Interactive Summarization and Exploration of Top Aggregate Query Answers*

Yuhao Wen, Xiaodan Zhu, Sudeepa Roy, Jun Yang
Department of Computer Science, Duke University
{ywen, xdzhu, sudeepa, junyang}@cs.duke.edu

ABSTRACT

We present a system for summarization and interactive exploration of high-valued aggregate query answers to make a large set of possible answers more informative to the user. Our system outputs a set of clusters on the high-valued query answers showing their common properties such that the clusters are diverse as much as possible to avoid repeating information, and cover a certain number of top original answers as indicated by the user. Further, the system facilitates interactive exploration of the query answers by helping the user (i) choose combinations of parameters for clustering, (ii) inspect the clusters as well as the elements they contain, and (iii) visualize how changes in parameters affect clustering. We define optimization problems, study their complexity, explore properties of the solutions investigating the semi-lattice structure on the clusters, and propose efficient algorithms and optimizations to achieve these goals. We evaluate our techniques experimentally and discuss our prototype with a graphical user interface that facilitates this interactive exploration. A user study is conducted to evaluate the usability of our approach.

PVLDB Reference Format:
Yuhao Wen, Xiaodan Zhu, Sudeepa Roy, Jun Yang. Interactive Summarization and Exploration of Top Aggregate Query Answers. *PVLDB*, 11 (13): 2196-2208, 2018.
DOI: https://doi.org/10.14778/3275366.3275369

1. INTRODUCTION

Summarization and diversification of query results have recently drawn significant attention in databases and other applications such as keyword search, recommendation systems, and online shopping. The goal of both result summarization and result diversification is to make a large set of possible answers more informative to the user, since the user is likely not to view results beyond a small number. This brings the need to make the *top-k* results displayed to the user *summarized* (the results should be grouped and summarized to reveal high-level patterns among answers), *relevant* (the results should have high *value* or *score* with respect to user's query or a database query), *diverse* (the results should avoid repeating information), and also providing *coverage* (the results should cover top answers from the original non-summarized result set). In this paper, we present a framework to summarize and explore high valued aggregate query answers to understand their common properties easily and efficiently while meeting the above competing goals simultaneously. We illustrate the challenges and our contributions using the following example:

EXAMPLE 1.1. *Suppose an analyst is using the movie ratings data from the* MovieLens *website [28] to investigate average ratings of different genres of movies by different groups of users over different time periods. So the analyst first joins several relations from this dataset (information about movies, ratings, users, and their occupations) to one relation R, extracts some additional features from the original attributes (age group, decade, half-decade), and then runs the following SQL aggregate query on R (the join is omitted for simplicity). In this query,* `hdec` *denotes disjoint five-year windows of half-decades, e.g., 1990 (=1990-94), 1995 (=1995-99), etc.;* `agegrp` *denotes age groups of the users in their teens or 10s (i.e., 10-19), 20s (i.e., 20-29), etc.*

```
SELECT hdec, agegrp, gender, occupation, AVG(rating) AS val
FROM R
GROUP BY hdec, agegrp, gender, occupation
WHERE genres_adventure = 1
HAVING COUNT(*) > 50
ORDER BY val DESC
```

The top 8 and bottom 8 results from this query are shown in Figure 1a. To have a quick summary of these 50 result tuples, The data analyst is interested in seeing the summary in at most four rows to have an idea of the viewers and time periods with a high rating for the adventure genre.

One straightforward option is to output the top 4 result tuples from Figure 1a, but they do not summarize the common properties of the intended viewers/times periods. In addition, despite having high scores, they have attribute values that are close to each other (*e.g.*, male students in their 20s) leading to repetition of information and sub-optimal use of the designated space of $k = 4$ rows. More importantly, *the top k original tuples may give a wrong impression on the common properties of high-valued tuples even if they all share those properties*. For instance, three out of top four tuples share the property `(20s, M)`, but it is misleading, since a closer look at Figure 1a reveals that many tuples with low values (49th, 46th, 44th) share this property too, suggesting that male viewers in their 20s may or may not give high ratings to the adventure genre.

*This work was supported in part by NSF awards IIS-1408846, IIS-1552538, IIS-1703431, IIS-1718398, IIS-1814493, NIH award 1R01EB025021-01, a grant from the Knight Foundation, and a Google Faculty Award. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies.

Proceedings of the VLDB Endowment, Vol. 11, No. 13
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3275366.3275369

Rank	hdec	agegrp	gender	occupation	val
1	1975	20s	M	Student	4.24
2	1980	20s	M	Programmer	4.13
3	1980	10s	M	Student	3.96
4	1980	20s	M	Student	3.91
5	1985	20s	M	Programmer	3.86
6	1980	20s	M	Engineer	3.83
7	1985	10s	M	Student	3.77
8	1985	20s	M	Student	3.76
.....					
43	1995	30s	M	Marketing	3.02
44	1995	20s	M	Technician	2.92
45	1995	30s	M	Entertainment	2.91
46	1995	20s	M	Executive	2.91
47	1995	30s	F	Librarian	2.84
48	1995	30s	M	Student	2.81
49	1995	20s	M	Writer	2.51
50	1995	20s	F	Healthcare	1.98

(a) Top-8 and bottom-8 tuples with values as score

hdec	agegrp	gender	occupation	avg val	
1975	**20s**	**M**	**Student**	**4.24**	▼
1980	∗	**M**	∗	**3.96**	▼
1985	**20s**	**M**	**Programmer**	**3.86**	▼
1985	∗	**M**	**Student**	**3.76**	▼

(b) Clusters with average score

hdec	agegrp	gender	occupation	avg val	rank
1975	**20s**	**M**	**Student**	**4.24**	▼
1975	20s	M	Student	4.24	1
1980	∗	**M**	∗	**3.96**	▼
1980	20s	M	Programmer	4.13	2
1980	10s	M	Student	3.96	3
1980	20s	M	Student	3.91	4
1980	20s	M	Engineer	3.83	6
1985	**20s**	**M**	**Programmer**	**3.86**	▼
1985	20s	M	Programmer	3.86	5
1985	*	**M**	**Student**	**3.76**	▼
1985	10s	M	Student	3.77	7
1985	20s	M	Student	3.76	8

(c) Original result tuples (with ranks) in the clusters

Figure 1: Illustrating our framework for $k = 4$, $L = 8$, $D = 2$. In general, original result tuples outside top-L can belong to the output clusters, although here we happen to have a solution that covers just the top-L result tuples.

Therefore, we aim to achieve a summarization with the following desiderata: (i) it should be simple and memorable (*e.g.*, male students or `(M, Students)`), (ii) it should be diverse (*e.g.*, `(1975, 20s, M, Student)` and `(1980, 20s, M, Student)` might be too similar), and (iii) it should be discriminative (*e.g.*, the properties like `(20s, M)` covering both high and low valued tuples should be avoided). Furthermore, it should be achieved at an interactive speed and displayed using a user-friendly interface.

In recent years, work has been done to *diversify* a set of result tuples by selecting a subset of them (discussed further in Section 2), *e.g.*, *diversified top-k* [23] takes account of diversity and relevance while selecting top result tuples; *DisC diversity* [7] takes into account similarity with the tuples that have not been selected, and diversity and relevance in the selected ones. In contrast, we intend to output summarized information on the result tuples by displaying the common attribute values in each cluster to give the user a holistic view of the result tuples with high value. In this direction, the *smart drill-down* [18] framework helps the user explore summarized "interesting" tuples in a database, but it does not focus on aggregate answers, or helping the user choose input parameters and understand consecutive solutions, which are two key features of our framework. As discussed in Section 2 and observed in experiments in our initial exploration, standard clustering or classification approaches do not give a meaningful summary of high-valued results as well. In particular, we support summarization and interactive exploration of aggregate answers in the following ways each posing its own technical challenges.

(1) Summarizing Aggregate Answers with Relevance, Diversity, and Coverage. To meet the desiderata of a good summarization, the basic operation of our framework involves generating a set of *clusters* summarizing the common properties or common attribute values of high-valued answers (Section 3). If all elements in a cluster do not share the same value for an attribute, then the value of that attribute is replaced with a '$*$' [1]. The clusters can be expanded to show the elements contained in them to the user. To compute the clusters, our framework can take (up to) three parameters as input: (i) *size constraint* k denotes the number of rows or clusters to be displayed (k = 4 in Example 1.1), (ii) *coverage parameter* L, requiring that the top-L tuples in the original ranking must be covered by the k chosen clusters, and (iii) *distance parameter* D, requiring that the summaries should be at least distance D from each other to avoid repeating similar information.

EXAMPLE 1.2. *Suppose we run our framework for the query in Example 1.1 with parameters $k = 4, L = 8$, and $D = 2$, i.e., the user would like to see at most 4 clusters, these clusters should cover top 8 tuples from Figure 1a, and any two clusters should not have identical values for more than two attributes. Our framework first displays the four clusters shown in Figure 1b along with the average scores of result tuples contained in them.*

The user may choose to investigate any of these clusters by expanding the cluster on our framework (clicking ▼). If all four clusters are expanded by the user, the second-layer will reveal all original result tuples they cover, as shown in Figure 1c. In this particular example, no other tuples outside top 8 have been chosen by our algorithm (which is also the optimal solution), but in general, the selected clusters may contain other tuples (high-valued but not necessarily in top L).

The above example illustrates several advantages and features of our framework in providing a meaningful and holistic summary of high-valued aggregate query answers. *First,* the original top 8 result tuples are not lost thanks to the second layer, whereas the properties that combine multiple top result tuples are clearly highlighted in the clusters in the first layer. *Second,* the chosen clusters are diverse, each contributing some extra novelty to the answer. *Third,* the clustering captures the properties of the top result tuples that distinguish them from those with low values. For instance, the cluster for `(20s, M)` does not appear in the solution, since this property is prevalent in both high-valued and low-valued tuples as discussed before. Clearly, this could not be achieved by simply clustering top L tuples by k clusters. This is ensured by our objective function that aims to maximize the *average* value of the tuples covered by all clusters (instead of maximizing the sum).

To achieve the solution as described above, we make the following technical contributions in the paper:

- To ensure that the chosen clusters cover answer tuples with high values, we formulate an optimization problem that takes k, L, D as input, and outputs clusters such that the *average value* of the tuples covered by these clusters is maximized. We study the complexity of the above problem (both decision and optimization versions) and show NP-hardness results (Section 4).

[1] Our framework and algorithms can be extended to more fine-grained generalizations of values beyond $*$ (by introducing a concept hierarchy over the domain) [34].

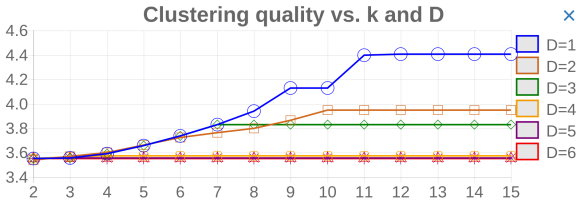


Figure 2: Visualization for parameter selection: how results vary for different k and D (some lines overlap).

- We design efficient heuristics satisfying all constraints using properties of the *semi-lattice structure* on the clusters imposed by the attributes (Section 5).
- We perform extensive experimental evaluation using the MovieLens [28] and TPC-DS benchmark [21] datasets (Section 7).

(2) Interactive Clustering and Parameter Selection. The intended application of our framework is an interactive exploration of query results where the user may keep updating k, L, or D to understand the key properties of the high-valued aggregate answers. One challenge in this exploration is to select values of k, L, D while ensuring interactive speed, since straightforward implementations of our algorithms would not be fast enough. To support parameter selection, we provide the user with a holistic view of how the objective varies with different choices of parameters. This view helps users identify "flat regions" (uninteresting for parameter changes) vs. "knee points" (possibly interesting for parameter changes) in the parameter space. One example is shown in Figure 2, where given selected values of $L = 15$ as in Section 6.1, how the average value of the solutions (y-axis) varies with different k (x-axis) is shown. This figure illustrates that if k is changed from $k = 11$ to $k = 7$, there will be a drop in the overall value. The user can select different legends for different lines and check the value in detail by hovering over a point. This visualization also helps the user validate the choice of parameters, *e.g.*, if a smaller value of k can give a similar quality result, the user may want to reduce the value of k to have a more compact solution. This feature not only helps in guiding the user select parameter values[2], it also serves as a *precomputation* step to retrieve the actual solutions for different combinations of input parameters k, L, and D at an interactive speed. (Section 6).

- We develop techniques for *incremental computation and efficient storage* for solutions for multiple combinations of input parameters using an *interval tree* data structure.
- We implement multiple optimization techniques to further speed up computation of these solutions. We evaluate the effect of these optimizations experimentally. Eventually we achieved 30x-1000x speed up from these optimizations, which helped us achieve our goal of interactive speed.

Roadmap. We discuss the related work in Section 2 and define some preliminary concepts in Section 3. The above sets of results are discussed in Sections 4, 5 and 6. The experimental results are presented in Section 7. A user study is conducted on Section 8. We conclude in Section 9 with scope of future work. Some details are deferred to the full version[34] due to space constraints.

[2] To further assist in parameter selection, our system also allows visual comparison of two successive solutions showing how the clusters are redistributed. We formulated an optimization problem to enable clean visualization and provided optimal solutions. Due to space constraints, the details are in the full version [34] and in our demonstration paper in SIGMOD 2018 [35].

2. RELATED WORK

First we discuss three recent papers relevant to our work that consider *result diversification* or *result summarization*: smart drill-down [18], diversified top-k [23], and DisC diversity[7]. We explored using or adapting the approaches proposed in these papers for our problem, but since they focus on different problems, as expected, the optimization, objective, and the setting studied in [18, 23, 7] do not suffice to meet the goals in our work; There are several other related work in the literature that we briefly mention below. Qualitative comparison results and details are discussed in [34].

Smart drill-down [18]: In a recent work, Joglekar et al. [18] proposed the *smart drill-down* operator for interactive exploration and summarizing interesting tuples in a given table. The outputs show top-k rules (clusters) with don't-care $*$-values. The goal is to find an ordered set of rules with maximum score, which is given by the sum of product of the *marginal coverage* (elements in a rules that are not covered by the previous rules) and *weight* of the rules (a "goodness" factor, *e.g.*, a rule with fewer $*$ is better as it is more specific). In [34], we show with examples that this approach is not suitable for summarizing aggregate query answers, since it will prefer common attribute values prevalent in many tuples and may select rules containing both high- and low-valued tuples.

Diversified top-k [23]: Qin et al. [23] formulated the top-k result diversification problem: given a relation S where each element has a score and any two elements have a similarity value between them, output at most k elements such that any two selected elements are dissimilar (similarity $>$ a threshold τ), and maximize the sum of the scores of the selected elements. [23] considers diversification, but it does not consider result summarization using $*$-values (it chooses individual representative elements instead). In addition to lacking high level properties, this adapted process would possibly lose the holistic picture since some low-valued elements may be assigned to the chosen representatives from the top elements.

DisC diversity [7]: Drosou and Pitoura [7] proposed *DisC* diversity: given a set of elements S, the goal is to output a subset S' of smallest size such that all the elements in S are *similar to* at least one element in S' (*i.e.*, have distance at most a given threshold τ), whereas no two elements in S' are similar to each other (distance is $> \tau$). Here diversification can be achieved similar to [23]. However, it ignores the values or relevance of the elements (unlike us or [23]), and has no bound on the number of elements returned (unlike us, [23, 18]). Therefore this approach may not be useful when the user wants to investigate a small set of answers, and it does not provide a summary of common properties of high valued tuples.

Classification and clustering: Classification and clustering have been extensively studied in the literature. Various classification algorithms like Naive Bayes Classifier [20] and decision trees [24] are widely used and are easy to implement. A simpler variation of our problem—separating top-L elements from others—can be cast as a classification problem. However, this formulation would completely ignore values of elements outside top L, whereas our problem considers all element values and uses the top-L elements only as a coverage constraint. One could also formulate the problem of clustering the top-L elements and apply the standard k-means algorithm [14] and its variants (*e.g.*, [15, 33]). However, such algorithms do not produce clusters with simple and concise descriptions, and their clustering criteria do not consider values of elements outside top L. Therefore, it is necessary to find a new approach other than traditional clustering and classification.

Other work on result diversification, summarization, and exploration: Diversification of query results has been extensively studied in the literature for both query answering in databases and other applications [4, 2, 12, 41, 37, 11, 36, 8, 40, 25, 3, 1, 27, 6,

32, 23, 7, 10, 39, 31, 38, 26, 17, 16]. These include the *MMR (Maximal Marginal Relevance)*-based approaches, the *dispersion* problem studied in the algorithm community, *diverse skyline*, summarization in *text and social networks*, relational data summarization and OLAP data cube exploration among others. The MMR-based and dispersion approaches consider diversification of results, outputting a small, diverse subset of relevant results, but do not summarize all relevant results. Others focus on various application domains and all have problem definitions different from this work.

3. PRELIMINARIES

Let S denote the result relation of some query Q. We assume that the schema of S consists a set of m *grouping attributes*, denoted where $\mathcal{A}_{groupby} = \{A_1, A_2, \cdots, A_m\}$, as well as a real-valued *score or value attribute*, signifying relevance or importance of each result tuple with respect to Q. A common example is when S is produced by an aggregation query Q of the following form:

```
SELECT A_groupby, aggr AS val
FROM R     -- base relation or output of a subquery
GROUP BY A_groupby
-- optional HAVING condition
ORDER BY val DESC
```

Here, R can be a base relation or the output of a complex subquery involving multiple tables; $aggr$ can be any SQL aggregate expression that outputs a real number (involving, e.g., `COUNT`, `SUM`, `MAX`, `MIN`, `AVG`); Example 1.1 is a case where the aggregate is `AVG`.[3] We call the tuples of S *original elements*, and suppose there are n of them. Even if the number of grouping attributes m is small, n might be large due to large domains of the participating attributes. Therefore, a user is often interested in the top L original elements (denoted by S^*_L) with the highest scores.

Clusters. To display a solution with relevance, diversity, and coverage, our output is provided in *two layers*: the top layer displays a set of *clusters* that hide the values of some attributes by replacing them with *don't-care* ($*$) values, and the second layer contains the original elements covered by them.

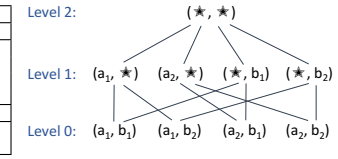
For every original element t in the output S of Q, let $\texttt{val}(t)$ denote the value or score of t. Other than the value, each $t \in S$ has m attributes $A_1, \cdots, A_m$ with active domains $D_1, \cdots, D_m$ respectively. A *cluster* C on S has the form: $C \in \prod_{i=1}^{m} D_i \cup \{*\}$. Let $\mathcal{C}$ denote the set of all clusters for relation S. We assume that the m attributes $A_1, \cdots, A_m$ have a predefined order, and therefore we omit their names to specify a cluster. For instance, for $m = 4$ attributes A_1, A_2, A_3, A_4, $(a_1, b_1, *, *)$ implies that $(A_1 = a_1) \wedge (A_2 = b_1)$, and the values of A_3 and A_4 are don't-care ($*$). We denote the value of an attribute A_i of C by $C[A_i]$; where $C[A_i] \in D_i \cup \{*\}$, $i \in [1, m]$. In particular, each element t in S also qualifies as a cluster, which is called a *singleton cluster*.

A cluster C *covers* another cluster C' if $\forall i \in [1, m]$, $C[A_i] = *$ or $C[A_i] = C'[A_i]$. Since each element t in S is also a cluster, each cluster C covers some elements from S. Further, the notion of coverage naturally extends to a subset of clusters $\mathcal{O}$. For $C \in \mathcal{C}$, $\texttt{cov}(C) \subseteq S$ denotes the elements covered by C, and for $\mathcal{O} \subseteq \mathcal{C}$, $\texttt{cov}(\mathcal{O}) \subseteq S$ denotes the elements covered by at least one cluster in $\mathcal{O}$, *i.e.*, $\texttt{cov}(\mathcal{O}) = \cup_{C \in \mathcal{O}} \texttt{cov}(C)$. Figure 3a shows two clusters

[3]While we primarily work with group-by-having aggregate queries in this paper, there is *no restriction* on the form of the SQL queries supported, other than the requirement that the result relation has a score attribute in addition to other attributes that can be used for grouping. Our algorithms and system can work with any SQL query result bearing this structure. They also apply directly to cases where the scores do not come from SQL queries, e.g., when they are specified manually by domain experts or computed by another procedure outside the database system.

	A	B	C	D
C_1	$*$	$*$	c_1	d_1
	a_1	b_2	c_1	d_1
	a_1	b_3	c_1	d_1
	a_1	b_4	c_1	d_1
	a_2	b_1	c_1	d_1
C_2	a_2	b_1	$*$	d_1
	a_2	b_1	c_1	d_1
	a_2	b_1	c_4	d_1

(a)



(b)

Figure 3: (a) Example clusters, and (b) semilattice on clusters.

$C_1 = (*, *, c_1, d_1)$, $C_2 = (a_2, b_1, *, d_1)$, and the elements they cover. Note that two clusters may have overlaps in elements they cover. Here C_1, C_2 have overlap on the tuple (a_2, b_1, c_1, d_1).

Distance function. While the distance between two elements is straightforward (the number of attributes where their values differ[4], the distance between two clusters has several alternatives due to the presence of the don't care ($*$) values. We define the distance between two clusters as the number of attributes where they do not have the same value from the domain. The distance function can be shown to be a metric and it exhibits monotonicity property (discussed in Section 4) that we use in our algorithms.

DEFINITION 3.1. *The distance $d(t, t')$ between two elements t, t' is the number of attributes where their values differ, i.e., $d(t, t') = |\{i \in [1, m] : t[A_i] \neq t'[A_i]\}|$. The distance between two clusters C, C' is the number of attributes where either (i) at least one of the values is $*$, or (ii) the values are different in C, C': $d(C, C') = |\{i \in [1, m] : C[A_i] = *, \text{or}, C'[A_i] = *, \text{or}, C[A_i] \neq C'[A_i]\}|$.*

In Figure 3a, the distance between $C_1 = (*, *, c_1, d_1)$ and $C_2 = (a_2, b_1, *, d_1)$ is 3 due to the presence of $*$-s in A_1, A_2, A_3. Intuitively, the distance between two clusters is the maximum possible distance between any two elements that these two clusters may contain, and therefore is measured by counting the number of attributes where they do not agree on a value from the domain. The distance function can also be explained in terms of similarity measures between two tuples or clusters: if the distance between two clusters is $\geq D$, then the number of common attribute values between them is $\leq m - D$ where m is the total number of attributes.

4. FRAMEWORK

In this section, we formally define the optimization problem (Section 4.1), discuss the semilattice structure and properties of the clusters (Section 4.2), and give the complexity of the optimization problem (Section 4.3).

4.1 Optimization Problem Definition

Let $\texttt{avg}(C)$ denote the average value of all elements contained in a cluster C (*i.e.*, $\texttt{avg}(C) = \frac{\sum_{t \in \texttt{cov}(C)} \texttt{val}(t)}{|\texttt{cov}(C)|}$), and $\texttt{avg}(\mathcal{O})$ denote the average value of the elements covered by a set of clusters $\mathcal{O}$.

DEFINITION 4.1. *Given relation S with original tuples and their values, size constraint k, coverage constraint L, distance constraint D, and set $\mathcal{C}$ of possible clusters for S, a subset $\mathcal{O} \subseteq \mathcal{C}$ is called a* feasible solution *if all the following conditions hold: (1) **(Size** k**)** The number of clusters in $\mathcal{O}$ is at most k, i.e., $|\mathcal{O}| \leq k$. (2) **(Coverage** L**)** $\mathcal{O}$ covers all top-L elements in S, i.e., $S^*_L \subseteq \texttt{cov}(\mathcal{O})$. (3) **(Distance** D**)** The distance between any two clusters C_1, C_2 in $\mathcal{O}$ is at least D, i.e., $d(C_1, C_2) \geq D$. (4) **(Incomparability)** No clusters in $\mathcal{O}$ cover any other cluster in $\mathcal{O}$ (equivalently, the clusters should form an* antichain *in the semilattice discussed*

[4]In this paper we focus on categorical attributes; other distance functions suitable for numeric attributes is a direction for future work (Section 9).

in Section 4.2). The objective (called `Max-Avg`*) is to find a feasible solution $\mathcal{O}$ with maximum average value* $\texttt{avg}(\mathcal{O})$.

The first three conditions in the above definition correspond to the input parameters, whereas the last condition eliminates unnecessary information from the returned solution. All these three parameters, k, D, and L, are optional and can have a default value; *e.g.*, the default value of k can be n, if there is no constraint on the maximum number of clusters that can be shown. If maintaining diversity in the answer set is not of interest, then D can be set to 0. Similarly, if coverage is not of interest, L can be set to 0 (to display a set of clusters with high overall value), or 1 (to cover the element with the highest value in S), or to k (to cover the original top-k elements from S). To maintain all the constraints, the chosen clusters may pick up some *redundant elements* $t \notin S_L*$ that do not belong to the top-L elements.

The optimization objective, called `Max-Avg`, intuitively highlights the important attribute-value pairs across all tuples with high values in S, even if they are outside the top-L elements.[5] In any solution, the value of each covered element contributes only once to the objective function, hence the selected clusters in $\mathcal{O}$ do not get any benefit by covering the elements with high value multiple times. In fact, the optimal solution when $D = 0$ and $k \geq L$ is obtained by selecting top-k original elements. The optimal solution considers the average value instead of their sum since otherwise, always the *trivial solution* $(*, *, \cdots, *)$ covering all elements and satisfying all constraints will be chosen.

4.2 Semilattice on Clusters and Properties

A *partially ordered set* (or, *poset*) is a binary relation $\leq$ over a set of elements that is reflexive ($a \leq a$), antisymmetric ($a \leq b, b \leq a \Rightarrow a = b$), and transitive ($a \leq b, b \leq c \Rightarrow a \leq c$). A poset is a *semilattice* if it has a *join or least upper bound* for any non-empty finite subset. The coverage of elements described in Section 3 naturally induces a semilattice structure on our clusters $\mathcal{C}$, where for any two clusters $C, C' \in \mathcal{C}$, $C \leq C'$ if and only if C' covers C, *i.e.*, $\texttt{cov}[C] \subseteq \texttt{cov}[C']$. If $C \leq C'$, then C' is called an *ancestor* of C in the semilattice, and C is a *descendant* of C'. Equivalently, if a cluster C_{up} covers another cluster C_{down} by replacing exactly one attribute value of C_{down} by the don't care value ($*$), then we draw an edge between them, and put C_{up} at one level higher than C_{down} in the semilattice (this gives a *transitive reduction* of the poset). Level ℓ of the semilattice is the set of clusters with exactly $\ell *$ values. Figure 3b shows the semilattice structure of $\mathcal{C}$ that has two attributes A_1 and A_2, where the domains are $D_1 = \{a_1, a_2\}$ and $D_2 = \{b_1, b_2\}$. The distance function described in Section 3 has a nice monotonicity property that we use in devising our algorithms in Section 5 (proof is in [34]):

PROPOSITION 4.2. **(Monotonicity)** *Let $\mathcal{O}$ be a set of clusters. Let λ be the minimum distance between any two clusters in $\mathcal{O}$ as defined in Definition 3.1, i.e. $\lambda = \min_{C,C' \in SC} d(C, C')$. Let $SC' = (SC \setminus \{C_1\}) \cup \{C_2\}$, where a cluster C_1 is replaced by another cluster C_2 such that C_2 covers C_1 (i.e., C_2 is an ancestor of C_1 in the semilattice). Let λ' be the minimum distance in SC', i.e., $\lambda' = \min_{C,C' \in SC'} d(C, C')$. Then $\lambda' \geq \lambda$.*

Assuming the semilattice structure in Figure 3b, note that $\{(a_1, b_2), (*, b_1)\}$ satisfies the distance constraint for $D = 2$. If we replace (a_1, b_1) by one of its ancestors $(a_1, *)$, the new two clusters $\{(a_1, *), (*, b_1)\}$ also satisfies the constraint for $D = 2$.

[5] We also investigated an alternative objective called `Min-Size` that minimizes the number of redundant elements. However it may miss some interesting global properties covering many high-valued elements in S, and is less useful for summarization.

4.3 Complexity Analysis

The optimization problem can be solved in polynomial time in *data complexity* [30] if the size limit k is a constant. This is because we can iterate over all possible subsets of the clusters of size at most k, check if they form a feasible solution, and then return the one with the maximum average value. However, this does not give us an efficient algorithm to meet our goal of interactive performance. For example, if $k = 10$, the domain size of each attribute is 9, and the number of attributes is 4, the number of clusters (say N) can be 10^4, and the number of subsets will be of the order of $N^k = 10^{40}$.

When k is variable, the complexity of the problem may arise due to any of the four factors in Definition 4.1: the size constraint k, the coverage parameter L, the distance parameter D, and the incomparability requirement that the output clusters should form an antichain. Due to multiple constraints, it is not surprising that in general, even checking if there is a non-trivial feasible solution is NP-hard. In particular, when $k \leq L$, simply the requirement of covering L original elements by k clusters in a feasible solution lead to NP-hardness without any other constraints. However, in the case when $k \geq L$ (the user is willing to see L clusters), the decision and optimization problems become relatively easier.

In the full version [34] we show the following: (1) $k \geq L$, $D = 0$: Top-k elements give the optimal solution, since adding any redundant element worsens the `Max-Avg` objective. (2) $k \geq L$, arbitrary D: A non-trivial feasible solution always exists, since we can pick arbitrary ancestors of each top-L element from level $D-1$ satisfying all the constraints. However, the optimization problems are NP-hard. (3) $k < L$, $D = 0$: Even checking whether a non-trivial feasible solution exists is NP-hard. (4) $k < L$, arbitrary D: The same hardness as above holds.

Although the optimization problem shows similarity with set cover, for a formal reduction, we need to construct an instance of our problem by creating a set of tuples and ensure that the 'sets' in this reduction conform to a semi-lattice structure. To achieve this, we give reductions from the **tripartite vertex cover** problem that is known to be NP-hard [19], and construct instances S with only $m = 3$ attributes. The NP-hardness proof the optimization problem for $k \geq L$ is more involved than the NP-hardness proof for the decision problem for $k < L$, since in the former case the coverage constraint with $k \geq L$ does not lead to the hardness.

5. ALGORITHMS

Given that the optimization problem for the case $k \geq L$, and even the decision problem for the case $k < L$, are NP-hard, we design efficient heuristics that are implemented in our prototype and are evaluated by experiments later. Not only finding provably optimal solutions for our objectives is computationally hard, but designing efficient heuristics for these optimization problems is also non-trivial. The optimization problem in Definition 4.1 has four orthogonal objectives for feasibility: incomparability, size constraint k, distance constraint D, coverage constraint L. In addition, the chosen clusters should have high quality in terms of their overall average value. In Section 5.1, we discuss the `Bottom-Up` algorithm that starts with L singleton clusters satisfying the coverage constraint, and merges clusters greedily when they violate the distance, incomparability, or the size constraints. Then in Section 5.2, we discuss an alternative to `Bottom-Up` that we call the `Fixed-Order` algorithm that builds a feasible solution incrementally considering each of the top-L elements one by one. In general, `Bottom-Up` gives better quality solution and as discussed in Section 6, is amenable to processing of multiple parameter settings as precomputation, whereas `Fixed-Order` is more efficient, hence in Section 5.3 we describe a `Hybrid` algorithm combining these two.

Algorithm 1 The `Bottom-Up` algorithm

Input: Size, coverage, and distance constraints k, L, D
1: $\mathcal{O}$ = set of L singleton clusters with the top-L elements.
2: */* First phase to enforce distance */*
3: **while** $\mathcal{O}$ has two clusters with distance $< D$ **do**
4: Let P_D be the pairs of clusters in $\mathcal{O}$ *at distance* $< D$.
5: Perform `UpdateSolution`$(\mathcal{O}, P_D)$.
6: **end while**
7: */* Second phase to enforce size limit k, almost the same as above except all pairs of clusters are considered. */*
8: **while** $|soln| > k$ **do**
9: Let P_{all} be the *all* pairs of clusters in $\mathcal{O}$.
10: Perform `UpdateSolution`$(\mathcal{O}, P_{all})$.
11: **end while**
12: **return** $\mathcal{O}$

13: **Procedure** `UpdateSolution`$(\mathcal{O}, P)$
14: **Input**: current solution $\mathcal{O}$, a set of pairs P of clusters
15: $(C_1, C_2) =$ `argmax`$_{(C_1,C_2)\in P}$`avg`$(\mathcal{O} \cup \mathrm{LCA}(C_1, C_2))$
16: Perform `Merge`$(\mathcal{O}, C_1, C_2)$.

5.1 The Bottom-Up Greedy Algorithm

Here we start with L singleton clusters with the top-L elements as our current solution $\mathcal{O}$, which satisfies the coverage and incomparability constraints, but may violate size and distance constraints. Then we iteratively merge clusters in two phases: the *first phase* ensures that no two clusters in $\mathcal{O}$ are within distance D of each other, the *second phase* ensures that the number of clusters is k or less. The following invariants are maintained by the algorithm at all time steps: (1) *(Coverage)* Clusters in $\mathcal{O}$ cover the top-L answers. (2) *(Incomparability)* No cluster in $\mathcal{O}$ covers another. (3) *(Distance)* The minimum distance among the pairs of clusters in $\mathcal{O}$ never decreases. During the execution of the algorithm, the only operation is *merging of clusters*, therefore, the coverage invariant above is always maintained. Further, the `Merge` procedure described below maintains the incomparability invariant.

The `Merge`$(\mathcal{O}, C_1, C_2)$ procedure. Given two clusters $C_1, C_2 \in \mathcal{O}$, the `Merge`$(\mathcal{O}, C_1, C_2)$ procedure replaces C_1, C_2 by a new cluster $C_{new} = \mathrm{LCA}(C_1, C_2)$, their *least common ancestor*, and also removes any other cluster in $\mathcal{O}$ that is also covered by C_{new}. $\mathrm{LCA}(C_1, C_2)$ is computed simply by replacing by $*$ any attribute whose values in C_1, C_2 differ. For instance, the LCA of $(a_1, *, c_1, *)$ and $(a_1, b_2, c_2, *)$ is $(a_1, *, *, *)$. Further, if another cluster $(a_1, b_3, *, *)$ belongs to $\mathcal{O}$, `Merge` would also remove this cluster, since it is covered by $(a_1, *, *, *)$.

In addition to maintaining the coverage condition, the merging process does not add any new violations to the distance condition in $\mathcal{O}$. This follows from the monotonicity of the distance condition given in Proposition 4.2. However, due to the merging process, the value of the solution may decrease, since $\mathrm{LCA}(C_1, C_2)$ covers all the elements covered by C_1, C_2 and all other clusters that are removed from $\mathcal{O}$, and can potentially cover some more.

The bottom-up algorithm is given in Algorithm 1. The `UpdateSolution`$(\mathcal{O}, P)$ procedure used in this algorithm takes the current solution $\mathcal{O}$ and a set of pairs of clusters P to be considered for merging, and greedily merges a pair. The first and second phases of Algorithm 1 are very similar, the only difference being the pairs of clusters P they consider for merging. In the first phase, only the pairs with distance $< D$ are considered, whereas in the second phase, all pairs of clusters in $\mathcal{O}$ are considered for merging.

We also implemented and evaluated other variants of bottom-up algorithms: (i) when we start at the clusters at level $D-1$ (instead of individual top-L tuples that satisfy the distance constraint), and (ii) when we greedily merge pairs C_1, C_2 with maximum value of `avg`$(\mathrm{LCA}(C_1 C_2))$ (instead of maximum average value of the overall solution after merging). Both these variants had efficiency and quality comparable or worse than the basic `Bottom-Up` algorithm as observed in our experiments.

5.2 The Fixed-Order Greedy Algorithm

The `Fixed-Order` algorithm maintains a set of clusters $\mathcal{O}$, and considers top-L elements in descending order by value. It decides whether the next element is already covered by an existing cluster in $\mathcal{O}$ or can be added as it is (satisfying D and k constraints); otherwise `Fixed-Order` merges the element with one of the existing clusters in greedy fashion. All constraints (k, D, and incomparability of clusters) are maintained after each of the top-L is processed, so at the end the coverage on top-L is satisfied too. `Fixed-Order` considers a smaller solution space than `Bottom-Up`, since it processes each top-L element in an online fashion, and therefore may return a solution with worse value. However, instead of all pairs of initial clusters (quadratic in number of clusters) it considers each cluster only once (linear), resulting in better running time than `Bottom-Up`. Details and pseudocode for `Fixed-Order` are shown in the full version[34].

We also consider two variants of `Fixed-Order` and evaluate them later in experiments: i) *k-means*-`Fixed-Order`, where we first run the k-means clustering algorithm [14] (with random seeding) on the top L elements, find the minimum pattern covering all elements in each of the resulting clusters, and make `Fixed-Order` process these k patterns first before moving on to the top L elements (in descending-value order); ii) *random*-`Fixed-Order`, where we first pick k element at random from the top L elements to process first, before moving on to the remaining top L elements (still in descending-value order). Both variants introduce some randomness in the results, and *k-means*-`Fixed-Order` has considerable higher initial processing overhead. However, as we shall see in Section 7, they do not produce higher-quality results.

5.3 The Hybrid Greedy Algorithm

`Bottom-Up` tends to produce results with higher quality than `Fixed-Order`, and can process multiple k, D values at the same time as discussed in Section 6, but usually requires more iterations than `Fixed-Order`. In order to get a good trade-off, we introduce the hybrid algorithm with two phases - the `Fixed-Order` phase and `Bottom-Up` phase. For a given k, L, and D, the first phase for `Hybrid` is the same as `Fixed-Order`, but with a larger number of $c \times k$ initial singleton clusters with constant $c > 1$. After covering all top-L elements in $c \times k$ clusters, `Hybrid` goes into the `Bottom-Up` phase to reduce the number of clusters from $c \times k$ to k using the `Merge` procedure that can collect redundant elements. Like `Bottom-Up`, `Hybrid` also helps in incremental computation for different choices of parameters as discussed in the next section.

6. PARAMETER SELECTION

One of the main challenges in a system with multiple input parameters is choosing the input parameter combination carefully to help the user explore new interesting scenarios in the answer space. To help the user choose interesting values of k, L, D, we provide an overall view of the values of the solutions (average value of all element covered by the clusters chosen by our algorithm) that at the same time precomputes the results for certain parameter combinations and helps in interactive exploration. In Section 6.1 we describe the visualization facilitating parameter selection, in Section 6.2 we discuss how the precomputation is achieved to plot these graphs, and in Section 6.3 we discuss a number of optimizations for interactive performance of our approach.

6.1 Visual Guide for Parameter Selection

Figure 2 gives an example of visualization showing the overview of the solutions that is generated for each chosen value of L, and illustrates the values of the solutions for a range of choices on D and k. The y-axis shows the average value of the tuples covered by the chosen clusters by our algorithm (Definition 4.1), the value of k (in a chosen range) varies along the x-axis, and different lines correspond to different values of D (also in a chosen range).

With the help of this visualization, the user can avoid selecting certain *uninteresting* or *redundant* parameter combinations. For example, with the visualization in Figure 2, a user can quickly see that the bottom-left region (where $k = 2, 3$) is uninteresting, with low average values. The user also sees that certain ranges of parameter settings are not worth exploring as they do not affect the solution quality or in very predictable ways: *e.g.*, for $D = 1$, the range of $k > 12$ yields almost the same solution quality, while for $k \in [2, 9]$, the quality changes predictably with k. On the other hand, the "knee points" (*e.g.*, $k = 9, 11$ for $D = 1$) suggest good choices of parameters. The visualization also reveals the trade-off between different choices of D; *e.g.*, at $k = 9$, the user can decide between a solution set with a higher value ($D = 1$) or more diversity ($D = 2$). Note that in Figure 2, curves for different D values may overlap, which suggests ranges of D values with little impact on solution quality, allowing the user to work on "bundles" of D values instead of individually. If the user cares about curves for individual D values, legends on the right are clickable to hide particular curves to reveal others that overlap.

6.2 Incremental Computation and Storage

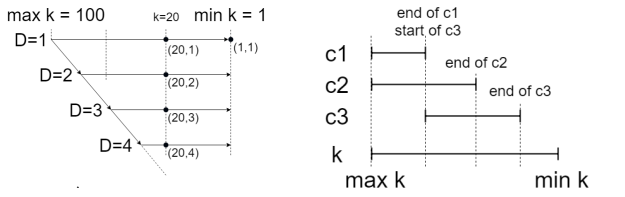
To be able to generate plots in Figure 2, one obvious approach is running an algorithm from Section 5 for all combinations of k and D given an L value. However, for interactive exploration, this approach is sub-optimal. The `Hybrid` algorithm (and `Bottom-Up`) exhibits *two levels of* incremental properties that help in computing the solutions for a range of k, D values in a batch.

In `Hybrid`, for a given value of L, the `Fixed-Order` phase outputs a set of initial clusters that can be used for all combinations of k, D, and therefore, this step can run only once. Remembering this intermediate solution, the `Bottom-Up` phase can run for all D values from the stored status. For each D, it computes results for all k values (ranging from the maximum to the minimum value) since in every round of iteration, two clusters are merged to reduce the number of clusters by one. The procedure for this incremental computation is shown in Figure 4a. In the following, we discuss how we materialize and index solutions for efficient retrieval.

Retrieval Data Structure. The computed solutions for different k, D values serve as pre-computed solutions when the user wants to inspect the solution in detail for a certain choice of k, L, D. The obvious solution for storage is to record the set of output clusters for every choice of (k, D). However, we implemented a combined retrieval data structure for storage that is both space and time efficient based on the following observation in the execution of `Hybrid` (and `Bottom-Up`) algorithm:

PROPOSITION 6.1. **(Continuity)** *Given solution cluster lists $\mathcal{O}_1, \mathcal{O}_2, ..., \mathcal{O}_r$ where r rounds are executed, for any cluster $c \in \mathcal{O}_a$ where $1 \leq a < i$, once c is removed from $\mathcal{O}_i$ at the end of round i (because of merging), for all $j > i$, $c \notin \mathcal{O}_j$.*

In other words, once a cluster is merged and therefore vanishes from the set of clusters in the solution, it never comes back. Hence, if $\mathcal{O}_{L,D,k}$ denotes the solution for a given combination of L, D, k, the set of values of k for which a cluster $c \in \mathcal{O}_{L,D,k}$ forms a continuous interval. Therefore, instead of storing the set of clusters



(a) Incremental computation structure (b) Interval structure for storing existence of clusters

Figure 4: Incremental computation and interval structure.

for all values of k and D, given an L value (where the solutions may have substantial overlap), we use an *interval tree*[5] S_D for each value of D to store the range of k for which a cluster appears in $\mathcal{O}_{L,D,k}$ storing only the `maximum` (or starting) and `minimum` (or ending) k value for this cluster (see Figure 4b). It reduces the number of solutions (sets of clusters) to be stored from $O(N_k \times N_D)$ (where N_k and N_D denote the total number of k and D values under consideration respectively) to $O(N_d)$. Further, the interval tree data structure supports efficient retrieval in time $O(\log N_k)$[5]. .

6.3 Optimizations

A number of additional optimizations are implemented to make the system efficient and interactive as described below.

Delta judgment. In every iteration (called *round*) of greedy cluster merging in the `Hybrid` (and `Bottom-Up`) algorithm, clusters are merged such that the average value of the clusters in the resulting solution is maximized using the `UpdateSolution` function (Algorithm 1). Let $\mathcal{O}_i$ be the set of clusters at the end of a round i, $T_i = \texttt{cov}(\mathcal{O}_i)$ be the tuples covered by $\mathcal{O}_i$, $v_i = \texttt{avg}(\mathcal{O}_i)$ be the average value of $\mathcal{O}_i$, and $T_c = \texttt{cov}(c)$ be the tuples covered by a given cluster c. The naive way of executing `UpdateSolution` in round $i + 1$ involves comparing the tuple list T_c of a given cluster c ($= LCA(C_1, C_2)$ as mentioned in Algorithm 1) and the current set of covered tuples T_i, finding out new tuples in $T_c \setminus T_i$ to obtain $T_i \cup T_c$ as potential T_{i+1}, and recalculating the objective $\texttt{avg}(T_i \cup T_c)$ based on the new tuples. However, it takes a huge amount of time doing all the tuple-wise comparison for all possible clusters that are eligible to be merged in this round. Instead, we incrementally keep track of the marginal benefit (as sum and count to compute the average) that a cluster c brings to the new solution $\mathcal{O}_{i+1}$ compared to $\mathcal{O}_i$ as follows (pseudocode in Algorithm 2).

The basic idea is that the improvement in the total average value that a cluster c brings to solution $\mathcal{O}_i$ is due to the tuples in $T_c \setminus \mathcal{O}_i$, and that it brings to $\mathcal{O}_{i-1}$ is due to the tuples in $T_c \setminus \mathcal{O}_{i-1}$. The difference can be computed by keeping track of the new tuples that appear in $T_i \setminus T_{i-1}$, and comparing them with the tuples in T_c. In addition, we incrementally store $\Delta_{i,c,sum}$ and $\Delta_{i,c,count}$ (the sum of values and the count of tuples in $T_c \setminus T_i$, incrementally computed from $\Delta_{i-1,c,sum}, \Delta_{i-1,c,count}$). Hence the tentative new average value of the solution $\mathcal{O}_{i+1}$ if we add c to $\mathcal{O}_i$ can be computed as $v_{i+1} = \frac{v_i \times |T_i| + \Delta_{i,c,sum}}{|T_i| + \Delta_{i,c,cnt}|}$. This optimization evaluates the `UpdateSolution` procedure efficiently since the above computations need comparisons between (i) the list containing $T_i \setminus T_{i-1}$ and (ii) T_c, and $T_i \setminus T_{i-1}$ is likely to be much smaller than T_i. This gives $30x$ speedup in our experiments.

Cluster generation and mapping to tuples. The semilattice structure on the clusters given an L value is required to run our algorithms that may contain a number of clusters in a naive implementation. To reduce this space to contain only the relevant clusters, clusters are first generated by each tuple in top-L, which ensures that each generated cluster is a possible cluster covering at least one tuple in top-L. Besides, we need to maintain

Algorithm 2 The `Delta-Judgment` Procedure

Input: Marginal score benefit Δ_{sum}, marginal amount benefit Δ_{cnt}, round indicator i indicating when were Δ values last updated, current round $j+1$, difference list $T_{(j,j-1)} = T_j \setminus T_{j-1}$

/* $\Delta_{i-1,c,sum} = \Delta_{sum}$ and same for $\Delta_{i-1,c,cnt}$ in this case*/
v_{j+1} = new score to be calculated.
v_j = current score.
/ Marginal benefits are far outdated, -1 is default value; $i <= j-1$ means $\Delta_{i-1,c,sum}$ and $\Delta_{i-1,c,cnt}$ cannot be updated directly using $T_{(j,j-1)}$ */*
if $i = -1$ **or** $(j - i \geq 1)$ **then**
    $\Delta_{cnt} = \sum(val(T_j \setminus T_c))$
    $\Delta_{sum} = |T_j \setminus T_c|$
    / Marginal benefits were updated last round (round j) and can use $T_{(j,j-1)}$ for comparison */*
else if $i = j$ **then**
    $\Delta_{cnt} = \sum(val(T_{(j,j-1)} \setminus T_c))$
    $\Delta_{sum} = |T_{(j,j-1)} \setminus T_c|$
    / Marginal benefits were updated in the same round */*
else if $i = j + 1$ **then**
    / Do nothing. It is up-to-date. */*
end if
$v_{j+1} = (v_j \times |\mathcal{O}_j| + \Delta_{sum})/(|\mathcal{O}_j| + \Delta_{cnt}))$
/ Update the round indicator */*
$i = j + 1$
return v_{j+1}

mappings between clusters and the tuples they contain, for which tuples generate *matching expressions* for their target clusters and search through the cluster list (instead of starting with a cluster and searching for matching tuples). Experiments in Section 7.3 shows the benefit - $100x - 1000x$ speedup in running time.

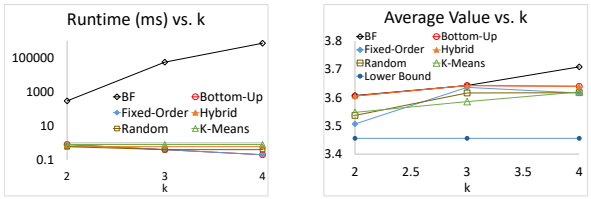
Hash values for fields. The value of an attribute is often found to be text (or other non-numeric value). While storing information on the clusters, we maintain hashmaps for each field between actual values and integer hash values, and store the hash values inside each cluster (mapped back to the original values in the output). This optimization reduces the running time of the order of $50x$.

7. EXPERIMENTS

We develop an end-to-end prototype with a graphical user interface (GUI) to help users interact with the solutions returned by our two-layered framework. The prototype is built using Java, Scala, and HTML/CSS/JavaScript as a web application based on Play Framework 2.4, and it uses PostgreSQL at the backend (screenshots of the graphical user interface can be found in the demonstration paper for our system [35]). In this section we experimentally evaluate our algorithms using our prototype by varying different parameters (Section 7.1), and then test the precomputation and guidance performance (Section 7.2). The effects of optimizations are given in Section 7.3, scalability of our algorithms for a larger dataset is discussed in Section 7.4.

Datasets. In most of the experiments, we use the MovieLens 100K dataset [29, 28, 13]. We join all the tables in the database (for movie-ratings, users, their occupation, etc) and materialize the universal table as *RatingTable*. Each tuple in this rating table has 33 attributes of three types: (a) *binary* (*e.g.*, whether or not the movie is a comedy or action movie), (b) *numeric* (*e.g.*, age of the user), and (c) *categorical* (*e.g.*, occupation of the user). We join the tables as a precomputation step to avoid any interference while measuring the running time of our algorithms.

The other dataset we use is TPC-DS benchmark [21] primarily for evaluating scalability of our algorithms. The table we materialized via generator is *Store_Sales*, which contains 23 attributes and 2,880,404 tuples in total. The aggregate queries used for these two datasets (average rating for MovieLens and average net profit for TPC-DS) can be found in [34]. All experiments were run on a 64-bit Ubuntu 14.04.4 LTS machine, with Intel Core i7-2600 CPU (4096 MB RAM, 8-core, 3.40GHz).



(a) Running time: L=5, D=3 (b) Value: L=5, D=3

Figure 5: Comparison with brute-force.

7.1 Varying Parameters

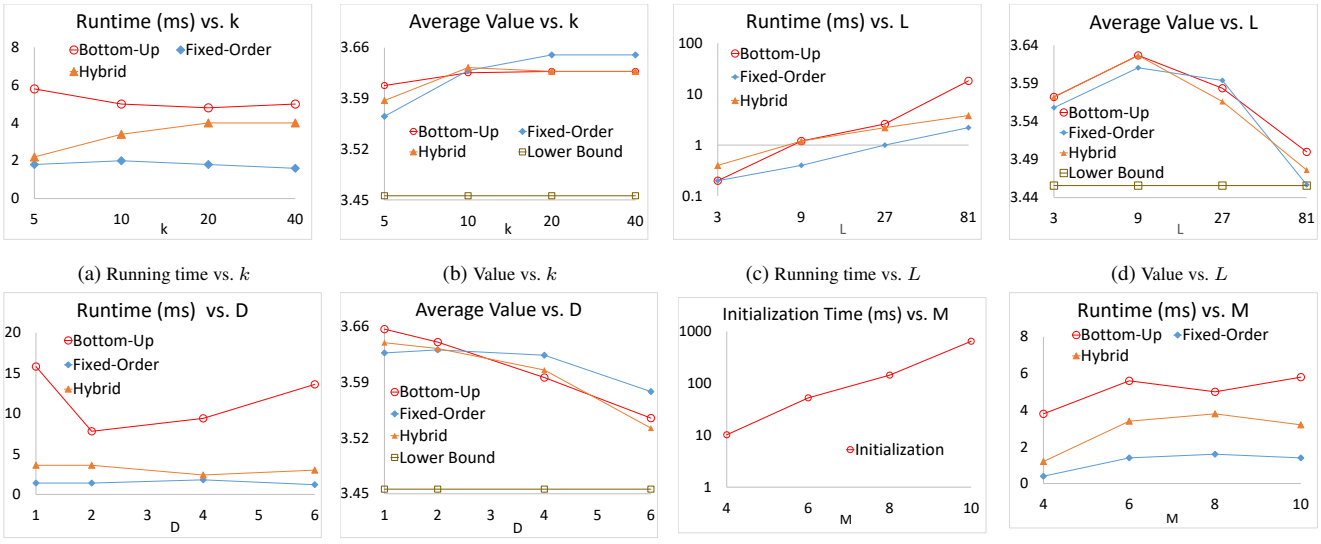
Unless mentioned otherwise, the three algorithms from Section 5 are compared in this section: (i) `Bottom-Up`, (ii) `Fixed-Order`, (iii) `Hybrid`. In the plots showing the values, we also include (iv) `Lower Bound`: value of the trivial (feasible) solution (a single cluster with don't-care $*$ values for all attributes) as a baseline.

Comparison with baselines. We compare our algorithms with two baselines: the brute-force algorithm considers all possible cluster combinations, and outputs the global optimal; the lower-bound algorithm simply returns the trivial answer containing one single cluster with all attributes as "*"s, which is always feasible for any value of k, L, D. We also consider two variants of `Fixed-Order`: random and k-means, discussed in Section 5.2. Figure 5a shows the running time for $L = 5, D = 3$ and $k = 2, 3, 4$ (lower-bound is omitted because it returns trivial answers). Even with such small parameter values, the brute-force algorithm is not practical: *e.g.*, at $k = 4$, it takes more than 2.5 hours. Figure 5b compares the average values produced by different algorithms. Since the random and k-means variants of `Fixed-Order` are randomized, we report their average values over 100 runs each. From Figure 5b, we see that the results of `Fixed-Order` and its variants are comparable with brute-force's, and are much better than the trivial solution. Another observation is that neither random nor k-means variant improves the quality of plain `Fixed-Order`. Further, they introduce more variance in the result quality (0.033 for random and 0.045 for k-means in terms of combined standard deviation), and slightly increase the running time. Therefore, in the rest of the section, we focus on the plain `Fixed-Order` algorithm.

Effect of size parameter k. Figure 6a shows the running time varying k. The running time of `Fixed-Order` is the best as it never considers more than k candidate merges per step; in contrast, `Bottom-Up` may consider a quadratic number of candidate merges per step and it is slower than `Fixed-Order` as a consequence. `Hybrid` is in the middle for runtime as expected. Furthermore, $D = 3$ helps bound the size of $\mathcal{C}_\ell$ and hence the cost of computing the set cover. The running time tends to decrease with bigger k for both `Fixed-Order` and `Bottom-Up`; the reason is that fewer merges are needed to reach the desired k. However, for `Hybrid`, since larger k makes the candidate pool larger and might bring in more calculation in the second phase (`Bottom-Up` phase), the run time for `Hybrid` tends to get closer to `Bottom-Up`.

The average value of `Fixed-Order` is lower than the value of `Bottom-Up` or `Hybrid` as explained in Section 5, although gets better with larger k in Figure 6b.

Effect of coverage parameter L. Figure 6c shows that running time of all algorithms increase as the number of elements to be covered L increases. Since `Fixed-Order` depends linearly on L, it is less affected by L, whereas `Bottom-Up` treats individual elements



(a) Running time vs. k (b) Value vs. k (c) Running time vs. L (d) Value vs. L

(e) Running time vs. D (k=10, L=40) (f) Value vs. D (k=10, L=40) (g) Initialization time vs. m (k=L=20) (h) Running time vs. m (k=L=20)

Figure 6: Experimental results varying parameters. The default values of parameters are $m = 8$, $k = 3$, $L = 40$, $D = 3$.

as clusters and may incur quadratic time w.r.t. L. For `Hybrid`, with the restriction of the size of the candidate pool determined by k, the run time increase is slower than `Bottom-Up` and is comparable with `Fixed-Order`. Note that in Figure 6d, the upper bound decreases since with L increasing, the average value of the top-L elements decreases. All three algorithms seem to be close in terms of average values, but `Bottom-Up` has the highest value most of the times and `Hybrid` usually gets results close or equal to `Bottom-Up`.

Effect of distance parameter D. In Figure 6e, `Fixed-Order` is mostly unaffected by D since the distance value is checked only once when an element is considered. `Hybrid` is relatively constant as well given that when the distance check starts, the number of unchecked tuples is limited by the candidate pool. For `Bottom-Up` as D increases, the run time drops first and then climbs. It may be caused by the existence of a balance point on number of calculations between distance insurance (phase 1 in `Bottom-Up`) and greedy merge (phase 2 in `Bottom-Up`).

The average value of the output (the value of objective function) is highest when $D = 1$ (since singleton clusters are collected for $L = k = 20$), then drops with D going up as shown in Figure 6f.

Effect of number of attributes m. Varying the number of grouping attributes m also illustrates the effect of varying input data size. Since our algorithms run on the output of an aggregate query, as m increases, our input data size $|S| = n$ is likely to increase (for the m values in Figure 6g and 6h, the size of the input ranges from 140 to 280). When a new query comes, the system performs an initialization step of constructing clusters and the semi-lattice structure. This initialization time is shown in Figure 6g. This step is performed only once per query, varying k and D does not need another initialization. Our implementation takes from $10ms$ when $m = 4$ to about $1s$ when $m = 10$. Note that this is the number of group-by attributes in the top-k aggregate query, not in the original dataset. So it is likely to have a small value ≤ 10. Figure 6h has the running time of the algorithms for $k = L = 20, D = 3$ and shows that all the algorithms return results in real time (in a few ms) after the initialization step.

7.2 Cost and Benefit of Precomputation

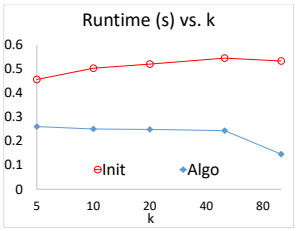
The performance evaluation for precomputation is shown by varying k, L and D separately, and comparing the running time of `Hybrid` between precomputation implementation and non-precomputation (single) implementation.

Effect of size parameter k. In this experiment, $L = 1000$, $D = 2$ and $N = 2087$ are fixed. Five k values are chosen: $5, 10, 20, 50, 100$. The running time result is shown in Figure 7a: the initialization time hardly changes with k growing since k does not affect the initialization process. Given that a larger k requires less operations in `Bottom-Up` phase to reach the target k, the running time for the algorithm (`Hybrid`) has a descending trend.

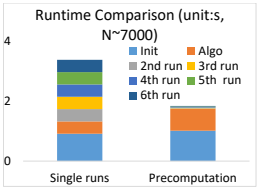
Effect of coverage parameter L. The fixed parameters are $k = 20$, $D = 2$, and $N = 2087$. Three L values are selected for the experiment:$L = 200, 500, 1000$. The running time results for single version and precomputation version are presented in Figures 7c and 7d. Both implementations have rising trend with respect to L and share similar initialization times as expected. Although under the same parameter combinations, algorithm runtime for single implementation is much lower than precomputation time in the other implementation (about 1/3 to 1/4), but the retrieval time for precomputation implementation is extremely short (tens of milliseconds), which can make up for the time in multiple runs.

Effect of total elements N. Here three parameters k, L, D are fixed as $k = 20$, $L = 500$ and $D = 2$. We varied total input elements to test the system's performance with relatively higher capacity: $N = 927, 2087$ and 6955. The running time result is shown in Figures 7e and 7f. The changing trends are similar with those in Figures 7c and 7d, but a significant increase for the initialization time can be observed with N growing. This is caused by materializing more possible clusters brought by variety of tuples.

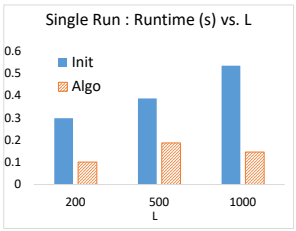
Single run vs. multiple runs. From Figure 7c, 7d, 7e and 7f, the information is enough for comparing precomputation and non-precomputation versions on both single run and multiple runs scenario - For a single run, precomputation process is unused, making precomputation version the slower and more expensive choice; For multiple runs with similar setup, the precomputation version has increasingly more benefits brought by the rapid retrieval process taking tens of milliseconds. In order to provide a quantitative comparison, we provide Figure 7b with $N = 6955$: if only a single run is required, the single version of `Hybrid` is clearly faster and cheaper than the precomputation version. However, when the third run finishes, the precomputation version is already faster; When all



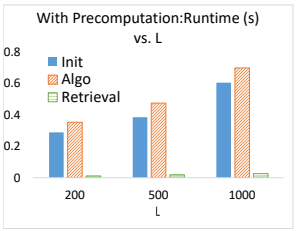
(a) Runtime vs. k



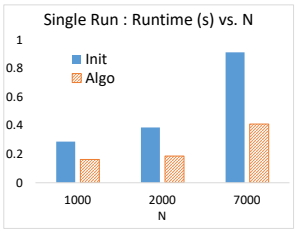
(b) Runtime comparison, $N = 7000$



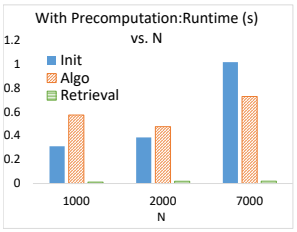
(c) Single runtime vs. L



(d) Runtime vs. L; with precomputation

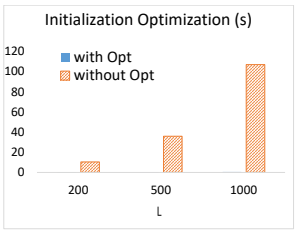


(e) Single runtime vs. N

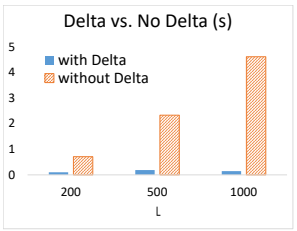


(f) Runtime vs. N; with precomputation

Figure 7: Experimental results varying parameters, and with or without precomputation



(a) Initialization running time with and without optimization, coverage L varies



(b) Algorithm running time with and without optimization, coverage L varies

Figure 8: Experiments on effects of optimizations

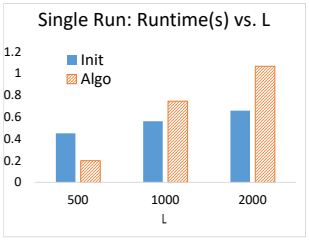
six runs finish, the single version takes about two times in terms of running time compared with the precomputation version.

Timing for Guidance Visualization. We evaluated the running time for the generation of guidance visualization under different queries. The generation times are similar among different number of attributes - 20-40 milliseconds when the number of attributes is from 4 to 10 with $N = 2087$ in MovieLens dataset, meeting the requirement for interactive performance.

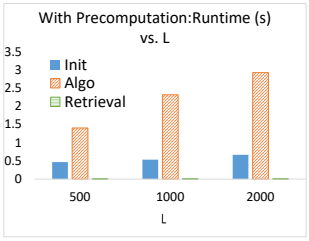
7.3 Benefit of Optimizations

Cluster generation and mapping to tuples. Since L is the only factor that affects the initialization time when the input size N is fixed, in this experiment, L varies among $200, 500$ and 1000 while others are fixed: $k = 20, D = 2, N = 2087$. The result is presented in Figure 8a. Only the running time of initialization is drawn because the optimizations in this section only affect the initialization time. The optimizations - cluster generation and cluster-tuple mapping - provide significant performance improvement by cutting down the running time from > 100s for $L = 1000$ to 0.5s.

Delta Judgment. The effect by introducing *Delta Judgment* is



(a) TPC-DS: Single running time vs. coverage L



(b) TPC-DS: With precomputation running time vs. coverage L

Figure 9: TPC-DS experimental results varying parameters and with or without precomputations.

shown in Figure 8b. Given that L is also the most effective variable to affect the running time, the experimental settings are the same as the experiment for Figure 8a. However, only the running time of the algorithm is plotted since *Delta Judgment* has no effect on the running time of initialization. The result in Figure 8b shows that the *Delta Judgment* successfully improves the algorithm's efficiency from 4.6s to 0.15s when $L = 1000$, which is the slowest case in the experiment in this section.

7.4 Scalability with a Larger Dataset

In order to evaluate the scalability of our algorithms we perform an experiment with TPC-DS dataset on *Store_Sales* table. The parameters are set to $k = 20, D = 2$ and $N = 47361$. Coverage parameter L varies among $500, 1000$ and 2000. Both the single and precomputation version are evaluated using this set of parameters. From the results shown in Figure 9a and Figure 9b, the initialization time is interactive - about 1s for the the largest parameters: $L = 2000$ and $N = 47361$. However, even for the single version, the running time of the algorithm increases to more than 1s compared with 200ms from results in Figures 7e and 7f, and for the precomputation version it increases to ~ 2.5s. Although the running time increases, the total running time (~ 3.5s) for precomputation is still interactive. Note that the size of the answers (N) output by a query is likely to be much smaller than the size of the dataset, even for a big dataset.

8. USER STUDY AND SURVEY

We conducted a user study with the following high-level goals: (1) to compare our approach with an alternative that adapts decision trees [24] and (2) to evaluate the utility of user-specified parameters in our approach. Specifically, we want to know: (1) whether our new problem formulation provides any advantage over adapting existing methods to the same usage scenarios; and (2) whether allowing user-specified parameters in our problem formulation is warranted in order to capture the range of different usage scenarios and/or user preferences. In addition, we informally solicited feedback during the demonstration of our system at *SIGMOD* 2018 [35] to assess the effectiveness of our interactive features in Sections 6.

8.1 User Study Setup

Dataset and queries. All data are drawn from the *MovieLens RatingTable* as described in Section 7. Queries are based on the same aggregate query template introduced as in Example 1.1, with an additional `WHERE` condition and variations in query constants and group-by attributes across user tasks.

Adapted decision tree. As discussed in Section 2, no existing method suits our problem setting. After exploring various possibilities, we decided to adapt the method of decision trees [24] as it offers the closest match with our application scenarios. The structure of a decision tree naturally induces summaries of top-L

tuples in the form of predicates, which are easier for users to interpret than other classifiers. It is also discriminative, as opposed to simply running clustering algorithms over the top-L tuples while ignoring low-value tuples. We use the standard implementation provided by Python's `scikit-learn` package [22]; we tune the height of the decision tree such that the number of "positive" leaf nodes (wherein top-L tuples are the majority) as close as possible to, but no greater than, k. Note that the cluster patterns under this approach can be more complex than ours, as they may involve non-equality comparisons and negations. This additional complexity increases discrimination, but makes the patterns more difficult to interpret and internalize—a hypothesis we shall test with our study.

Tasks. Each study subject is asked to carry out three groups of tasks (*task groups*): (i) *varying-method*, (ii) *varying-k*, and (iii) *varying-D*. The first group is designed to compare our approach and decision trees. The last two are designed to evaluate the utility of making parameters k and D in our approach specifiable by users.[6] To account for the possible learning effect, we sequence the task groups differently among study subjects—half go through the sequence varying-(method, k, D), while the remaining go through varying- (k, D, method).

Before each task group, we familiarize the subject with the aggregate query result as well as the tasks; Then, we give the subject a series of questions, organized into three sections in order. Each question asks the subject to classify a given tuple, whose value is hidden, into one of three categories: "top" (value among the top L of all tuples), "high" (value above or equal to the average, but outside the top L), and "low" (value below average). The three sections are based on the same "working set" of clusters, but differ in the information the subject can access:

- *Patterns-only*, 6 questions: The subject can see the clusters and their associated patterns, but not the membership within clusters or the table of all query result tuples. This section is designed to test how well the cluster patterns help users understand the data.
- *Memory-only*, 6 questions: The subject cannot access any information; all questions must be answered from memory. This section is designed to test the extent to which users can internalize the insights learned from the cluster patterns for later use. We ensure that these six tuples are distinct from those chosen before.
- *Patterns+members*, 8 questions: The subject can see the clusters patterns as well as the covered result tuples. This section is designed to test how our full-fledged cluster UI can help user explore data. The 8 tuples are chosen and reordered randomly from the 12 tuples used in the previous two sections.

After these three sections, we present two sets of clusters: one is the working set, the other is obtained under a different setting (but for the same aggregate query and L) for comparison. We then ask the subject to choose a preferred set for the tasks just performed. For a *varying-method* task group, the cluster to compare is produced by decision trees, under the same k setting (D does not apply to decision trees); For a *varying-k* task group, the cluster to compare is produced by our approach under another k, while other parameters remain the same; For a *varying-D* task group, the cluster to compare is produced by our approach under another D.

Participants and assignment of tasks. There are 16 participants - 14 of them are graduate students at Duke University (12 in computer science and 2 others), while the remaining 2 are Duke undergraduates. They all have some prior experience working with tabular data and are capable of handling tasks in our user study.

[6]We do not evaluate the utility of making L user-specifiable, as it should be evident that what "top" tuples mean depends on the situation—e.g., a small L means interest in characterizing really high-valued tuples, while a larger L means interest in tuples whose values are "good enough."

Recall that each of the three task groups compares two sets of clusters. There are $2^3 = 8$ possible assignments in total. We assign two subjects to each of these 8 possibilities, each goes through one of the two task group sequences. Finally, we ensure that tuples in our questions are equally distributed among all subjects.

Metrics. We record the time for each subject to complete each of the three sections in each of the three task groups. We evaluate the accuracy of answers using the standard accuracy measure of $\frac{TP+TN}{TP+FP+FN+TN}$ based on confusion matrices [9], and we define two variants: *T-accuracy* focuses on discerning the top tuples from the rest, where "positive" means being in top L; *TH-accuracy* focuses on discerning the top and high tuples from the low ones, where "positive" means being in either top or high category.

8.2 User Study Results

Table 1 summarizes both the quantitative results (subjects' performance in terms of time and accuracy for classifying tuples into categories) and qualitative results (subjects' preferences between the clustering outputs compared) of our user study.

Varying-method task group. For this task group, we set $L = 50, k = 10, D = 1$ for our approach, and $L = 50, k = 10$ for the method based on decision trees. For this scenario, tree depth of 7 gives exactly 10 positive leaf nodes.

First, note that among the three sections, memory-only is the fastest, patterns-only is considerably slower, and patterns+members is the slowest. This observation holds both for our approach and for decision trees (as well as under each setting of other task groups). This universal trend can be intuitively explained by the fact that users tend to spend more time on a question if more information is presented to them.

As for accuracy, patterns+members has the highest accuracy, and patterns-only is usually no worse than memory-only. This trend also makes intuitive sense as users are generally able to achieve higher accuracy if aided with more information. Across settings, patterns+members is always nearly perfect, as expected.

Comparing our approach and decision trees in terms of time spent by study subjects, our approach is consistently faster over the three sections. The biggest advantage is seen in the patterns-only section, suggesting that our patterns are much easier to apply. The advantage is less pronounced in the other two sections. For patterns+members, a possible explanation is that users spend bulk of the time examining detailed memberships. For memory-only, our conjecture is that decision tree patterns are so difficult to recall that our subjects realized quickly that spending more time did not help.

In terms of accuracy, our approach is better than decision trees for the patterns-only and memory-only sections (recall that patterns+members is always nearly perfect across settings). It is understandable for decision trees to have lower TH-accuracy, because they are trained to separate only the top tuples from the rest, while our approach considers the values of all tuples covered by the patterns. On the other hand, while the T-accuracy for decision trees is good for patterns-only, it drops significantly for memory-only, because decision tree patterns are difficult for users to memorize. In comparison, the accuracy of our approach degrades very little from patterns-only to memory-only, which is the evidence that users can internalize insights from our simple patterns quite well.

Finally, when asked which method they prefer, the overwhelming majority of the subjects (14/16) chose our approach over decision trees. The key reason cited was the simplicity of our patterns.

Varying-k task group. In this task group, we fix $L = 30$ and $D = 1$, and compare $k = 5$ vs. $k = 10$. Note that with the bigger k, we expect to have more clusters with more specific patterns, leading to higher discrimination but more complex summaries.

Table 1: Summary of results from the user study

	Task group	Varying-method		Varying-k		Varying-D	
		Decision tree	Our method	$k = 5$	$k = 10$	$D = 1$	$D = 3$
Patterns-only	Time/question	25.7 ± 6.6	$\boxed{23.5 \pm 6.5}$	$\boxed{19.6 \pm 6.6}$	22.5 ± 5.9	13.5 ± 3.1	$\boxed{9.6 \pm 2.9}$
	T-accuracy	0.792 ± 0.121	$\boxed{0.854 \pm 0.132}$	0.771 ± 0.139	$\boxed{0.833 \pm 0.088}$	$\boxed{0.771 \pm 0.087}$	0.750 ± 0.108
	TH-accuracy	0.646 ± 0.075	$\boxed{0.854 \pm 0.075}$	0.625 ± 0.121	$\boxed{0.708 \pm 0.121}$	0.771 ± 0.087	$\boxed{0.813 \pm 0.098}$
Memory-only	Time/question	9.6 ± 3.8	$\boxed{8.3 \pm 2.9}$	11.1 ± 4.3	$\boxed{9.8 \pm 4.2}$	8.4 ± 2.1	$\boxed{6.6 \pm 3.4}$
	T-accuracy	0.625 ± 0.150	$\boxed{0.792 \pm 0.121}$	$\boxed{0.771 \pm 0.139}$	0.667 ± 0.125	0.646 ± 0.116	$\boxed{0.667 \pm 0.108}$
	TH-accuracy	0.625 ± 0.174	$\boxed{0.792 \pm 0.121}$	$\boxed{0.667 \pm 0.108}$	0.625 ± 0.121	0.771 ± 0.139	$\boxed{0.875 \pm 0.083}$
Patterns+members	Time/question	22.9 ± 4.2	23.7 ± 2.4	$\boxed{21.0 \pm 6.2}$	22.5 ± 4.2	$\boxed{13.5 \pm 1.9}$	14.3 ± 3.3
	T-accuracy	0.922 ± 0.165	$\boxed{0.953 \pm 0.061}$	0.938 ± 0.088	0.953 ± 0.061	0.906 ± 0.104	$\boxed{0.953 \pm 0.061}$
	TH-accuracy	0.750 ± 0.088	$\boxed{0.844 \pm 0.104}$	0.938 ± 0.088	$\boxed{0.97 \pm 0.054}$	0.844 ± 0.054	$\boxed{0.922 \pm 0.087}$
	Overall preferred	12.5%	$\boxed{87.5\%}$	43.8%	$\boxed{56.2\%}$	37.5%	$\boxed{62.5\%}$

Table 1 shows the user study results, where times are in seconds, and accuracies are between 0 and 1; we report average and standard deviation over all subjects. Better performances (shorter times and higher accuracies) and stronger preferences are highlighted with box enclosures, unless the advantage is too small. From Table 1, we see that the bigger k leads to more time spent as long as patterns are accessible to the subjects, i.e., for patterns-only and patterns+members. However, for memory-only, the bigger k actually results in less time spent; one conjecture is that complex summaries are so difficult to recall from memory that some subjects simply stopped trying and resorted to guessing. This observation is consistent with the low accuracies seen under the bigger k for memory-only, further discussed below.

In terms of accuracy, favor turns from the smaller k to the bigger k for patterns-only and patterns+members, pointing to a clear trade-off between time and accuracy. On the other hand, for memory-only, the trend is reversed: accuracies under the bigger k drop dramatically and become lower than under the smaller k, because the subjects had trouble recalling the summaries from their memory. In comparison, under the smaller k, accuracies for memory-only are at least as good as those for patterns-only.

Finally, when asked whether they prefer the smaller or bigger k, a slight majority of the subjects preferred the bigger, but still a significant fraction $(7/16)$ preferred the smaller. There is no clear winner here, unlike the case for the varying-method task group.

Varying-D tasks. We fix $L = 10$ and $k = 7$, and compare $D = 1$ vs. $D = 3$. $D = 1$ represents a looser constraint, and in this case leads to detailed summaries and higher discriminative power; the trade-off, of course, is that patterns appear less diverse.

As we can see from Table 1, the bigger D leads to faster answer speed and higher accuracy in most cases, with just two exceptions: the smaller D is more accurate in terms of T-accuracy for patterns-only, and it is faster for patterns+members. Both can be explained by the fact that, here some clusters produced by the bigger D happen to have more general patterns and cover more tuples. Without access to cluster membership, T-accuracy would suffer because these clusters may cover some high-valued (but necessarily top-valued) tuples. With access to cluster membership, T-accuracy would not be a problem, but more tuples take longer to examine.

Although the performance results appear to favor the bigger D (looser constraint), preferences are divided. A majority of the subjects do prefer the bigger D, but still a sizable number of them $(6/16)$ prefer the smaller D, which produces more diverse patterns.

Learning effect. We assess the possible learning effect by comparing the quantitative result within one experimental sequence (varying-method first, then varying-k and varying-D), and compare with the results in Table 1. The differences are minor, and the relative ordering of approaches by performance largely stays the same, so the conclusions drawn above still stand. Because of space constraints, details are shown in our full paper [34].

8.3 Informal User Survey Results

To measure the effectiveness of the interactive feature described in Sections 6, we asked attendees who visited our demo booth at *SIGMOD* 2018 to fill out an informal survey. We received 18 responses, and the results are summarized below:

Did you find the visualizations helpful?	Yes, very much	Yes	Not that much	Not at all
For parameter selection	4	13	1	0

The vast majority of the responses are positive. Some constructive criticisms were offered too. One pointed out that the visualization for guiding interactive parameter selection still requires extensive explanation before users can understand and benefit from it. Another pointed out that instead of showing all choices of k and D in this visualization, it might be possible to use the data behind this visualization to narrow down the choices further.

8.4 Summary and Discussion

The high-level findings are: (1) our approach is more suitable to the designed tasks than the decision trees, thanks to the simplicity of our patterns by design; (2) while more specific and detailed clusters can offer better accuracy, this advantage dissipates when users no longer see the cluster patterns directly, because they are much less memorable; (3) parameters k and D affect the complexity of our clustering results and present various trade-offs (e.g., accuracy vs. efficiency), so users have different preferences.

It is also worth noting that while we did not explicitly compare with the approach of simply showing the top L tuples with no summarization at all, it be seen as an extreme case where $k = L$ and $D = 1$. Hence, the general observation we made when comparing parameter settings applies here too: showing the top L tuples alone would provide the most detailed information, but that would be very difficult to use and memorize.

9. CONCLUSIONS

In this paper, we presented a framework for summarizing and exploring high-valued query answers, considering factors such as relevance, diversity, and coverage. We studied optimization problems for these tasks, developed efficient algorithms, evaluated the approach using real and benchmark datasets, and showed that our implementation is capable of interactive exploration in real time. We also conducted a user study to demonstrate the utility of our approach. There are several directions for future work. While we mainly focused on categorical attributes, for numeric attributes one can consider other distance functions (*e.g.*, L_p norms) and description of clusters (*e.g.*, ranges). One can also consider objective functions other than average and see how our algorithms can be adapted to other objectives. Other sophisticated visualizations to better help the users are also worth exploring.

10. REFERENCES

[1] Z. Abbassi, V. S. Mirrokni, and M. Thakur. Diversity maximization under matroid constraints. In *KDD*, pages 32–40, 2013.

[2] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong. Diversifying search results. In *WSDM*, pages 5–14. ACM, 2009.

[3] A. Borodin, H. C. Lee, and Y. Ye. Max-sum diversification, monotone submodular functions and dynamic updates. In *PODS*, pages 155–166, 2012.

[4] J. Carbonell and J. Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *SIGIR*, pages 335–336, 1998.

[5] T. H. Cormen. *Introduction to algorithms*. MIT press, 2009.

[6] T. Deng and W. Fan. On the complexity of query result diversification. *ACM TODS*, 39(2):15:1–15:46, May 2014.

[7] M. Drosou and E. Pitoura. Disc diversity: result diversification based on dissimilarity and coverage. *PVLDB*, 6(1):13–24, 2012.

[8] W. Fan, X. Wang, and Y. Wu. Diversified top-k graph pattern matching. *PVLDB*, 6(13):1510–1521, 2013.

[9] T. Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[10] P. Fraternali, D. Martinenghi, and M. Tagliasacchi. Top-k bounded diversification. In *SIGMOD*, pages 421–432, 2012.

[11] O. Gkorgkas, A. Vlachou, C. Doulkeridis, and K. Nørvåg. Finding the most diverse products using preference queries. In *EDBT*, pages 205–216, 2015.

[12] S. Gollapudi and A. Sharma. An axiomatic approach for result diversification. In *WWW*, pages 381–390, 2009.

[13] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *ACM TIIS*, 5(4):19:1–19:19, Dec. 2015.

[14] J. A. Hartigan. *Clustering algorithms*. Wiley, 1975.

[15] Z. Huang. Extensions to the k-means algorithm for clustering large data sets with categorical values. *Data mining and knowledge discovery*, 2(3):283–304, 1998.

[16] H. Jagadish, J. Madar, and R. T. Ng. Semantic compression and pattern extraction with fascicles. In *VLDB*, volume 99, pages 7–10, 1999.

[17] H. Jagadish, R. T. Ng, B. C. Ooi, and A. K. Tung. Itcompress: An iterative semantic compression algorithm. In *ICDE*, pages 646–657. IEEE, 2004.

[18] M. Joglekar, H. Garcia-Molina, and A. G. Parameswaran. Interactive data exploration with smart drill-down. In *ICDE*, pages 906–917, 2016.

[19] D. C. Llewellyn, C. A. Tovey, and M. A. Trick. Erratum: Local optimization on graphs. *Discrete Applied Mathematics*, 46(1):93–94, 1993.

[20] K. P. Murphy. Naive bayes classifiers. *University of British Columbia*, 18, 2006.

[21] R. O. Nambiar and M. Poess. The making of tpc-ds. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1049–1058. VLDB Endowment, 2006.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[23] L. Qin, J. X. Yu, and L. Chang. Diversifying top-k results. *PVLDB*, 5(11):1124–1135, 2012.

[24] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[25] S. S. Ravi, D. J. Rosenkrantz, and G. K. Tayi. Heuristic and special case algorithms for dispersion problems. *Operations Research*, 42(2):299–310, 1994.

[26] S. Sarawagi. User-adaptive exploration of multidimensional data. In *VLDB*, pages 307–316, 2000.

[27] Y. Tao. Diversity in skylines. *IEEE Data Eng. Bull.*, 32(4):65–72, 2009.

[28] `http://grouplens.org/datasets/movielens/`.

[29] `http://movielens.org`.

[30] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 137–146, 1982.

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

[32] M. R. Vieira, H. L. Razente, M. C. N. Barioni, M. Hadjieleftheriou, D. Srivastava, C. Traina, and V. J. Tsotras. On query result diversification. In *ICDE*, pages 1163–1174, 2011.

[33] K. Wagstaff, C. Cardie, S. Rogers, S. Schrödl, et al. Constrained k-means clustering with background knowledge. In *ICML*, volume 1, pages 577–584, 2001.

[34] Y. Wen, X. Zhu, S. Roy, and J. Yang. Interactive Summarization and Exploration of Top Aggregate Query Answers. arXiv:1807.11634[cs.DB], July 2018.

[35] Y. Wen, X. Zhu, S. Roy, and J. Yang. Qagview: Interactively summarizing high-valued aggregate query answers. In *SIGMOD*, pages 1709–1712, 2018.

[36] D. Xin, H. Cheng, X. Yan, and J. Han. Extracting redundancy-aware top-k patterns. In *KDD*, pages 444–453, 2006.

[37] C. Yu, L. Lakshmanan, and S. Amer-Yahia. It takes variety to make a world: Diversification in recommender systems. In *EDBT*, pages 368–378, 2009.

[38] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex sql queries using automatic summary tables. In *SIGMOD*, pages 105–116, 2000.

[39] W. Zheng, X. Wang, H. Fang, and H. Cheng. Coverage-based search result diversification. *Information Retrieval*, 15(5):433–457, 2012.

[40] X. Zhu, A. B. Goldberg, J. V. Gael, and D. Andrzejewski. Improving diversity in ranking using absorbing random walks. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 97–104, 2007.

[41] C.-N. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen. Improving recommendation lists through topic diversification. In *WWW*, pages 22–32, 2005.