Finding groups in data. an introduction to cluster analysis. *Wiley Series in Probability and Mathematical Statistics. Applied Probability and Statistics, New York: Wiley*, 1990.

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

[60] C. Koch. A visual query language for complex-value databases. *ArXiv Computer Science e-prints*, 2006.

[61] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu. SystemT: a system for declarative information extraction. *SIGMOD Record*, 37(4):7–13, 2008.

[62] M. Kuntz and R. Melchert. Pasta-3's graphical query language: Direct manipulation, cooperative queries, full expressive power. In *VLDB*, pages 97–105, 1989.

[63] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*, 2001.

[64] W. G. Lehnert, J. McCarthy, S. Soderland, E. Riloff, C. Cardie, J. Peterson, F. Feng, C. Dolan, and S. Goldman. Umass/hughes: description of the circus system used for muc-5. In *MUC*, pages 277–291, 1993.

[65] C. Li, M. Wang, L. Lim, H. Wang, and K. C.-C. Chang. Supporting ranking and clustering as generalized order-by and group-by. In *SIGMOD Conference*, pages 127–138, 2007.

[66] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish. Regular expression learning for information extraction. In *EMNLP*, pages 21–30, 2008.

[67] B. Liu and H. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *ICDE*, 2009.

[68] B. Liu and H. V. Jagadish. Datalens: Making a good first impression. In *SIGMOD Conference, Demonstration Track*, 2009.

[69] B. Liu and H. V. Jagadish. Using trees to depict a forest. In *VLDB*, 2009.

[70] J. Miller, P. Maes, and B. Shneiderman. Intelligent software agents vs. user-controlled direct manipulation: A debate (panel). In *CHI Extended Abstracts*, pages 105–106, 1997.

[71] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Trans. Knowl. Data Eng.*, 13(1):124–141, 2001.

[72] F. Morii. A generalized k-means algorithm with semi-supervised weight coefficients. In *ICPR (3)*, pages 198–201, 2006.

[73] K. Mouratidis, D. Papadias, and S. Papadimitriou. Medoid queries in large spatial databases. In *SSTD*, pages 55–72, 2005.

[74] K. Mouratidis, D. Papadias, and S. Papadimitriou. Tree-based partition querying: a methodology for computing medoids in large spatial datasets. *VLDB J.*, 17(4):923–945, 2008.

[75] A. Nandi and H. V. Jagadish. Qunits: queried units in database search. In *CIDR*, 2009.

[76] R. T. Ng and J. Han. Clarans: A method for clustering objects for spatial data mining. *IEEE Trans. on Knowl. and Data Eng.*, 14(5):1003–1016, 2002.

[77] R. Nosofsky and S. Zaki. Exemplar and prototype models revisited: Response strategies, selective attention, and stimulus generalization. *Learning, Memory*, 28(5):924–940, 2002.

[78] C. Olston, A. Woodruff, A. Aiken, M. Chu, V. Ercegovac, M. Lin, M. Spalding, and M. Stonebraker. Datasplash. In *SIGMOD*, pages 550–552, 1998.

[79] C. R. Palmer and C. Faloutsos. Density biased sampling: An improved method for data mining and clustering. In *SIGMOD Conference*, pages 82–92, 2000.

[80] F. Pan, W. W. 0010, A. K. H. Tung, and J. Yang. Finding representative set from massive data. In *ICDM*, pages 338–345, 2005.

[81] L. Parsons, E. Haque, and H. Liu. Subspace clustering for high dimensional data: a review. *ACM SIGKDD Explorations Newsletter*, 6(1):90–105, 2004.

[82] F. Peng and A. McCallum. Accurate information extraction from research papers using conditional random fields. In *HLT-NAACL*, 2004.

[83] R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw-Hill Boston, 2003.

[84] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, pages 381–390, 2001.

[85] F. Reiss et al. An algebraic approach to rule-based information extraction. In *ICDE*, 2008.

[86] E. Riloff. Automatically constructing a dictionary for information extraction tasks. In *KDD*, 1993.

[87] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. 2009.

[88] C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. Silva. Querying and re-using workflows with vistrails. In *ACM SIGMOD*, 2008.

[89] W. Shen, P. DeRose, R. McCann, A. Doan, and R. Ramakrishnan. Toward best-effort information extraction. In *sigmod*, pages 1031–1042, 2008.

[90] W. Shen et al. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, 2007.

[91] H. Shin and R. Nosofsky. Similarity-scaling studies of dot-pattern classification and recognition. *Journal of Experimental Psychology: General*, 121(3):278–304, 1992.

[92] B. Shneiderman. A computer graphics system for polynomials. *The Mathematics Teacher*, 67(2):111–113, 1974.

[93] B. Shneiderman. The future of interactive systems and the emergence of direct manipulation. *Behaviour & Information Technology*, 1(3):237–256, 1982.

[94] B. Shneiderman. Direct manipulation: a step beyond programming languages. *IEEE Computer*, 16(8):57–69, 1983.

[95] B. Shneiderman, D. Byrd, and W. B. Croft. Sorting out searching: A user-interface framework for text searches. *Commun. ACM*, 41(4):95–98, 1998.

[96] J. Smith and J. Minda. Prototypes in the mist: The early epochs of category learning. *Learning, Memory*, 24(6):1411–1436, 1998.

[97] S. G. Soderland. Learning text analysis rules for domain-specific natural language processing. Technical report, Amherst, MA, USA, 1996.

[98] M. Spenke and C. Beilken. A spreadsheet interface for logic programming. In *CHI*, pages 75–80, 1989.

[99] M. Spenke, C. Beilken, and T. Berlage. Focus: The interactive table for product comparison and selection. In *UIST*, pages 41–50, 1996.

[100] S. Tata, J. M. Patel, J. S. Friedman, and A. Swaroop. Declarative querying for biological sequences. In *ICDE*, page 87, 2006.

[101] C. Thompson, M. Califf, and R. Mooney. Active Learning for Natural Language Parsing and Information Extraction. In *ICML*, pages 406–414, 1999.

[102] E. F. Tjong Kim Sang and F. De Meulder. Introduction to the CoNLL-2003 Shared Task: Language-independent Named Entity Recognition. In *CoNLL at HLT-NAACL*, pages 142–147, 2003.

[103] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *SIGMOD Conference*, pages 535–548, 2009.

[104] J. Ullman. *Principles of database and knowledge-base systems, Vol. 1.* Computer Science Press, Inc. New York, NY, USA, 1988.

[105] J. Ullman. *Principles of database and knowledge-base systems, Vol. 2.* Computer Science Press, Inc. New York, NY, USA, 1988.

[106] E. Vee, U. Srivastava, J. Shanmugasundaram, P. Bhat, and S. Amer-Yahia. Efficient computation of diverse query results. In *ICDE*, pages 228–236, 2008.

[107] A. Witkowski, S. Bellamkonda, T. Bozkaya, G. Dorman, N. Folkert, A. Gupta, L. Sheng, and S. Subramanian. Spreadsheets in rdbms for olap. In *SIGMOD*, 2003.

[108] A. Witkowski, S. Bellamkonda, T. Bozkaya, A. Naimat, L. Sheng, S. Subramanian, and A. Waingold. Query by excel. In *VLDB*, pages 1204–1215, 2005.

[109] T. Wu, X. Li, D. Xin, J. Han, J. Lee, and R. Redder. Datascope: Viewing database contents in google maps' way. In *VLDB*, pages 1314–1317, 2007.

[110] R. Xu and I. Donald Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645, 2005.

[111] A. Yates, M. Banko, M. Broadhead, M. J. Cafarella, O. Etzioni, and S. Soderland. Textrunner: Open information extraction on the web. In *HLT-NAACL (Demonstrations)*, pages 25–26, 2007.

[112] C. Yu and H. V. Jagadish. Schema summarization. In *VLDB*, pages 319–330, 2006.

[113] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. In *SIGMOD Conference*, pages 103–114, 1996.

[114] S. Zhao and R. Grishman. Extracting relations with integrated information using kernel methods. In *ACL*, 2005.

[115] M. M. Zloof. Query-by-example: the invocation and definition of tables and forms. In *VLDB*, pages 1–24, 1975.