DataLens: Making a Good First Impression

Bin Liu
Department of EECS
University of Michigan
Ann Arbor, USA
binliu@umich.edu

H.V. Jagadish
Department of EECS
University of Michigan
Ann Arbor, USA
jag@umich.edu

ABSTRACT

When a database query has a large number of 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. This approach is well-suited for information retrieval, and for some database queries, such as similarity queries or under-specified (or keyword) queries with known (or guessable) user preferences. However, standard database query results comprise a set of tuples, with no associated ranking. It is typical to allow users the ability to sort results on selected attributes, but no actual ranking is defined.

An alternative approach is not to try to show the estimated best results on the first page, but instead to help users learn what is available in the whole result set and direct them to finding what they need. We present DataLens, a framework that: i) generates the most representative data points to display on the first page without sorting or ranking, ii) allows users to drill-down to more similar items in a hierarchical fashion, and iii) dynamically adjusts the representatives based on the user's new query conditions. To the best of our knowledge, DataLens is the first to allow hierarchical database result browsing and searching at the same time.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation**]: User Interfaces

General Terms

Human Factors, Design

Keywords

interface, representative, exploration

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. In this demonstration we show how best to present these results to the user.

The "Many-Answers Problem" has been well documented [4]: 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 1 below, where a user searches a used car database for a Honda Civic.

Example 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?

Most current work attempts 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 (e.g., [7, 5]) 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. The absence of user preference or query history/workload also prevents approaches such as results categorization [3, 6] and ranking-based navigation [12] from being applicable in this scenario.

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 [1]. 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 needs, the first page has to deliver a strong message that there are interesting results in the remaining pages. Browsing results from a relational database is similar from that of a search engine, and we must ensure that the first page of results catch user's full attention.

Supported in part by NSF grant numbers 0438909 and 0741620.

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.
ACM 978-1-60558-551-2/09/06.

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 1: DataLens 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 2: After Zooming on First Tuple

2. THE DATALENS FRAMEWORK

In this demonstration, 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 [11]. 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 those examples, we should make them as representative as possible.

To accomplish the above task, we propose a framework called *DataLens* for database systems that: i) automatically displays the best *representative* result tuples in the first screen of results when the result set is large, ii) at user's request, displays more tuples similar to a particular representative, and iii) consistently adapts to user's subsequent query operations (selections and zooming). A typical first page is shown in Figure 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 ranges 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, DataLens shows representatives from the subset of cars (Figure 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 can add a condition `price < 10,000`. The next screen of results is 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.

3. NOVEL ISSUES

In this section, we first outline the challenges in building DataLens and then briefly explain our solutions.

3.1 Challenges

Two challenges must be addressed before one can construct an effective interface such as the one shown in Figure 1. We discuss these below. Let the first page of results be limited to k tuples. We call tuples on the first page *representatives* of the whole result set. We choose k-medoids as the representatives of a data set.

Representative Finding Challenge We need to efficiently find representatives for the result set. That is, find the k-medoids with the minimum average distance (or other chosen metric). There are two related requirements here: i) the first batch of representatives for a query should be generated quickly, and ii) zooming operations should incur minimal computation.

Query-Refinement Challenge In addition to zooming, DataLens needs to adapt to user query refinement. Once the user sees the first screen of representatives, she may dynamically refine her query results based on the information she gathered from seeing some data. For example, Ann may decide to restrict her search to cars with less than 60,000 miles (instead of the 80,000 originally specified). This may make some current representatives invalid. In interactive querying, we cannot afford to recompute the representatives from scratch. Instead, we should seek to incrementally maintain the set of representatives.

3.2 Generating Initial Representatives

We need to be able to generate k-medoids efficiently. Formally, for a set of data points $\boldsymbol{O}$, *k-medoids* is a subset $\boldsymbol{M}$ with cardinality k that minimizes the average distance from each point in $\boldsymbol{O}$ to the closest point in $\boldsymbol{M}$. In the literature there are many k-medoid clustering methods such as PAM [8], CLARA [8], BIRCH [13], and CLARANS [10]. They mostly focus on high quality clustering, while we target high efficiency and consistency under new query conditions. In interactive querying, we expect that users can tolerate less than perfect representatives but not obvious delays.

We propose a cover-tree [2] based algorithm that is uniquely suited for DataLens. For each level i of the cover-tree, it satisfies four properties: i) each node is also a data point, ii) a node appears in level $i+1$ if it appears in level i, iii) nodes are separated by at least $D(i)$, which is a monotonically decreasing function of i, and iv) nodes are within distance $D(i)$ to its children in level $i+1$, and are within $2 \times D(i)$ to all its descendants. These properties naturally cluster data in the tree, where the root of a subtree is a good representative of data in the subtree. We pre-compute a cover tree for the data set and use tree-assisted heuristics to efficiently find the k-medoids.

3.3 Query Adaptation

Users can add ad-hoc refinement conditions or zoom-in on tuples in DataLens. If we were to re-compute the representatives from scratch, we would have to re-build the cover tree index and re-run the algorithm outlined above, which is too costly in interactive querying. We propose to re-use the original index to generate new medoids.

Selection. 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 2D case, but high dimensional cases are easy to generalize to. For example, if we use Price as x-axis and Mileage as y-axis in 2D space,

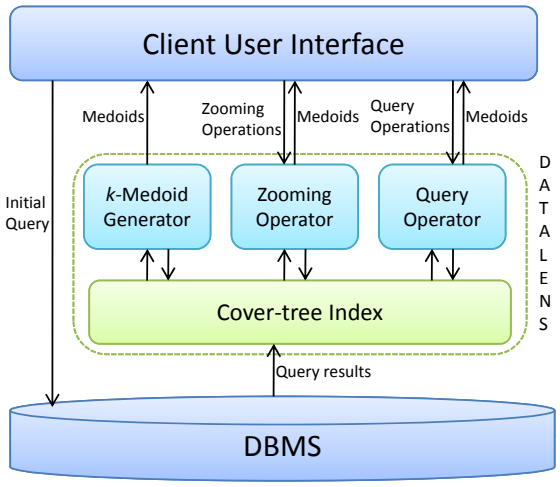

Figure 3: DataLens System Architecture

adding the selection condition "price < 12000" removes all data points that are to the right of line $x = 12000$. For a node in the level i of the cover tree, all its descendants must be within distance $2D(i)$. We can use this fact to estimate the percentage of the subtree that remains valid after the selection condition by considering the area in the valid region. By considering this percentage value as the utility of each cover tree node, we can modify our heuristics to generate k-medoids when the tree is only partially valid.

Zooming-in. If the user chooses to click on a tuple, we first fetch all nodes in the cover tree that are associated with this tuple, namely, all cover tree nodes that contributed to the generation of this representative. We then re-run our algorithm on this smaller set of nodes (and their subtrees) to generate k-medoids around the chosen tuple.

4. SYSTEM ARCHITECTURE

The architecture of DataLens is shown in Figure 3. When a query is initially sent from the client user interface to the DBMS, query results are fed to DataLens, which interacts with the client in this query session. DataLens 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.

Beside the indexer, the core of DataLens 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 representatives after user performs a zooming operation; and the query operator, which dynamically adjusts the medoids according to user's new query conditions. DataLens can be implemented as a module in a DBMS or a layer between the client and the DBMS.

5. DEMONSTRATION

In this demonstration, DataLens is implemented as a layer between PostgreSQL server and the *SheetMusiq* spreadsheet query interface [9], through which users can query a database using mouse-clicks in a direct manipulation fashion. The interfaces examples shown in Figures 1 and 2 are implemented in *SheetMusiq*. When a new query is sent to the database server through the client, representatives are immediately shown. Users can click on a particular item in the spreadsheet to zoom in to more similar items, or pose new filtering conditions. Users will see how DataLens dynamically adjusts to the query operations and maintains a good set of representatives.

We use two datasets: "Car" crawled from Yahoo! Autos and "House" crawled from realtors.com. Car dataset contains 4k cars with typical attributes such as mileage and price. House dataset contains 3k houses listed, containing information on price, living area size, total property size, and number of bedrooms. We also offer users the opportunity to plug in their own dataset (in a format compatible with PostgreSQL).

6. REFERENCES

[1] 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.

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

[3] K. Chakrabarti, S. Chaudhuri, and S. won Hwang. Automatic categorization of query results. In *SIGMOD Conference*, pages 755–766, 2004.

[4] 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.

[5] S. Chaudhuri and L. Gravano. Evaluating top-selection queries. In *VLDB*, pages 397–410, 1999.

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

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

[8] 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.

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

[10] 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.

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

[12] 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.

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