Interactive Summarization and Exploration of Top Aggregate Query Answers

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 (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 optimizations 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.

ACM Reference Format:

. 1997. Interactive Summarization and Exploration of Top Aggregate Query Answers. In *Proceedings of ACM Woodstock conference (WOODSTOCK'97).* ACM, New York, NY, USA, Article 4, 16 pages. https://doi.org/10.475/123_4

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:

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

WOODSTOCK'97, July 1997, El Paso, Texas USA

© 2016 Copyright held by the owner/author(s).

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

Example 1.1. *Suppose an analyst is using the movie ratings data from the* MovieLens *website [29] 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 score 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.

The obvious 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 4 tuples share the properties `(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 (or male student viewers) may or may not give high rating to the adventure genre. Therefore, *running a clustering algorithm on top-k original tuples does not work as a meaningful summary.*

In recent years, work has been done to *diversify* a set of result tuples by selecting a subset of them (discussed further in Section 8), *e.g.*, *diversified top-k* [24] takes into account diversity and relevance while selecting top result tuples; *DisC diversity* [9] takes into account similarity with the tuples that are not 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* [19] 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 or understand consecutive solutions that are two key features of our framework. In particular, we support summarization and interactive exploration

Rank	hdec	agegrp	gender	occupation	value
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 score	
1975	**20s**	**M**	**Student**	**4.24**	▼
1980	*	**M**	*	**3.96**	▼
1985	**20s**	**M**	**Programmer**	**3.86**	▼
1985	*	**M**	**Student**	**3.76**	▼

(b) Top-layer of clusters with average score

hdec	agegrp	gender	occupation	avg score	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 two-layered 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.

of aggregate answers in the following ways each posing its own technical challenges.

Summarizing Aggregate Answers with Relevance, Diversity, and Coverage. The basic operation of our framework involve *two layers*: the *top layer* shows a set of *clusters* summarizing the common properties or common attribute values of of high-valued answers, and the *bottom layer* shows the *elements* (the original answer tuples) they contain. To compute the clusters, our framework can take (up to) three parameters as input: (i) *size constraint* k denotes the number of rows to be displayed in the *top layer* of the framework ($k = 4$ 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 in the top layer, 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 as the top layer of the solution 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 (*e.g.*, (20s, M) is not chosen as discussed before), which could not be achieved by simply clustering top L tuples by k clusters.

To achieve the solution as described above, we make the following technical contributions in the paper (Section 3):

- 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 problems (both decision and optimization versions) and show NP-hardness results.
- We design efficient heuristics using properties given by the natural *semi-lattice structure* on the clusters imposed by the attribute values.
- We perform extensive experimental evaluation using the MovieLens [29] and TPC-DS benchmark [22] datasets.

Interactive Clustering and Parameter Selection. The intended application of our framework is an interactive exploration of query results where the user can gradually update D, L, or k to understand the key properties of the high-valued aggregate answers. One of the challenges in this exploration is to select useful values of k, L, D. To support this, we provide the user with an overview of the solution space. One example is shown in Figure 2 (also see Figure 6), where given selected values of $L = 8$ and $D = 2$ as in Example 1.2, how the average value of the solutions varies with different k is shown. This figure illustrates that if k is changed from $k = 3$ to $k = 2$, there will be a drop in the overall value, and even if k is set to > 5, only five clusters will be returned.

This features not only helps in guiding the user select parameter values, it also serves as a *precomputation* step to retrieve the actual solutions at an interactive speed (Section 5).

- We develop techniques for *incremental computation* to compute the solutions for multiple combinations of input parameters and store them efficiently using an *interval tree* data structure.

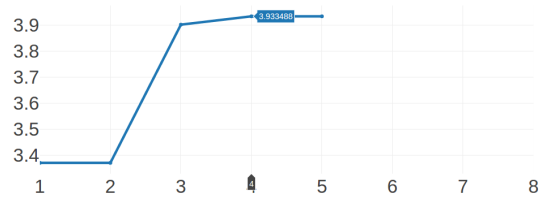


Figure 2: Visualization for parameter selection in Example 1.2: how results vary for different k when $D = 2$.

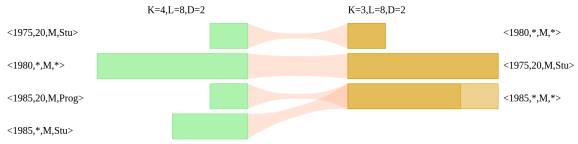


Figure 3: Visualizing changes between two consecutive clustering in Example 1.2: from $k = 4$ to $k = 3$.

- We implement multiple optimization techniques to further speed up computation of these solutions. We evaluate the effect of these optimizations experimentally.

Visualizing Changes in Clustering. When the user explores the solution space updating an input parameter, in some scenarios the solution can change marginally, whereas in others it can change drastically. To help the user understand how two consecutive solutions compare with each other, our framework produces a visualization showing the old and new solution, and how the tuples in these clusters are redistributed (also the size of the cluster, the fraction of top-L tuples contained in them, etc.). An example is shown in Figure 3 that shows that if $k = 4$ in Example 1.2 is changed to $k = 3$, then two of the clusters will merge to form the new solution (Section 6).

- To achieve a *clean* visualization, we formulate an optimization problem that minimizes the crossing of the bands showing flow of tuples from old to new clusters.
- We give an optimal poly-time algorithm by reducing this problem to min-cost perfect matching in bipartite graphs.

Roadmap. We define some preliminary concepts in Section 2. The above three sets of results are discussed in Sections 3, 5, and 6 respectively. Section 4 describes the algorithms and the experimental results are presented in Section 7. We discuss the related work in Section 8 and conclude in Section 9 with scope of future work. Further details appear in the appendix and some details are deferred to the full version due to space constraints.

2 PRELIMINARIES

Let R be a relation with attributes $\mathcal{A}$, which can either be an input table (base relation) or a derived relation (intermediate or final query result). Let $\mathcal{A}_{groupby} \subseteq \mathcal{A}$ be a set of *grouping attributes* used in the group-by clause where $|\mathcal{A}| = m$. Let $aggr(B)$ be any aggregate function allowed by SQL that outputs a real number. Therefore, we are considering a query Q of the form:

```
SELECT A_groupby, aggr(B) as val   -- (Q)
FROM R
GROUP BY A_groupby
ORDER BY val DESC
```

We denote the output of this query as S, where each tuple in S is called an *original element*. Here `val` denotes the *score* or *value* of each output tuple in S. Usually, the query will output n tuples (*i.e.*, $|S| = n$). Even if the number of attributes m in the group-by clause is small, n might be a large number due to large domains of the participating attributes. Therefore, a user frequently runs a top-L query to retrieve the top-L tuples (denoted by S_L^*) with highest scores (adding a `LIMIT L` clause to the above query).

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 the names of the attributes to specify a cluster. For instance, for $m = 4$ attributes A_1, A_2, A_3, A_4, the cluster $(a_1, b_1, *, *)$ 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 $C \in \mathcal{C}$, $\text{cov}(C) \subseteq S$ denotes the elements covered by C, and for $\mathcal{O} \subseteq \mathcal{C}$, $\text{cov}(\mathcal{O}) \subseteq S$ denotes the elements covered by at least one cluster in $\mathcal{O}$, *i.e.*, $\text{cov}(\mathcal{O}) = \cup_{C \in \mathcal{O}} \text{cov}(C)$. Figure 4 shows two 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)

	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

Figure 4: Example clusters and the elements they cover.

Distance function. While the distance between two elements is straightforward (the number of attributes where their values differ), 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. This distance

function can be shown to be a metric and it exhibits monotonicity property (discussed in Section 3) that we use in our algorithms.

Definition 2.1. *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 one of the values is $*$ or the values are different: $d(C, C') = |\{i \in [1, m] : C[A_i] = *, \text{or}, C'[A_i] = *, \text{or}, C[A_i] \neq C'[A_i]\}|$.*

In Figure 4, 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.

3 A TWO-LAYERED FRAMEWORK

In this section, we discuss the technical details for our two-layered framework: Section 3.1 formally defines the optimization problem, Section 3.2 discusses the semilattice structure and properties of the clusters, and Section 3.3 discusses the complexity of the optimization problem.

3.1 Optimization Problem Definition

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

Definition 3.1. *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., $S^*_L \subseteq \mathsf{cov}(\mathcal{O})$.*
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 two clusters in $\mathcal{O}$ cover each other (equivalently, the clusters should form an* antichain *in the semilattice discussed in Section 3.2).*

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

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. In this process the output clusters may cover additional redundant elements. 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 the total value of the covered elements, since otherwise, always the *trivial solution* $(*, *, \cdots, *)$ (discussed below) covering all elements in S will be chosen.

The cluster $(*, *, \cdots, *)$ with don't care value of all attributes $A_1, \cdots, A_m$ and containing all elements is a trivial feasible solution satisfying all the properties in Definition 3.1. However, the average value of the trivial solution can be low due to the presence of many redundant elements. In addition, inclusion of this cluster prohibits inclusion of any other cluster due to the incomparability condition. However, in some cases, the trivial solution may form an optimal solution for Max-Avg (details deferred to the full version due to space constraints).

We also investigated an alternative objective called Min-Size that minimizes the number of redundant elements and work better if the user intends to inspect fewer elements in the second layer (details in full version). However it may miss some interesting global properties covering many high-valued elements in S, and is less useful for summarizing high-valued aggregate answers.

3.2 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 2 naturally induces a semilattice structure[1] 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.*, $\mathsf{cov}[C] \subseteq \mathsf{cov}[C']$. 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 ℓ $*$ values. Since there are m attributes in the relation S, there are $m + 1$ levels in the semilattice, the top-most level has m $*$-s (m-th level), *i.e.*, the trivial solution, whereas the bottom-most level has 0 $*$-s (0-th level), *i.e.*, the singleton clusters containing the original elements. Figure 5 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 $D_2 = \{b_1, b_2\}$.

[1] It is not a lattice since the clusters do not have a *greatest lower bound or meet.*

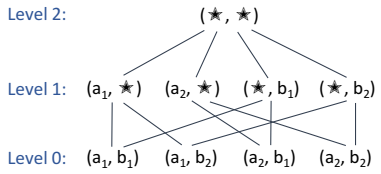


Figure 5: Semilattice on clusters; $m = 2$.

This distance function described in Section 2 has a nice property of being monotone with respect to the partial orders in the lattice that we use in devising our algorithms in Section 4 (The proof is presented in Appendix A.1):

Proposition 3.2. **(Monotonicity)** *Let O be a set of clusters (that may also include elements). Let λ be the minimum distance between any two clusters in O as defined in Definition 2.1, i.e. $\lambda = \min_{C,C' \in SC} d(C, C')$.*
Let $SC' = (SC \setminus \{C1\}) \cup \{C2\}$, where a cluster $C1$ is replaced by another cluster $C2$ such that $C2$ covers $C1$ (i.e., $C2$ is an ancestor of $C1$ 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 5, 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 distance constraint for $D = 2$.

3.3 Complexity Analysis

In this section, we discuss the complexity of the optimization problem. If the size limit k is a constant, the problem can be solved in polynomial time. 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 in practice. 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 $N^k = 10^{40}$. Therefore, our goal is to find efficient algorithms that run in polynomial time in k as well.

The complexity of the problem may arise due to any of the four factors in Definition 3.1: 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 in the top layer of the solution), the decision and the optimization problems become relatively easier as summarized below:

- $k \geq L, D = 0$: Top-k elements give the optimal solution, since adding any redundant element worsens both the `Max-Avg` and the `Min-Size` objectives.
- $k \geq L$, 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.
- $k < L, D = 0$: Even checking whether a non-trivial feasible solution exists is NP-hard.
- $k < L$, 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 [20], 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. One of the proofs is given in the Appendix (Section A.2), the rest are deferred to the full version due to space constraints.

4 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 3.1 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 4.1, 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 4.2, 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 whereas `Fixed-Order` is more efficient, hence in Section 4.3 we describe a `Hybrid` algorithm combining these two.

4.1 The Bottom-Up Greedy Algorithm

Here we start with L singleton clusters with the top-L elements as our current solution 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 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 O cover the top-L answers. (2) *(Incomparability)* No cluster in O covers another. (3) *(Distance)* The minimum distance among the pairs of clusters in 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`(O, C_1, C_2) **procedure.** Given two clusters $C_1, C_2 \in O$, the `Merge`(O, C_1, C_2) procedure replaces C_1, C_2 by a new cluster $C_{new} = \text{LCA}(C_1, C_2)$, their *least common ancestor*, and also removes any other cluster in O that is also covered by C_{new}. $\text{LCA}(C_1, C_2)$ 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 O, `Merge` would also remove this cluster, since it is covered by $(a_1, *, *, *)$.

Apart from the coverage condition, the merging process does not add any new violations to the distance condition in O. This follows from the monotonicity of the distance condition given in Proposition 3.2. However, due to the merging process, the value of the solution may decrease, since $\text{LCA}(C_1, C_2)$ covers all the elements covered by C_1, C_2 and all other clusters that are removed from O, and can potentially cover some more.

The bottom-up algorithm is given in Algorithm 1. The `UpdateSolution`(O, P) procedure used in this algorithm takes the current solution O and a set of pairs of clusters P to be considered for merging, and greedily merges a pair is given in Algorithm 2. The first and second phases of Algorithm 1 are very similar, the only difference being the pairs of clusters P it considers for merging. In the first phase, only the pairs with distance $< D$ are considered, whereas in the second phase, all pairs of clusters in O are considered for merging.

Algorithm 1 The `Bottom-Up` algorithm

Input: Size, coverage, and distance constraints k, L, D

O = set of L singleton clusters with the top-L elements.
/ First phase to enforce distance */*
while O has two clusters with distance $< D$ **do**
  Let P_D be the pairs of clusters in O *at distance* $< D$.
  Perform `UpdateSolution`(O, P_D).
end while
/ Second phase to enforce size limit k, almost the same as above except all pairs of clusters are considered. */*
while $|soln| > k$ **do**
  Let P_{all} be the *all* pairs of clusters in O.
  Perform `UpdateSolution`(O, P_{all}).
end while
return O

Algorithm 2 `UpdateSolution`(O, P) for `Max-Avg`.

Input: Current set of clusters O, and pairs of clusters to be considered for merging P

*/** $\text{LCA}(C_1, C_2)$ *denote the smallest cluster containing* C_1, C_2 *(see text) */ /* Choose the pair that gives highest value of the new solution after merging */*
$(C_1, C_2) = \texttt{argmax}_{(C_1, C_2) \in P} \texttt{avg}(O \cup \text{LCA}(C_1, C_2))$
Perform `Merge`(O, C_1, C_2) (see text).

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 $\texttt{avg}(\text{LCA}(C_1 C_2))$ (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.

4.2 The Fixed-Order Greedy Algorithm

The `Fixed-Order` algorithm (Algorithm 3) maintains a set of clusters O, considers top-L elements in order, decides whether the next element is already covered by an existing cluster in O or can be added as is (satisfying D and k constraints); otherwise merges it with one of the existing clusters in greedy fashion. All the 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), so has better running time than `Bottom-Up`. Pseudocode for `Fixed-Order` is shown as Algorithm 3 in Appendix (Section A.3).

4.3 The Hybrid Greedy Algorithm

In order to combine the advantages of both `Bottom-Up` (better quality) and `Fixed-Order` (better efficiency), we introduce the hybrid algorithm. It has 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 ck, $c > 1$ is a constant, initial singleton clusters. After covering all top-L elements in ck clusters, `Hybrid` goes into the `Bottom-Up` phase to reduce the number of clusters from ck to k using the `Merge` procedure that can collect redundant elements.

We also considered other variants of hybrid, *e.g.*, where the initial clusters are chosen by the randomized `k-Means++` algorithm [3] to ensure a good set of initial clusters with multiple runs of clustering in the fixed-order phase. However, the above version of hybrid algorithm gave comparable results and better running time compared to the other variants. In addition to producing solutions with larger values, `Bottom-Up` and `Hybrid` algorithms help in incremental computation for different choices of parameters as discussed in the next section.

5 INTERACTIVE 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. Our framework takes three parameters: size limit on the clusters k, distance lower bound D among solutions, and the coverage parameter L so that the clusters cover at least the top-L original answer tuples. 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 5.1 we describe the visualization facilitating parameter selection, in Section 5.2 we discuss how the precomputation is achieved to plot these graphs and how does it help in efficiently retrieving the answers. In Section 5.3 we discuss a number of optimizations that

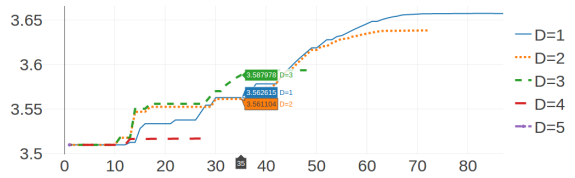


Figure 6: Guidance visualization with detail

make computation of the answers for multiple combinations of input parameters efficient.

5.1 Visualization Guiding Parameter Selection

Figure 6 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 3.1), the values 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). The right end of each line represents the position where the distance enforcement of the `Bottom-Up` phase finishes. Applying any k larger than that position for the given D will share the same result as the right end. Changing L value will result in a different plot.

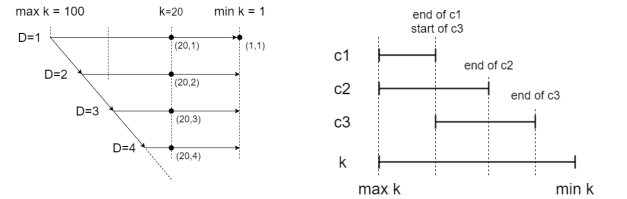
With the help of this visualization, the user can avoid selecting certain *uninteresting* or *redundant* parameter combinations. For example, the bottom-left region in Figure 6 can be considered as uninteresting with a low average value (only the trivial solution with all "$*$" belongs to the solution. Also the user can avoid changing k or D that lead to the same value of the solution (*e.g.*, for $D = 2$, the range of $k = [20, 30]$ gives almost the same values), which can make the exploration easy and more efficient for the user.

5.2 Incremental Computation and Storage

To be able to generate plots in Figure 6, one obvious approach is running one of the algorithms in Section 4 for all required combinations of k and D given an L value. However, for interactive exploration, this approach is sub-optimal. Further, the `Hybrid` algorithm (also `Bottom-Up`) exhibits *two levels of* incremental properties that helps in computing Fthe solutions for a range of k, D values without redoing the computations from the beginning.

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 7a. 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



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

Figure 7: Incremental computation and interval structure

time efficient based on the following observation in the execution of `Hybrid` (and `Bottom-Up`) algorithm:

Proposition 5.1. **(Continuity)** *Given solution cluster lists $O_1, O_2, ..., O_r$ where r rounds are executed, for any cluster $c \in O_a$ where $1 \leq a < i$, once c is removed from O_i at the end of round i (because of merging), for all $j > i$, $c \notin O_j$.*

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 $O_{L,D,k}$ denotes the solution for a given combination of L, D, k, the set of values of k for which a cluster $c \in O_{L,D,k}$ forms a continuous interval. Therefore, instead of storing the set of clusters for all values of D, k given an L value (where the solutions may have substantial overlap), we use an *interval tree*[7] S_D for each value of D that stores the range of k for which a cluster appears in $O_{L,D,k}$ storing only the `maximum` (or starting) and `minimum` (or ending) k value for this cluster (see Figure 7b). 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 $O(\log N_k)$[7]. .

5.3 Optimizations

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

5.3.1 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 2). Let O_i be the set of clusters at the end of a round i, $T_i = \text{cov}(O_i)$ be the tuples covered by O_i, $v_i = \text{avg}(O_i)$ be the average value of O_i, and $T_c = \text{cov}(c)$ 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 2) 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 $\text{avg}(T_i \cup T_c)$ 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 O_{i+1} compared to O_i as follows.

The basic idea is that the improvement in the total average value that a cluster c brings to solution O_i is due to the tuples in $T_c \setminus O_i$, and that it brings to O_{i-1} is due to the tuples in $T_c \setminus O_{i-1}$. The

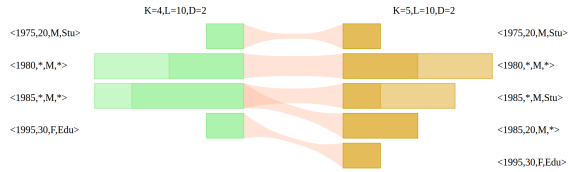


Figure 8: Solution comparison: clean visualization

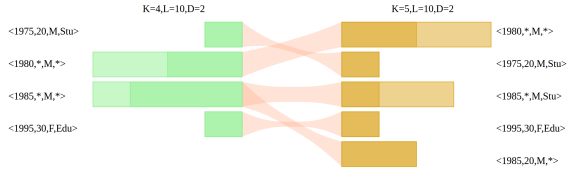


Figure 9: Solution comparison: cluttered visualization

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 O_{i+1} if we add c to 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. The effect of this optimization is evaluated in our experiments (Section 7.3.2) and a pseudocode is given in Algorithm 4 in the appendix.

5.3.2 Cluster Generation and Cluster-Tuple Mapping. The semi-lattice 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 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 the cluster and searching for matching tuples). Experiments in Section 7.3.1 shows the benefit - $100x - 1000x$ speedup in running time.

5.3.3 Hash Values for Fields. In practice, 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). We observed that this optimization reduced the running time of the order of $50x$.

6 VISUALIZING SUCCESSIVE SOLUTIONS

Figures 1 and Figure 15 show how a user can inspect the clusters and the elements they contain in the form of tables in our framework. However, for interactive exploration, it is also important that the user can see how the old solution changes to a new solution (if k, D, or L is updated), since in some cases the change can be incremental, whereas in other cases, changing a parameter may give a complete new solution. To support this comparison, we display the successive solutions and their overlaps (Section 6.1), and formulate it as an optimization problem for a clean display (Section 6.2).

6.1 Visualizing Changes

An example visualization is shown in Figure 8. Each box corresponds to a cluster in the solution. The left hand side boxes (in green) are result clusters for the previous run, and the right hand side boxes (in yellow) are clusters under the current parameters. The width of each box is proportional to the number of tuples contained in the cluster. Clusters connected with bands or ribbons contain shared tuples. The thicker a band is in the middle, the more common tuples it contains [2]. The parts of the boxes in darker color correspond to the fraction of top-L tuples contained in these clusters. Hovering the mouse over different regions shows details.

6.2 Optimizing Cluster Placement

A careful ordering of the cluster boxes help in displaying a cleaner visualization comparing two consecutive executions. For instance, Figure 9 is an example where placement of the clusters (boxes) leads to more *crossing* of the bands, whereas Figure 8 shows a better placement. The placement can have more effect when the number of clusters (the value of k) is larger. Therefore, we formulate the cluster placement problem as an optimization problem that intends to minimize crossing of the bands as follows.

Optimization problem. Let O_a and O_b denote the old and new set of clusters respectively in two consecutive runs containing clusters $O_a = \{c_{a1}, c_{a2}, ..., c_{am}\}$ and $O_b = \{c_{b1}, c_{b2}, ..., c_{bn}\}$. Let m_{ij} denote the number of shared tuples between clusters c_{ai} and c_{bj}, and M denote the set of all such m_{ij} values. We assume that the first clusters in both sides are placed at the same vertical position, and there are no gaps or overlaps between two adjacent clusters on either side. Consider two orderings of clusters on the left hand side and right hand side respectively in terms of their starting positions $P_a = p_{a1}, p_{a2}, .., p_{am}$ (a permutation of $[0, m-1]$), and $P_b = p_{b1}, p_{b2}, ..., p_{bn}$ (a permutation of $[0, n-1]$). We define a weighted *earth mover's distance*[26] d_{ij} between one left cluster c_{ai} and one right cluster c_{bj} to evaluate the amount of crossing due to a single band (from c_{ai} to c_{bj}) as:

$$d_{ij} = m_{ij} \times |p_{ai} - p_{bj}|$$

Since O_a is the previous cluster set, P_a is fixed and given. The goal of the optimization problem for this visualization is to output a good ordering P_b for O_b, and is formulated as follows:

Definition 6.1. ***Optimization for placement of clusters.*** *Given old and new clusters O_a, O_b, their overlaps M, and ordering P_a of the clusters on the left hand side, find an ordering P_b of the clusters on the right hand side that minimizes $D = \sum_{i=1}^{m} \sum_{j=1}^{n} d_{ij}$.*

Optimal solution using bipartite matching. The above optimization problem can be reduced to the minimum cost perfect

[2] This visualization shows some similarity with SANKEY diagrams that are widely used in the field of energy and material flow management [21]. and provide flow from one set of objects to another. However, popular SANKEY diagram libraries (*e.g.*, d3-sankey) focus more on managing placement among columns (horizontal positioning). For vertical positioning, they do multiple iterations to re-position objects to achieve a satisfying visualization. We do not need to consider multiple columns, and our visualization focuses on vertical positioning and ordering of objects

matching problem in a complete bipartite graph as follows. We form a weighted complete bipartite graph $G(U \cup V, E)$, where the n nodes in U correspond to the n clusters in O_b, and the n nodes in V correspond to the positions $1 \cdots n$. An edge (u, v) denotes the possibility when cluster c_{bu} is placed in position $v \in [1, n]$. The weight of the edge (u, v) is the cost $\sum_{i=1}^{m} d_{iu} = \sum_{i=1}^{m}(m_{iu} \times |p_{ai} - (v-1)|)$ (if c_{bu} is placed in position v, there will be $v-1$ clusters before it, and its position will be $v-1$), *i.e.*, the total contribution of cluster c_{bu} in the optimization objective in Definition 6.1 if it is placed in position $v \in [1, n]$. Since O_a and P_a are given and fixed, this weight can be computed in polynomial time as a precomputation step for each cluster in O_b and each position. A matching gives a positioning P_b on the clusters in O_b, and the minimum cost perfect matching, which has a polynomial time algorithm[14], gives an optimal solution to our optimization problem.

We also studied an alternative formulation of the above optimization problem, where instead of the *width* of the boxes being proportional to the number of tuples in clusters, the *height* is proportional to the number of tuples, *i.e.*, the starting positions P_a and P_b are no longer permutations of $[0, m-1]$, $[0, n-1]$ but also depend on the height of the individual clusters. However, we found that this variant is NP-hard by a reduction from the *earliness-tardiness job scheduling problem*[13]. The proof and details of this alternative formulation are deferred to an extended version of this paper due to space constraints.

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. The architecture of the system is described in the full version and a snapshot of the GUI is shown in Figure 15 in Section A.5. In this section we experimentally evaluate our algorithms using our prototype by varying different parameters (Section 7.1), and then test the precomputation performance (Section 7.2). The effect of optimizations are given in Section 7.3 and scalability of our algorithms for a larger dataset is discussed in Section 7.4. Additional discussion about the element selection order in `Fixed-Order` is provided in Section 7.5.

Datasets. In most of the experiments, we use the MovieLens 100K dataset [18, 29, 30]. 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 aggregate queries used in this section are of the following form.

```
SELECT ⟨ grouping attributes⟩, avg(rating) as val
FROM RatingTable
GROUP BY ⟨ grouping attributes⟩
HAVING count(*) > 50
ORDER BY val DESC
```

The other dataset we use is TPC-DS benchmark [22] primarily for evaluating scalability of our algorithms. The table we materialized via generator is *Store_Sales*, which contains 23 attributes and 2880404 tuples in total. The aggregate queries we use for TPC-DS related experiments are shown as follows.

```
SELECT ⟨ grouping attributes⟩, cast(avg(net_profit)
       as int) as val
FROM store_sales
GROUP BY ⟨ grouping attributes⟩
HAVING count(*) > 10
ORDER BY val DESC
```

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).

7.1 Varying Parameters

We present running time and quality of solution for the objective `Max-Avg`; the `Min-Size` objective yields similar conclusions and is omitted due to space constraints.

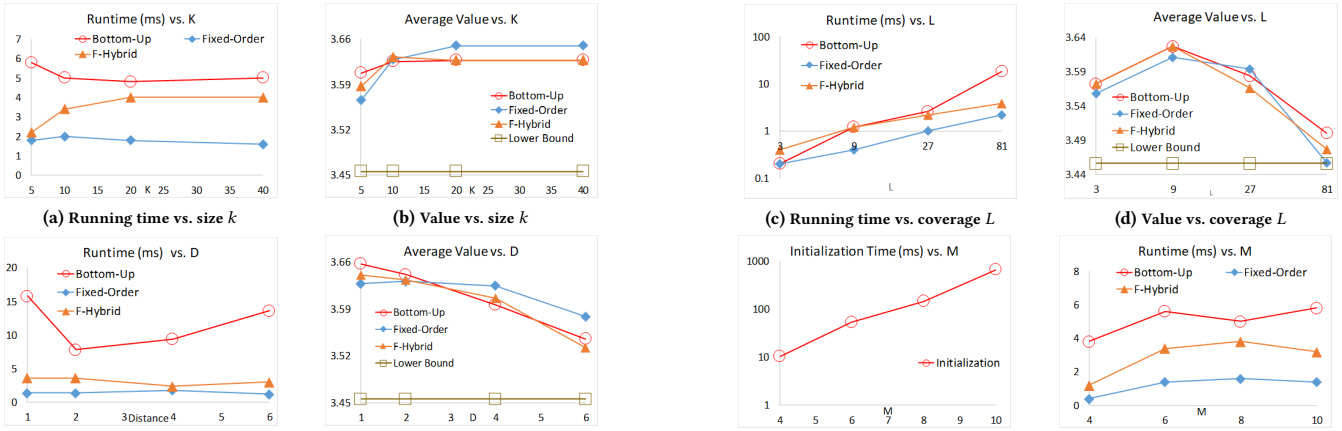
Unless mentioned otherwise, the three algorithms from Section 4 are compared in this section: (i) `Bottom-Up`, (ii) `Fixed-Order`, (iii) `Hybrid`. In the plots showing the values, we also include (iv) `Lower Bound`: value of the trivial solution (a single cluster with don't-care $*$ values for all attributes) as a baseline which is always feasible.

7.1.1 Comparison with brute-force algorithm. Figure 11a compares our algorithms with the brute-force algorithm ($L = 5, D = 3$ and $k = 2, 3, 4$). 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 11b compares the average values of our algorithms with brute-force, and shows that the results are comparable with brute-force's result and are much better than the trivial solution.

7.1.2 Effect of size parameter k. Figure 10a 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 C_ℓ and hence the cost of computing the set cover. The running times tend 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 brings 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 4, although gets better with larger k in Figure 10b.

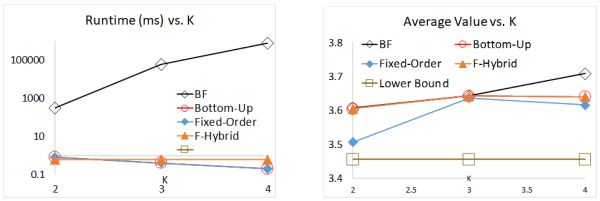
7.1.3 Effect of coverage parameter L. Figure 10c shows that running time of all algorithms increase as the number of elements to be covered L increases. Since `Fixed-Order` depends linearly on L, it is less affected by L, whereas `Bottom-Up` treat individual elements as clusters and may incur quadratic time w.r.t. L. For `Hybrid`, with the restriction of the canidate pool's size determined by k, the run time increase is slower than `Bottom-Up` and is comparable with `Fixed-Order`. Note that in Figure 10d, the upper bound decreases since with L increasing, the average value of the top-L elements



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

(e) Running time vs. distance D (k=10, L=40) (f) Value vs. distance D (k=10, L=40) (g) Initialization time vs. no. of attributes m (k=L=20) (h) Running time vs. no. of attributes m (k=L=20)

Figure 10: Experimental results varying parameters, algorithms, and optimization criteria. The default values of parameters are $m = 8$, $k = 3$, $L = 40$, $D = 3$.



(a) Running time (brute-force), L=5, D=3 (b) Value (brute-force), L=5, D=3

Figure 11: Experimental results varying parameters, algorithms, and optimization criteria. The default values of parameters are $m = 8$, $k = 3$, $L = 40$, $D = 3$.

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`.

7.1.4 Effect of distance parameter D. In Figure 10e, `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 10f.

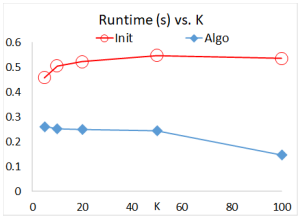
7.1.5 Effect of number of attributes m. Varying the number of grouping attributes m also illustrates the effect of varying input data size. Since our algorithms run on the output of an aggregate query, as m increases, our input data size $|S| = n$ is likely to increase (for the m values in Figure 10g and 10h, the size of the input ranges from 140 to 280). When a new query is entered by the user, the system performs an initialization step of constructing the clusters and the semi-lattice structure. This initialization time is shown in Figure 10g. This step is performed only once per query, and when the user investigates the result by varying k and D, this is not performed again. Our implementation takes from 10ms when $m = 4$ to and 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 10h has the running times 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

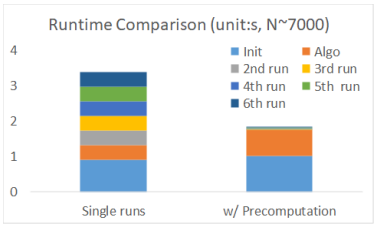
The performance evaluation for precomputation is shown by varying k, L and D separately, and compare the running time of `Hybrid` between precomputation implementation and non-precomputation (single) implementation. From Section 7.2.1 to Section 7.2.3, the discussion will be focused on the effect brought by varying parameters. Discussion on single run vs. multiple runs will be presented in Section 7.2.4.

7.2.1 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 12a: the initialization time hardly changes with the size parameter 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.

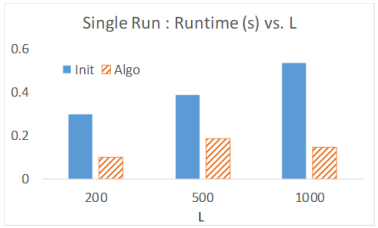
7.2.2 Effect of coverage parameter L. The fixed parameters are $k = 20$,$D = 2$ and $N = 2087$. Three L values are selected for the experiment:$L = 200, 500, 1000$. The running time results for single version and precomputation version are presented in Figure 12c and Figure 12d. Both implementations 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



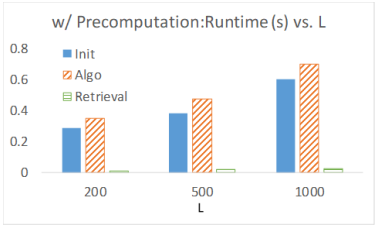
(a) Runtime vs. size k



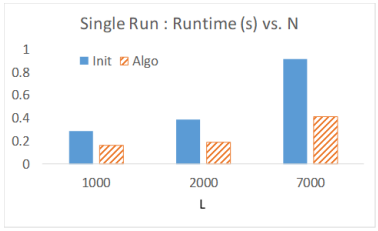
(b) Runtime comparison, total elements $N = 7000$



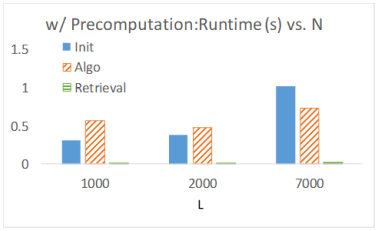
(c) Single runtime vs. coverage L



(d) With precomputation runtime vs. coverage L



(e) Single runtime vs. total elements N



(f) W/ precomputation runtime vs. total elements N

Figure 12: Experimental results varying parameters and with/without precomputation

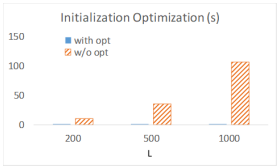
time for precomputation implementation is extremely short (tens of milliseconds), which can make up for the time in several runs.

7.2.3 Effect of total elements N. In this case, three parameters k, L, D are fixed with $k = 20$, $L = 500$, $D = 2$. The total number of input elements varies to test the system's performance with relatively higher capacity: $N = 927, 2087$ and 6955. The running time result is shown in Figure 12e and Figure 12f. The changing trends are similar with those in Section 7.2.2, but a significant increase for the initialization time can be observed with N growing. This is caused by the need for materializing more possible clusters brought by variety of tuples.

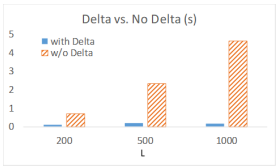
7.2.4 Single run vs. multiple runs. From Figure 12c, 12d, 12e and 12f, the information is enough for comparing precompution and non-precomputation versions on both single run and multiple runs scenario - For single runs, preomputation process is unused, making precomputation version the slower and more 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 12b with $N = 6955$ for a direct comparison. From Figure 12b, if only one single run is required, the single version of `Hybrid`is clearly faster and cheaper than the precomputation version. However, when the 3^{rd} run finished, the precomputation version was already faster than the implementation without precomputation; When all six runs finished, the single version took about two times in terms of running time compared with the precomputation version. The results are expected and perfect fit for our design.

7.3 Benefit of Optimizations

In order to show the advantage brought by optimization discussed in Section 5.3, we designed two set of experiments to highlight the effect. Section 7.3.1 shows the comparison result for initialization related optimization, and *Delta Judgment* related optimization are examined in Section 7.3.2.



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



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

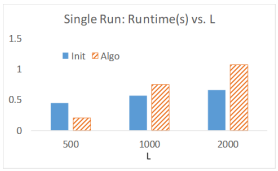
Figure 13: Experiment on effects of optimizations

7.3.1 Optimization in Initialization. The experiment in this section is designed for Section 5.3.2. 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 other variables are fixed: $k = 20, D = 2, N = 2087$. The result is presented in Figure 13a. 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 - provides significant performance improvement by cutting down the running time from > 100 seconds for $L = 1000$ to 0.5 seconds.

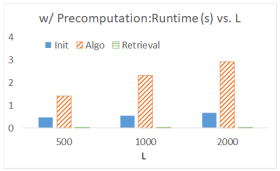
7.3.2 Delta Judgment. In this section, the effect by introducing *Delta Judgment* is shown in Figure 13b. Given that L is also the most effective variable to affect the running time, the experimental settings are the same as the experiment in Section 7.3.1. However, only the algorithm's running time is plotted since *Delta Judgment* has no effect on the running time of initialization. The result in Figure 13b proves that the *Delta Judgment* successfully improves the algorithm's efficiency from 4.6 seconds to 0.15 seconds 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 single and precomputation version are evaluated using this set of parameters. From the



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



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

Figure 14: TPC-DS experimental results varying parameters and w/o precomputations.

results shown in Figure 14a and Figure 14b, the initialization time is still interactive - about one second for the largest parameters: $L = 2000$ and $N = 47361$. However, even for the single version, the running time of the algorithm increases to more than one seconds compared with < 100 milliseconds from results in Section 7.2.3 and the precomputation running increases to 2.5 seconds. Although the running time increases, the total running time (3.5 seconds) for the precomupation version is still interactive.

7.5 *Fixed-Order* vs. *Random-Order*

`Random-Order` is a similar algorithm with the `Fixed-Order` algorithm expect that `Random-Order` randomly picks an element inside top-L each round. For the sake of comparing the data quality between the two algorithms, we record the results for `Random-Order` recursively and build a scatter plot comparing `Fixed-Order` and `Random-Order` under different parameters. The parameters are $k = 20, D = 2, N = 2087$, while $L = 100, 200$ and 500. for each combination, we run `Random-Order` for 100 times and record all output average values and add in the values of `Fixed-Order` to build the plot shown in Figure 16 (in the appendix). It can be shown that when L gets larger, the average values of `Random-Order` is increasingly higher than the results given by `Fixed-Order`. However, the deviation for `Random-Order` is large as well. As a result, in cases where parameters are small so that the running time would be small as well, `Random-Order` phase can replace `Fixed-Order` phase in `Hybrid` (into `R-Hybrid`). The `Random-Order` could run several times, pick the best run followed by the `Bottom-Up` phase.

8 RELATED WORK

First we discuss three recent papers relevant to our work that consider *result diversification* or *result summarization* are smart drill-down [19], diversified top-k [24], and disC diversity[9]. 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, or the setting studied in [9, 19, 24] do not suffice to meet the goals in our work; we discuss these results in Appendix A.4. There are several other related work in the literature that we briefly mention in this section, and discuss in detail in Appendix A.6.

Smart drill-down [19]: In a recent work, Joglekar et al. [19] 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). As we discuss in Section A.4, 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-valued and low-valued tuples.

Diversified top-k [24]: Qin et al. [24] formulated the top-k result diversification problem as follows: 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. [24] considers diversification, but it does not consider result summarization using $*$-values and it chooses representative elements instead. In addition to lacking high level properties with $*$ values, 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 [9]: Drosou and Pitoura [9] proposed the notion of *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 $> \tau$). Summarization and diversification can be achieved similar to [24]. However, it ignores the values or relevance of the elements (unlike us or [24]) and has no bound on the number of elements returned (unlike us, [19, 24]), therefore may not be useful when the user wants to investigate a small set of answers.

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 [1, 2, 4, 5, 8, 9, 11, 12, 15, 17, 24, 25, 27, 28, 31–38] . 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. We discuss these related work in further detail in Appendix A.6.

9 CONCLUSIONS

In this paper, we presented a framework for summarization and exploration of high-valued aggregate query answers. The framework outputs a set of clusters summarizing the attribute values of high-valued answer tuples, and maintains desired properties like diversity, small size, and coverage of original top elements. The framework facilitates interactive exploration by helping the user choose the input parameters easily, inspect clusters and the tuples they contain, and visualize the effect of changing parameters by comparing two consecutive solutions. We studied optimization problems for these tasks together with their complexity, explored properties of the solutions, developed efficient algorithms and optimizations, evaluated the approach using real and benchmark datasets, and showed that our implementation is capable of interactive exploration in real time. As future work, we plan to explore more sophisticated visualizations that can better help the users and study the relevant optimization problems further.

REFERENCES

[1] Zeinab Abbassi, Vahab S. Mirrokni, and Mayur Thakur. 2013. Diversity maximization under matroid constraints. In *KDD*. 32–40.
[2] Rakesh Agrawal, Sreenivas Gollapudi, Alan Halverson, and Samuel Ieong. 2009. Diversifying search results. In *Proceedings of the second ACM international conference on web search and data mining*. ACM, 5–14.
[3] David Arthur and Sergei Vassilvitskii. 2007. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1027–1035.
[4] Allan Borodin, Hyun Chul Lee, and Yuli Ye. 2012. Max-Sum Diversification, Monotone Submodular Functions and Dynamic Updates. In *PODS*. 155–166.
[5] Jaime Carbonell and Jade Goldstein. 1998. The Use of MMR, Diversity-based Reranking for Reordering Documents and Producing Summaries. In *SIGIR*. 335–336.
[6] Zhiyuan Chen and Tao Li. 2007. Addressing Diverse User Preferences in SQL-query-result Navigation. In *SIGMOD*. 641–652.
[7] Thomas H Cormen. 2009. *Introduction to algorithms*. MIT press.
[8] Ting Deng and Wenfei Fan. 2014. On the Complexity of Query Result Diversification. *ACM TODS* 39, 2 (May 2014), 15:1–15:46.
[9] Marina Drosou and Evaggelia Pitoura. 2012. Disc diversity: result diversification based on dissimilarity and coverage. *PVLDB* 6, 1 (2012), 13–24.
[10] Kareem El Gebaly, Parag Agrawal, Lukasz Golab, Flip Korn, and Divesh Srivastava. 2014. Interpretable and Informative Explanations of Outcomes. *PVLDB* 8, 1 (Sept. 2014), 61–72.
[11] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Diversified Top-k Graph Pattern Matching. *PVLDB* 6, 13 (2013), 1510–1521.
[12] Piero Fraternali, Davide Martinenghi, and Marco Tagliasacchi. 2012. Top-k Bounded Diversification. In *SIGMOD*. 421–432.
[13] Michael R Garey, Robert E Tarjan, and Gordon T Wilfong. 1988. One-processor scheduling with symmetric earliness and tardiness penalties. *Mathematics of Operations Research* 13, 2 (1988), 330–348.
[14] Michel X. Geomans. 2009. *Lecture notes on bipartite matching*. Massachussets Institute of Technology. http://math.mit.edu/~goemans/18433S09/matching-notes.pdf
[15] Orestis Gkorgkas, Akrivi Vlachou, Christos Doulkeridis, and Kjetil Nørvåg. 2015. Finding the Most Diverse Products using Preference Queries. In *EDBT*. 205–216.
[16] Lukasz Golab, Flip Korn, Feng Li, Barna Saha, and Divesh Srivastava. 2015. Size-Constrained Weighted Set Cover. In *ICDE*. 879–890.
[17] Sreenivas Gollapudi and Aneesh Sharma. 2009. An Axiomatic Approach for Result Diversification. In *WWW*. 381–390.
[18] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.* 5, 4, Article 19 (Dec. 2015), 19:1–19:19 pages.
[19] Manas Joglekar, Hector Garcia-Molina, and Aditya G. Parameswaran. 2016. Interactive data exploration with smart drill-down. In *ICDE*. 906–917.
[20] Donna Crystal Llewellyn, Craig A. Tovey, and Michael A. Trick. 1993. Erratum: Local Optimization on Graphs. *Discrete Applied Mathematics* 46, 1 (1993), 93–94.
[21] Schmidt M. 2008. The Sankey diagram in energy and material flow management. *Journal of industrial ecology* 12, 1 (2008), 82–94.
[22] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The making of TPC-DS. In *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 1049–1058.
[23] T. T. Nguyen, Q. V. H. Nguyen, M. Weidlich, and K. Aberer. 2015. Result selection and summarization for Web Table search. In *ICDE*. 231–242.
[24] Lu Qin, Jeffrey Xu Yu, and Lijun Chang. 2012. Diversifying Top-K Results. *PVLDB* 5, 11 (2012), 1124–1135.
[25] S. S. Ravi, Daniel J. Rosenkrantz, and Giri Kumar Tayi. 1994. Heuristic and Special Case Algorithms for Dispersion Problems. *Operations Research* 42, 2 (1994), 299–310.
[26] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. 2000. The earth mover's distance as a metric for image retrieval. *International journal of computer vision* 40, 2 (2000), 99–121.
[27] Sunita Sarawagi. 2000. User-Adaptive Exploration of Multidimensional Data. In *VLDB*. 307–316.
[28] Yufei Tao. 2009. Diversity in Skylines. *IEEE Data Eng. Bull.* 32, 4 (2009), 65–72.
[29] http://grouplens.org/datasets/movielens/. [n. d.]. ([n. d.]).
[30] http://movielens.org. [n. d.]. ([n. d.]).
[31] Erik Vee, Utkarsh Srivastava, Jayavel Shanmugasundaram, Prashant Bhat, and Sihem Amer-Yahia. 2008. Efficient Computation of Diverse Query Results. In *ICDE*. 228–236.
[32] Marcos R. Vieira, Humberto L. Razente, Maria C. N. Barioni, Marios Hadjieleftheriou, Divesh Srivastava, Caetano Traina, and Vassilis J. Tsotras. 2011. On Query Result Diversification. In *ICDE*. 1163–1174.
[33] Dong Xin, Hong Cheng, Xifeng Yan, and Jiawei Han. 2006. Extracting Redundancy-aware Top-k Patterns. In *KDD*. 444–453.
[34] Cong Yu, Laks Lakshmanan, and Sihem Amer-Yahia. 2009. It Takes Variety to Make a World: Diversification in Recommender Systems. In *EDBT*. 368–378.
[35] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. 2000. Answering Complex SQL Queries Using Automatic Summary Tables. In *SIGMOD*. 105–116.
[36] Wei Zheng, Xuanhui Wang, Hui Fang, and Hong Cheng. 2012. Coverage-based search result diversification. *Information Retrieval* 15, 5 (2012), 433–457.
[37] Xiaojin Zhu, Andrew B. Goldberg, Jurgen Van Gael, and David Andrzejewski. 2007. Improving Diversity in Ranking using Absorbing Random Walks. In *HLT-NAACL*. 97–104.
[38] Cai-Nicolas Ziegler, Sean M. McNee, Joseph A. Konstan, and Georg Lausen. 2005. Improving Recommendation Lists Through Topic Diversification. In *WWW*. 22–32.

A APPENDIX

A.1 Proof of Proposition 3.2

Proof of Proposition 3.2. Consider any cluster $C \in SC$ other than C_1. Suppose $d(C_1, C) = \ell$, *i.e.*, ℓ of the attributes contribute to the distance function. Fix such an attribute A. There are three possibilities. **(1)** $C_1[A] = C[A] = *$. If $C_2[A] = *$, it contributes 1 to $d(C_2, C)$. If $C_2[A] \neq *$, *i.e.*, an attribute value, then also it contributes 1 to $d(C_2, C)$. **(2)** $C_1[A] \neq *$, $C[A] \neq *$, and $C_1[A] \neq C[A]$. If $C_2[A] = *$, it contributes 1 to $d(C_2, C)$. If $C_2[A] \neq *$, *i.e.*, an attribute value, then it must be the same as $C_1[A]$, and therefore contributes to $d(C_2, C)$. **(3)** $C_1[A] = *$ and $C_1[A] \neq *$. Then $C_2[A] = C_1[A] = *$ and it contributes 1 to $d(C_2, C)$. **(3)** $C_1[A] \neq *$ and $C[A] = *$. Then either $C_2[A] = C_1[A]$ or, $C_2[A] = *$. In either case, it contributes 1 to $d(C_2, C)$. Summing over all A and considering all C, $\lambda' \geq \lambda$. □

A.2 NP-hardness Proofs

Theorem A.1. *The optimization problem for the objective of* `Max-Avg` *for the case when* $k \geq L$ *and* $D > 0$ *is NP-hard.*

Proof. The reduction is from the problem of finding a minimum vertex cover in a tri-partite graph G with partitions (X, Y, Z), which has shown to be NP-hard in [20]. The goal in this problem is to decide if the input graph has a vertex cover of size $\leq M$, *i.e.*, a subset of vertices $T \subseteq X \cup Y \cup Z$ and $|S| \leq M$ such that any edge in G has at least one endpoint in T[3]. First we give the proof for `Max-Avg`. Suppose G has N_e edges and N_v vertices.

Given such a graph G and a bound M, we construct a database instance S as follows. There are three attributes A_X, A_Y, A_Z. (i) *Top-L tuples from the edges of G:* Any edge of the form (x, y), $x \in X, y \in Y$ forms two tuples (x, y, Z^1_{xy}) and (x, y, Z^2_{xy}), where Z^1_{xy}, Z^2_{xy} are two unique values of the A_Z attribute for the edge (x, y) in G. Similarly, an edge (y, z), $y \in Y, z \in Z$ forms two tuples (X^1_{yz}, y, z), (X^2_{yz}, y, z), and an edge (x, z), $x \in X, z \in Z$ forms a tuple (x, Y^1_{xz}, z). (x, Y^2_{xz}, z). The weights of these tuples are 1. (ii) *Redundant tuples from the vertices of G:* For any vertex $x \in A_X$ in G, create a redundant tuple $(x, \gamma^2_x, \gamma^3_x)$. These redundant tuples have weight 0. Similarly, form redundant tuples for vertices $y \in A_Y$: $(\gamma^1_y, y, \gamma^3_y)$, and for vertices $z \in A_Z$: $(\gamma^1_z, \gamma^2_z, z)$ with weight 0. (iii) *More redundant tuples:* For each Z^1_{xy}, form $N_r = 2 * N_e * N_v$ redundant tuples of the form $(-, -, Z^1_{xy})$ with weight 0, where the positions with $-$ are filled

[3] [16] gives a reduction from the tri-partite vertex cover problem for *size-constrained weighted set cover* (given weights on the subsets, a size constraint k, a coverage fraction s, the goal is to return up to k sets that together contain at least sn elements and whose sum of weights is *minimal*). In contrast, in our setting, the weights are assigned on elements (not on subsets); the goal is to select at most k subsets with *maximum* value with the distance and other restrictions, that cover top-L original elements; and we show NP-hardness of the decision problem even if the elements are unweighted.

with unique attribute values. Similarly, form redundant N tuples for each of $Z^2_{xy}, Y^1_{xz}, Y^2_{xz}, X^1_{xz}$, and X^2_{yz}, placing these attribute values in their corresponding positions.

We set $k = M$, $L = 2 \times N_e$, $D = 3$. Note that only the tuples from the edges of the form (x, y, Z^1_{xy}), (x, y, Z^2_{xy}), (x, Y^1_{xz}, z), (x, Y^2_{xz}, z), (X^1_{yz}, y, z), (X^2_{yz}, y, z) form the top-L original elements and have to be covered. We claim that G has a vertex cover of size $\leq M$ if and only if S has a solution, a set of clusters SC, of value $\geq \frac{2N_e}{2N_e+M}$, where N_e is the number of edges in G.

(only if) Suppose G has a vertex cover T of size $M' \leq M$. For any $x \in X \cap T$, choose the cluster $(x, *, *)$ in SC; similarly for $y \in Y \cap T$ and $z \in Z \cap T$, choose the clusters $(*, y, *)$ or $(*, *, z)$ respectively in SC. These clusters have mutual distance = 3, are incomparable, have size $\leq M$, and cover all top-L elements. Each such cluster also covers a redundant element (with γ attribute value) of value 0. Therefore, the value of the solution is $\frac{2N_e \times 1 + M' \times 0}{2N_e+M'} \geq \frac{2N_e}{2N_e+M}$.

(if) Suppose S has a solution SC of value $\geq \frac{2N_e}{2N_e+M}$. Without loss of generality, any cluster in SC covers at least one of the top-L elements with value 1, otherwise it can be discarded without increasing the value of the average or size/distance/coverage of the solution (the redundant elements have value 0 and can only reduce the average). Also note that the trivial solution $(*, *, *)$ cannot be chosen, since it has value $\frac{2N_e+0}{2N_e+N_v+2N_e*N_r} \leq \frac{2N_e}{2N_e+2N_e*N_r} = \frac{1}{1+N_r} = \frac{1}{1+2N_eN_v}$, which is strictly less than the assumed value of $SC \geq \frac{2N_e}{2N_e+M}$.

(A) None of the chosen clusters in SC can be of the form $(*, *, Z^1_{xy})$ *(similarly for* $Z^2_{xy}, Y^1_{xz}, Y^2_{xz}, X^1_{xz}$*).* Suppose one cluster in SC is $(*, *, Z^1_{xy})$. Then it covers N_r redundant tuples that are not covered by any other cluster in SC. Suppose SC has N' redundant tuples all together from all other clusters. Then the average value of SC is $\frac{2N_e+0}{2N_e+N'+N_r} \leq \frac{2N_e}{2N_e+N_r} = \frac{2N_e}{2N_e+2N_eN_v} = \frac{1}{1+N_v}$, which is strictly less than $\frac{2N_e}{2N_e+M}$, the assumed value of SC since $M \leq N_v$.

Next we argue that each cluster in SC can have exactly two $*$-s, combining with (A) above, must be of the form $(x, *, *), (*, y, *), (*, *, z)$.

(B) The clusters in SC cannot have zero $*$*-s.* Suppose without loss of generality that for a top-L tuple (x, y, Z^1_{xy}), the singleton cluster (x, y, Z^1_{xy}) has been chosen in SC. Due to the incomparability condition, none of $(x, y, *), (x, *, *), (*, y, *)$ can belong to SC. Hence, to cover the other top-L tuple (x, y, Z^2_{xy}), one of $(x, y, Z^2_{xy}), (x, *, Z^2_{xy}), (*, y, Z^2_{xy})$ has to be chosen $(*, *, Z^2_{xy})$ cannot be chosen due to (A) above. However, these three clusters have distance 1, 2, 2 respectively from (x, y, Z^1_{xy}) violating the distance constraint $D = 3$.

(C) The clusters in SC cannot have one $*$*-s.* (i) Suppose for a top-L tuple (x, y, Z^1_{xy}), the cluster $(x, y, *)$ has been chosen in SC. Due to the incomparability condition, none of $(x, *, *), (*, y, *)$ can belong to SC, and any cluster with 1 or zero $*$ to cover top-L tuples from edges of the form (x, y') or (x', y) in G will have distance ≤ 2 with $(x, y, *)$, violating $D = 3$. If $(x, *, Z^1_{xy})$ is chosen (same for $(*, y, Z^1_{xy})$), to cover the other top-L tuple (x, y, Z^2_{xy}), one of $(x, y, Z^2_{xy}), (x, *, Z^2_{xy}), (*, y, Z^2_{xy})$ has to be chosen. Since the first two have distance 1 and 2 respectively from $(x, *, Z^1_{xy})$ violating the distance constraint $D = 3$, $(*, y, Z^2_{xy}$ must also belong to SC. However, $(x, *, Z^1_{xy})$ and $(*, y, Z^2_{xy})$ together rule out covering clusters for other top-L tuples from edges of the form (x, y') or (x', y) in G, which will have distance ≤ 2 from either of these two clusters, violating $D = 3$.

Hence the clusters in SC must be of the form $(x, *, *)$, $(*, y, *)$, or $(*, *, z)$, which corresponds to a vertex cover in G. Suppose there are K clusters in SC. Then the value of SC is $\frac{2N_e+0}{2N_e+K} \geq \frac{2N_e}{2N_e+M}$ by our assumption, hence $K \leq M$, and G has a vertex cover of size at most M. □

Theorem A.2. *The decision version of whether a feasible non-trivial solution exists for the problem defined in Definition 3.1 is NP-hard,even with three attributes, $D = 0$, $L = n$, and uniform weights of the elements*[4] *(proof in full version).*

Proof of Theorem A.2. The reduction is again from the problem of finding a minimum vertex cover in a tri-partite graph G with partitions (X, Y, Z) (see the proof of Theorem A.1).

Given such a graph G and a bound M, we construct a database instance S as follows. There are three attributes A_X, A_Y, A_Z. Any edge of the form $(x, y), x \in X, y \in Y$ forms a tuple (x, y, Z_{xy}), where Z_{xy} is a unique value of the A_Z attribute for the edge (x, y) in G. Similarly, an edge $(y, z), y \in Y, z \in Z$ forms a tuple (X_{yz}, y, z), and an edge $(x, z), x \in X, z \in Z$ forms a tuple (x, Y_{xz}, z). The total number of tuples in the relation S is n and each tuple has the same weight 1. We set $k = M$, $L =$ the number of edges in G, and claim that G has a vertex cover of size $\leq M$ if and only if S has a non-trivial solution, a set of clusters SC, of size $\leq k$.

(only if) Suppose G has a vertex cover T of size $\leq M$. For any $x \in X \cap T$, choose the cluster $(x, *, *)$ in SC; similarly for $y \in Y \cap T$ and $z \in Z \cap T$, choose the clusters $(*, y, *)$ or $(*, *, z)$ respectively in SC. Since T is a vertex cover, SC will cover all tuples in S and has size $\leq M = k$.

(if) Suppose S has a non-trivial solution SC of size $\leq k = M$. The clusters in SC can have $*$ in zero, one, or two positions (all three positions cannot be $*$ since SC is a non-trivial solution). We will argue that any cluster in SC can be replaced by a cluster with two $*$ and a vertex from G forming another feasible non-trivial solution without increasing the size of SC (the size may decrease). Consider any tuple of the form $t = (x, y, Z_{xy})$ in S (the other two cases (x, Y_{xz}, z) and (X_{yz}, y, z) follow similarly). If t is covered by a cluster of the form (x, y, Z_{xy}), $(x, y, *)$, $(x, *, Z_{xy})$, $(*, y, Z_{xy})$, or, $(*, *, Z_{xy})$, such clusters cannot cover any other tuple in S since Z_{xy} is unique, so replace such clusters by $(x, *, *)$ or $(*, y, *)$. After this is repeated for all tuples in S, the only types of clusters remaining in SC be of the form $(x, *, *)$, $(*, y, *)$, or $(*, *, z)$, which corresponds to a vertex cover in G, and the size of the solution has not increased. □

A.3 Omitted Pseudocodes from Sections 4.2 and 5

Algorithm 3 describes details of the `Fixed-Order` algorithm introduced in Section 4.2, and Algorithm 4 gives the pseudocode for the *delta judgment* optimization procedure from Section 5.

[4]The top-k original elements may not constitute an optimal solution for $D = 0$ when $L > k$.

Algorithm 3 The `Fixed-Order` algorithm

Require: Size, coverage, and distance constraints k, L, D

$O = \emptyset$.
for $i = 1$ to L **do**
  Let t_i be the i-th top element.
  if $t \in \text{cov}(O)$ **then**
    /* t_i *is already covered, do nothing.* */
  else if $|O| < k$ **then**
    if t_i is at distance $\geq D$ from all clusters in O **then**
      $O = O \cup \{t_i\}$.
    else
      Let P_D be the pairs of clusters $(C, \{t_i\})$ where $C \in O$ such that the distance between C, t_i is $< D$.
      Perform `UpdateSolution`(O, P_D).
    end if
  else/* *merge* t_i *with one of the clusters in* O */
    Let P be the pairs of clusters $(C, \{t_i\})$, $\forall C \in O$.
    Perform `UpdateSolution`(O, P).
  end if
end for
return O

Algorithm 4 The `Delta-Judgment` Procedure

Require: 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 before in the same round*/
else if $i = j + 1$ **then**
  /* *Do nothing. It is up-to-date.* */
end if
$v_{j+1} = (v_j \times |O_j| + \Delta_{sum})/(|O_j| + \Delta_{cnt}))$
/* *Update the round indicator* */
$i = j + 1$
return v_{j+1}

A.4 Qualitative Evaluation with Other Related Approaches

For the query in Example 1.1, we here compare the results adapting approaches in three related papers [9, 19, 24] that are most relevant to this work (*i.e.*, consider either summarization or diversification). Comparison with the MMR-based λ-parameterized diversification framework (no summarization) from [32] can be found in the full version due to space limitation. We use the brute-force or an efficient algorithms from these papers using $k = 4$, $D = 2$, and $L = 10$. As expected, the objectives proposed in these papers do not serve the purpose of summarizing aggregate query answers that we study in this paper.

A.4.1 Comparison with smart drill-down [19]. The goal in [19] is to find an ordered set of k rules (clusters with $*$ in our framework) R with maximum score $\texttt{score(R)} = \sum_{r \in R} \texttt{MCount(r, R)} \times \texttt{W(r)}$, where the marginal count $\texttt{MCount(r, R)}$ denotes the number of tuples covered by r but not covered by preceding rules in R, and $W(r)$ denotes the number of non-$*$ attributes in r. The two parameters focus on diversity (by preferring rules with high `MCount`) and good-ness (by preferring more specific rules) of rules. To compare it with our framework with *relevance* (summarizes aggregate results where tuples have values), we update the scoring function as $\texttt{score(R)} = \sum_{r \in R} \texttt{MCount(r, R)} \times \texttt{W(r)} \times \texttt{val(r)}$, where `val` denotes the average value of the elements covered by r that are uncovered so far by previous rules. Our framework also considered *coverage* of top-L elements, so we evaluated a greedy algorithm (shown to perform well in [19]) on two inputs: (i) all elements in the aggregate query results, and (ii) top-L elements and obtained the following results:

hdecade	agegrp	gender	occupation	avg score
Smart drill-down on top-10 elements				
*	20s	M	*	3.47
*	10s	M	Student	3.63
1995	30s	F	Educator	3.70
Smart drill-down on all elements				
1995	*	M	*	3.18
*	20s	M	*	3.47
1995	*	F	*	3.24
*	10s	M	Student	3.63

Comparing the above tables with Figure 1b and 1c, we see that the average score of the rules or clusters is much less than our output. Although the above rules capture `(20, M)` that is prevalent in the top-10 results in Figure 1a, it is not a characteristics of the top-valued elements; *e.g.*, elements ranked 44, 46, 49 (in total 18 out of 50 tuples) satisfy this rule leading to a low score.

A.4.2 Comparison with diversified top-k [24]. Given the elements with scores, using our terminology, the goal in [24] is to find at most k elements such that the distance between any two chosen elements is at least D and the sum of scores is maximized. This notion considers *diversity* and *relevance*. To also include coverage like ours, we ran it on top-L elements and got the following answers (by a brute-force implementation since the goal was qualitative evaluation). We show both actual value of the representative elements (`score`) as well as the average value of elements within distance $D-1$ of these chosen elements (`avg score`) below since intuitively the chosen elements cover these elements.

hdecade	agegrp	gender	occu.	score	avg score
Diversified top-k on top-10 elements					
1975	20s	M	Student	4.24	3.71
1980	20s	M	Programmer	4.13	3.77
1980	10s	M	Student	3.97	3.69
1995	30s	F	Educator	3.70	3.52

Although [9] optimizes for a different optimization goal, the above results illustrate that it does not perform well for our goal of

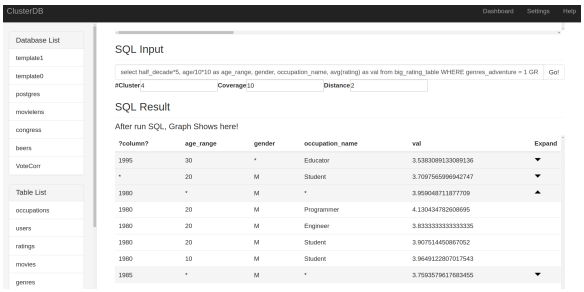


Figure 15: A snapshot of the GUI

summarization of top-valued elements with diversity. *First,* the average values of the "clusters" formed by the representative elements shown above are less than the values given by our approach in Figure 1b. These chosen values include the original top-10 elements within distance 1, but also include many elements with low values within such circle. For instance, the first element above covers 6 elements (including itself) and covers the 28th element with value 3.12. *Second,* the representative of a cluster is one of the original elements, and we are not getting summarized common properties of the cluster using $*$-attribute values.

A.4.3 Comparison with DisC diversity [9]. Given the distance parameter D and a set of elements P, the goal of [9] is to find a *DisC diverse subset* S^* of minimum size such that each element in P is at most distance D from some element in S^*, and no two elements in S^* are within distance D of each other. Like [24], this diversity notion naturally includes summarization, since an element can be assigned to the *cluster* corresponding to a point in S^* at distance $\leq D$. To also include the notion of *relevance* and *coverage of top-L*, we ran it too on top-L elements (a brute-force implementation) and obtained the following results. Like [24], [9] gives clusters with smaller scores than ours (*.e.g.*, the first tuple "covers" eight elements where the last one is of rank 28 and has value 3.31), and do not exhibit the common properties by $*$ values.

hdecade	agegrp	gender	occu.	score	**avg score**
DisC diversity on top-10 elements					
1980	20s	M	Student	3.91	3.81
1985	10s	M	Student	3.76	3.66
1995	30s	F	Educator	3.70	3.52
1985	20s	M	Engineer	3.65	3.62

A.5 GUI Snapshot

A snapshot of graphical user interface after the query is run and the clusters are displayed is shown in Figure 15. The user can put in a SQL query and their desired k,D and L values to get the ouput clusters.

A.6 Additional Related Work

In addition to the papers discussed in Section 8, diversification of query results has been extensively studied in the literature for both query answering in databases and other applications [1, 2, 4, 5, 8, 9, 11, 15, 17, 24, 25, 28, 32–34, 37, 38]. One of the formalisms to capture both *diversity and relevance* of a resultset is to balance these two objectives using a trade-off parameter λ specified by the user. This approach, called *MMR (Maximal Marginal Relevance)* aims to reduce redundancy while maintaining

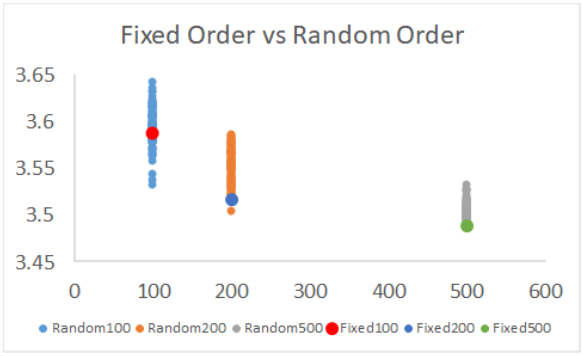


Figure 16: Random-Order vs. Fixed-Order (Section 7.5)

relevance of the chosen outputs for the input query, and was first used for re-ranking retrieved documents and in selecting appropriate passage for text summarization [5]. Gollapudi and Sharma [17] studied three variants of the objective function (max-sum, max-min, mono) based on the MMR criterion. Deng and Fan [8] studied the data complexity and combined complexity for these problems. Vieira et al. [32] conducted an experimental study of existing and new algorithms for the max-sum objective defined in [17] with some small modifications. Fraternali et al. [12] studied this objective for diversification of objects in a low-dimensional vector space. We compare results from Vieira et al. [32] with our work in the full version.

The *diversity* criterion has been studied algorithmically as the *facility dispersion problem* [25]. [4] studied the *max-sum dispersion problem* and the *max-sum diversification problem* (as in [17]) when the value of a subset of elements $w(S')$ is given by a monotone submodular function. Abbassi et al. [1] studied the diversity maximization of a set of points under matroid constraints. *Diverse skyline* [28] is another related direction.

Zhu et al. [37] proposed a ranking algorithm with applications in text summarization and social network analysis. Vee et al. [31] studied the problem of computing diverse query results for non-aggregate queries in online shopping applications. In the area of Information Retrieval (IR), Zheng et al. [36] studied search result diversification. using λ-parameterized MMR objective function [17, 32], but their "diversity score" is defined as the sum (over possible topics) of product of importance of a subtopic to the input query and how much a document covers this topic.

Chen and Li [6] considered the problem of categorizing query answers using clusters on a navigational tree by exploiting the query history of the users when different users have diverse preferences. Other approaches include relational data summarization [35] and Web table search taking into account schema/instance diversity, table popularity, and redundancy [23]. For result summarization and exploration in databases, Gebaly et al. [10] considered summarization of attributes using $*$ values to find factors that affect a binary (non-aggregate) attribute. Sarawagi explored (*e.g.*, [27]) sophisticated OLAP operators for helping the user visit unvisited interesting parts in a data cube.