Supporting Ranking and Clustering as Generalized Order-By and Group-By[*]

Chengkai Li[1] [†] Min Wang[2] Lipyeow Lim[2] Haixun Wang[2] Kevin Chen-Chuan Chang[1]
[1]Department of Computer Science, University of Illinois at Urbana-Champaign
cli@uiuc.edu, kcchang@cs.uiuc.edu
[2]IBM T.J. Watson Research Center
min@us.ibm.com, liplim@us.ibm.com, haixun@us.ibm.com

ABSTRACT

The Boolean semantics of SQL queries cannot adequately capture the "fuzzy" preferences and "soft" criteria required in non-traditional data retrieval applications. One way to solve this problem is to add a flavor of *"information retrieval"* into database queries by allowing fuzzy query conditions and flexibly supporting *grouping* and *ranking* of the query results within the DBMS engine. While ranking is already supported by all major commercial DBMSs natively, support of flexibly grouping is still very limited (*i.e.*, *group-by*).

In this paper, we propose to generalize *group-by* to enable flexible *grouping* (clustering specifically) of the query results. Different from clustering in data mining applications, our focus is on supporting efficient clustering of Boolean results generated at query time. Moreover, we propose to integrate ranking and clustering with Boolean conditions, forming a new type of *ClusterRank* query to allow structured data retrieval. Such an integration is nontrivial in terms of both semantics and query processing. We investigate various semantics of this type of queries. To process such queries, a straightforward approach is to simply glue the techniques developed for ranking-only and clustering-only together. This approach is costly since both ranking and clustering are treated as blocking post-processing tasks upon Boolean query results by existing techniques. We propose a summary-based evaluation method that utilizes bitmap index to seamlessly integrate Boolean conditions, clustering, and ranking. Experimental study shows that our approach significantly outperforms the straightforward one and maintains high clustering quality.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing, Relational databases*; H.2.8 [**Database Management**]: Database Applications; H.2.3 [**Database Management**]: Languages—*Query languages*

*This material is based upon work partially supported by NSF Grants IIS-0133199, IIS-0313260, the 2004 and 2005 IBM Faculty Awards. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies.

†Work partially performed while the author was visiting IBM.

Permission to make digital or hard copies of all or part 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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'07, June 11–14, 2007, Beijing, China.
Copyright 2007 ACM 978-1-59593-686-8/07/0006 ...$5.00.

General Terms

Algorithms, Performance, Experimentation

Keywords

retrieval, data exploration, ranking, top-k, query processing, clustering, grouping

1. INTRODUCTION

The ubiquitous usage of databases for managing structured data, compounded with the expanded reach of the Internet to end users, has brought forward new data "retrieval" scenarios: In new applications, such as E-commerce or multimedia, users want to find best-matching tuples over a database, with only "fuzzy" preferences and "soft" criteria– We refer to this demand of fuzzy relevance over structured data as *data retrieval*, to parallel the well-established *information retrieval* over unstructured text. While successful in traditional business environments, SQL-based querying has become increasingly inadequate for such new scenarios. As an important first step, many recent works have focused on *ranked* (or *top-k*) queries (*e.g.*, [9, 3, 8, 6, 1, 15, 7, 20, 14]). Toward a systematic support, while *ranking* conceptually generalizes *Order-By* into fuzzy ordering, this paper aims at generalizing *Group-By* into "fuzzy" grouping– or clustering– to form a more complete suite of solutions for data retrieval.

Example 1 (Motivating Scenario: House Search): Consider user Amy looking for a house in Chicago, from a database `House`(id, price, size, zipcode, longitude, latitude, rating). She will consider houses priced below $300k, and is willing to pay a little more *if* the size is large. She would like to consider different areas– She will accept even more expensive choices *if* they are near the lakeshore; but *if* there is no such choices, she would like to look for better prices in the suburb. Notice the many "if"s in her preference, which depend on what the database can offer. Will a SQL-based database, such as *realtor.com*, support her "exploration" by querying effectively? ■

As Example 1 illustrated, data retrieval essentially mandates *result exploration*, for users to explore what choices are available in the database, and how they match the query criteria. We ask: What functions shall we support for such exploration?

While we want to equip SQL-based querying with such exploration, the answers seem to, interestingly, lie in the design of SQL itself: With the test of time, SQL has proven to be a well-designed, compact suite of constructs, balancing both functionality and simplicity. Can we draw inspiration from SQL constructs to support data retrieval, in particular, to explore query results? For this pur-

pose, **order-by** and **group-by** both stand out as the pillars for organizing results– *e.g.*, for our example House relation, we may

$$\textbf{group-by}\ \text{zipcode}\quad \textbf{order-by}\ min(\text{price}). \qquad (S_1)$$

By ordering and grouping on attribute values, RDBMS can organize results for tabular presentation and report generation.

To begin with, many recent works have attempted to generalize **order-by** beyond "crispy" result ordering– For the same syntactic construct, these efforts seek to generalize the semantics from *ordering* of attribute values to *ranking* of matching qualities. Therefore, from ordering to ranking, the generalization boils down to 1) supporting fuzzy scoring functions (instead of only attributes) and 2) targeting at only *top-k* (partial) results (instead of total ordering). For our House example, we may order by some preferred balance of price, size, and realtor rating, and look for only the top 10 results:

$$\textbf{order-by}\ \frac{\text{size}}{\text{price}} \cdot \text{rating}\quad \textbf{limit}\ 10 \qquad (S_2)$$

Drawing these insights from SQL constructs, as **order-by** has been generalized into ranking, as a parallel step toward result exploration, we propose to generalize **group-by**, which this paper focuses on. Just like from ordering to ranking, we believe current grouping has two major limitations: *First*, the prerequisite of data understanding: As a dilemma, while grouping should help users to learn the data distribution of available choices in the database, in its current semantics, users must know this "distribution" in order to specify a good grouping scheme– For instance, is **group-by** zipcode meaningful? (How if there are 1000 different zipcodes, thus 1000 groups?) *Second*, the limitation of equality partitioning: Inherited from SQL's "crispiness," current grouping semantics partitions the space only by "identical" values. For instance, is **group-by** longitude, latitude meaningful? (As no houses will share the same coordinates, we should instead group by their proximity.)

Our solution, much like from ordering to ranking, is to generalize "crispy" grouping to "fuzzy" grouping– or *clustering* [16, 13]. As a well-established technique for data exploration, in abstraction, with input of attributes $c_1, \ldots, c_m$ and a result size of t, clustering will output t groups, or *clusters*, that best partition the space according to how objects are *similar* in $c_1, \ldots, c_m$ (instead of strict equality of values). It thus ameliorates the two limitations simultaneously: For the input specification, users simply specify the desired number of t clusters, much like the desired result size k in *top-k* ranking, and the system will automatically weigh in the data distribution to generate t clusters. (So even if there are 1000 zipcodes, they will be grouped into a small number of t clusters.) Further, as the grouping criteria, clustering will form partitions by data distribution. Similar objects that do not share strictly identical values in $c_1, \ldots, c_m$ will still be grouped. (Thus grouping on longitude and latitude will put together houses in similar locations.)

This clustering, or fuzzy grouping, maybe expressed[1] as follows, with an additional "**into** t" to indicate the target number of groups that the fuzziness should achieve. For our example, a user may want to cluster houses into 5 groups by their location proximity:

$$\textbf{group-by}\ \text{longitude, latitude}\quad \textbf{into}\ 5 \qquad (S_3)$$

This paper will examine this generalization from grouping to clustering, for supporting data retrieval with SQL. What should be the "fuzziness" of grouping– or, what clustering algorithms should we assume? Ideally, in a comprehensive setting, the system shall support a set of clustering schemes as operators to choose, and even allow extensions by, say, external functions. However, as a first step to start with, and to focus on the essence of the problem, this paper will study *k-means* as the clustering scheme, because it is the most well-known and widely applicable partitioning-based clustering method [13] and is by far the most popularly used method in scientific and industrial applications [2]. Our framework can apply other distance-based clustering methods, as long as the distance or similarity functions are based on the proximity of attribute values. Section 4.2 will discuss such extensions. Thus, in a more general setting, to specify our choice of clustering, we may express it as:

$$\textbf{group-by}\ \textit{k-means}(\text{longitude, latitude})\quad \textbf{into}\ 5 \qquad (S'_3)$$

Putting together, in a SQL-like syntax, we may express the complete suite of clustering and ranking in the following form.

select	$\ldots$		
from	$T_1, \ldots, T_s$		
where	$\mathcal{B}(b_1, \ldots, b_h)$		
group by	$c_1, \ldots, c_m$	**into**	t
order by	$\mathcal{F}(r_1, \ldots, r_n)$	**limit**	k

In this complete form, our generalization boils down to two challenges: *First*, for *the context of* SQL, as SQL has been well developed for managing structured data, our clustering must integrate with the core Boolean constructs. What does such integration mean? As **group-by** is meant to execute after the **where** clause with Boolean selection or join, our clustering should similarly partition with respect to the "dynamic" result $\sigma_{\mathcal{B}}(T_1 \times \ldots \times T_s)$. That is, the dynamic Boolean result, instead of the static tables, is the "population" whose data distribution will define the clusters. How can we execute clustering efficiently after such dynamic filtering?

Second, for *our objective of data retrieval*, as both pillars of result exploration, clustering and ranking must be seamlessly integrated. What does such integration mean? In standard SQL the combination of **group-by** and **order-by** will lead to ordering among groups (thus S_1 will return groups ordered by their minimal prices). While such *order-among-groups* is a useful semantics, we believe it is equally (if not more) important to support *order-within-groups*. In data retrieval scenarios, as evident from similar functions for text retrieval (*e.g.*, Web search), when clustering and ranking are combined (*e.g.*, *vivisimo.com*), clustering will partition the results into alternative groups, and ranking then orders answers within each group. In fact, for crispy **group-by**, this order-within-groups semantics can be realized in OLAP *functions*[2] [30], which was introduced in SQL-99 and supported by major RDBMSs. Thus, for combining ranking and clustering, we consider different groups as equal alternatives, and only those answers within the same group are compared in ranking. For example, we may cluster houses by areas (as in snippet S_3) and only rank among those in the same area by prices and sizes (as in S_2). Can we support such integration of ranking "within" clustering efficiently?

In this paper, we propose to support clustering and ranking together, with the order-within-groups semantics, as a generalization of **group-by** and **order-by** to support fuzzy data retrieval applications. Our solutions are built upon summary-based clustering and ranking using dynamically constructed data summary, incorporating Boolean conditions at query time. We have implemented this framework by utilizing bitmap index to construct such summary on-the-fly and to integrate Boolean filtering, clustering, and ranking. Our results show that this approach significantly outperforms a straightforward approach that is available in current database systems. In summary, this paper makes the following contributions:

- **Concept: Generalizing Group-By for Fuzzy Grouping**. We propose to support clustering with SQL as a generalization for

[1] We focus on the generalization of "functionality" and not syntax. The same can be expressed with OLAP functions. See footnote 2.

[2] In DB2, $rank()$ **over** (**partition by** $attr_1, \ldots, attr_m$ **order by** v) groups tuples by $attr_i$ and orders tuples in each group by v.

group-by, parallel ranking for **order-by**. Moreover, to the best of our knowledge, ours is the first in the literature to propose the integration of fuzzy grouping and ranking in relational databases.

- **Framework: Summary-based Processing**. We develop on-the-fly summary construction and summary-based clustering and ranking, for efficient query support.

The rest of the paper is organized as follows. We review the related work in Section 2. In Section 3, we define a new type of queries, explore its semantics in supporting fuzzy ranking and grouping, and discuss the challenges in processing such queries. We give an overview of our solutions in Section 4. We further present the detailed data structure and algorithms in Section 5 and optimization heuristics in Section 6. The experimental results are discussed in Section 7. Section 8 concludes the paper.

2. RELATED WORK

Clustering has been extensively studied for years in many areas including machine learning, pattern recognition, and data mining [16, 13]. However, in typical data mining setting, analytical tasks such as clustering is more or less an infrequent or one-shot operation, over a static dataset, and performed by a small number of analysts. In contrast, the fuzzy clustering engaging us in this paper is a day-to-day operation, upon dynamic results of Boolean conditions, over different clustering attributes, and requested by a large number of users. Therefore our focus is to efficiently support such *on-the-fly* clustering and yet maintain high quality of clustering. Moreover, we integrate clustering with Boolean filtering and ranking. The framework proposed in this paper is not aiming to replace the existing clustering algorithms. Instead, we simply adopt existing algorithms and focus on how to cluster over data summary dynamically constructed.

Various clustering algorithms exploited summary of data during clustering, *e.g.*, [31, 11]. In particular, there are grid-based clustering and data mining algorithms such as STING [27] and WaveCluster [25]. They pre-compute and store the grids beforehand. Our summary-based clustering shares the same insight of clustering by the unit of bucket instead of individual tuple. However, we emphasize on the need in our target applications to construct the grid *on-the-fly* for coping with dynamic Boolean conditions, clustering attributes, and ranking attributes that are specified at query time, and to integrate filtering, clustering, and ranking altogether.

Ranking (*top-k*) query has gained great interests in the database field recently (*e.g.*, [9, 3, 8, 6, 1, 15, 7, 20, 14]). Note that the *top-k* processing techniques in the literature may not be applicable in our scenarios. It is well known that *top-k* algorithms are optimized for small k. As k increases, their performances degrade quickly and become worse than straightforward materialize-then-sort approach. In our case, we must get the top k tuples within each cluster, some of them may be globally ranked low. For example, the houses of one region in general may be more expensive and smaller than the ones in other regions, therefore even the top houses in this region are ranked quite low globally. That does not mean the houses in the region are bad choices. In fact, the reality may be the opposite since, say, that region is safe and scenaric. Using *top-k* algorithms under this situation will not be beneficial.

Note that an idea of using candidate buckets for pruning was applied in answering *top-k* queries [6]. However, they rely on static pre-collected multi-dimensional histogram, while we utilize data summary that is dynamically constructed using bitmap index.

An automatic method for categorizing query results (instead of clustering) is proposed in [4]. They perform categorization as post-processing after Boolean query results are obtained, with the focus on minimizing users' navigation overhead. They do not consider integration with ranking either. The idea of categorizing database query results is also exemplified by the products from *Endeca*.

The idea of combining clustering and ranking has been proposed for organizing the results of Web search engines (*e.g.*, *vivisimo.com*), as well as for information retrieval systems [19]. To the best of our knowledge, our work is the first to investigate the problem in the setting of RDBMS, which has intrinsically different data and query processing model and thus presents significant new challenges.

3. OUR PROPOSAL: CLUSTERING + RANKING IN DATABASE QUERIES

3.1 The *ClusterRank* Query

In this paper, we introduce a new type of *ClusterRank* query. The semantics of such a query is to conceptually perform the following three steps. Note that we ignore less important operations in our context such as attribute projection.

- **Filtering**: Upon a base relation or the Cartesian product of base relations, we apply Boolean function $\mathcal{B}$, resulting in a relation of qualifying tuples, $\sigma_{\mathcal{B}}$;
- **Clustering**: The tuples in $\sigma_{\mathcal{B}}$ are partitioned into t clusters, based on the clustering attributes $c_1, \ldots, c_m$;
- **Ranking**: A scoring function $\mathcal{F}$ defined over a set of ranking attributes R assigns a ranking score $\mathcal{F}(R)[t]$ to each tuple t. Within each cluster, the top k tuples with the highest scores (or all if there are less than k tuples in the cluster) are returned. [3]

In relational database, currently no SQL syntax can support such queries, nor can OLAP functions express our query. Still, since OLAP functions support ranking within a group or a partition, the closest way to express our semantics maybe the following:

select ..., rank() **over** (**partition by** *k-means*$(t, c_1, \ldots, c_m)$
order by $\mathcal{F}(r_1, \ldots, r_n)$
) **as** $score_rank$
from $T_1, \ldots, T_s$
where $\mathcal{B}(b_1, \ldots, b_h)$
when score_rank $<= k$

Besides the fact that OLAP does not support functions such as *k-means*$(t, c_1, \ldots, c_m)$ in the **partition by** clause, there is a fundamental difference between the task achieved by the above query and the goal we want to achieve. The query treats *k-means*$(t, c_1, \ldots, c_m)$ as a black box, which prevents the system from optimizing the query. Instead, we focus on integrating the ranking and the clustering process in a tight manner, so that we can minimize the cost of the *ClusterRank* query.

In essence, our semantics is based on the concept of fuzzy clustering. We require that partitions have fuzzy boundaries, and we specify the total number of clusters, as in K-means. Borrowing the syntax of SQL we denote fuzzy clustering by "**group by** ... **into** ...", and our goal is to integrate it with the "**order by** ... **limit** ..." clause. The sketch of such a query is shown below [4].

select	...	
from	$T_1, \ldots, T_s$	
where	$\mathcal{B}(b_1, \ldots, b_h)$	
group by	$c_1, \ldots, c_m$	**into** t
order by	$\mathcal{F}(r_1, \ldots, r_n)$	**limit** k

[3]When there are ties in scores, an arbitrary deterministic "tie-breaker" can determine an order, *e.g.*, by unique tuple IDs.

[4]For simplicity, we assume **order by asc|desc** uses descending order as default, although ascending is the default in some systems.

More formally, a *ClusterRank* query Q is a SPJ query augmented with clustering and ranking conditions. The query consists of the following tables, attributes, functions, and constants.

- $\mathcal{T}$: a set of tables $\{T_1, \ldots, T_s\}$;
- $\mathcal{B}$: a *Boolean function* $\mathcal{B}$ over a set of attributes $b_1, \ldots, b_h$. The Boolean function $\mathcal{B}$ can be a complex Boolean condition such as conjunctions and disjunctions of sub-conditions;
- $c_1, \ldots, c_m$: a set of *clustering attributes*
- t: the number of clusters;
- $\mathcal{F}$: a *ranking function* (a.k.a. *scoring function*) over the *ranking attributes* $r_1, \ldots, r_n$;
- k: the number of top tuples to retrieve within each cluster.

We want to point out that clustering, by nature, is a fuzzy and unstable operation, as different algorithms and configurations on the same data will generate different clusters. Therefore on the one hand, in contrast to the deterministic semantics of conventional database queries, a *ClusterRank* query may generate different answers in each run. On the other hand, such non-determinism is consistent with our goal of enabling fuzzy data retrieval and exploration, and we believe sacrificing the crispness of queries is worthy.

Note that the above syntax is for illustrating our concept only, as the use of SQL's **group by** has many restrictions. The main reason is that when **group by** is present, the columns in **order by** must either appear in the columns of **group by**, or be some aggregate functions. The meaning of such a query is to order the groups based on some grouping attributes or aggregate values over the groups. Moreover, we will not be able to specify the number of clusters desired, and **group by** does not allow function either.

Up till this moment, we have assumed only one semantics for *ClusterRank* queries, that is, returning top k tuples within each cluster. (Call it *global clustering/local ranking*.) However, we may extend our query model to embrace a richer set of semantics, tailored for various application needs. One example is *local clustering/global ranking*, where the clustering is only performed over the global top k tuples instead of $\sigma_{\mathcal{B}}$. Another example is *global clustering/global ranking*, where within each cluster, only those tuples that belong to the global top k (instead of local top k) are returned. Moreover, we may further allow ranking of the clusters by aggregate functions. While it is very interesting to study these alternative semantics and corresponding techniques for processing queries, we focus on *global clustering/local ranking* in this paper.

3.2 Challenges: The Problems with a Straightforward Approach

Literally following the semantics in Section 3.1, we obtain a straightforward approach for evaluating *ClusterRank* queries. That is to, (1) materialize intermediate Boolean results $\sigma_{\mathcal{B}}$; (2) cluster $\sigma_{\mathcal{B}}$; (3) sort all the tuples within each cluster; and (4) return the top k tuples in each cluster.

However, such *materialize-cluster-rank* approach is clearly an overkill due to the fact that it clusters and ranks all Boolean results although we only need the top k in each cluster. It can thus be very inefficient. Materializing $\sigma_{\mathcal{B}}$ itself can be expensive, especially with joins. The cardinality of $\sigma_{\mathcal{B}}$ can be large when Boolean conditions are not selective. As a costly procedure, clustering such a large $\sigma_{\mathcal{B}}$ is expensive and may take multiple iterations. Sorting the tuples in each cluster further adds to the overhead. Moreover the tuples may be dumped out and read in many times, between materializing $\sigma_{\mathcal{B}}$ and clustering, during the iterations in clustering, and for sorting them. All these result in significant disk I/O cost.

The high overhead of materialize-cluster-rank can seriously impact the usefulness of *ClusterRank* queries. It may be acceptable if the query was only one-shot, where the clustering and ranking results, or at least $\sigma_{\mathcal{B}}$, can be even materialized beforehand. This is clearly not the case in our target applications (*e.g.*, house search in Example 1), where users *on-the-fly* specify all kinds of Boolean conditions, form clusters upon different attributes, and apply different ranking criteria over different ranking attributes.

4. FRAMEWORK: OVERVIEW

In this section, we first give a high-level overview of our approach (Section 4.1), then specify the data and query model and assumptions (Section 4.2), and finally briefly introduce the background on bitmap index (Section 4.3).

4.1 Our Approach: Summary-Based *ClusterRank*

The materialize-cluster-rank approach in Section 3.2 is very costly since it involves a large amount of tuple-based operations. For clustering, the approach goes through every tuple and assigns it to its closest cluster, for iterations until the algorithm converges. For ranking, it computes the score of each tuple and sorts all the tuples in each cluster. Obviously, it obtains the clustering and ranking results for each tuple. However, the query requests only a small portion of the tuples processed, *i.e.*, the top k within each cluster. A natural question to ask is: To reduce the cost in processing each tuple individually, can we process at a "coarser" level?

Using appropriate data summary instead of all tuples in both clustering and ranking is our answer to this question. Below we outline the summary-based *ClusterRank* approach. For *clustering*, with any distance-based method, if two tuples are close enough to each other, it is natural to assign them to the same cluster. In our approach, we use a grid-based data summary to put similar tuples into the same "bucket" and then cluster at the bucket-level. To be more specific, we perform partitioning (or binning) on each clustering attribute. The intersection of the bins over the clustering attributes gives us a summary grid with buckets. If two tuples fall into the same bucket (*i.e.*, the same bin along each clustering attribute), we can consider them the "same" tuple, *i.e.*, inseparable. Thus a bucket is the smallest unit in our clustering. As long as the bucket size is appropriate, the quality of clustering on the buckets is comparable to that on the original tuples. However, the bucket-level clustering is much more efficient than the tuple-level one, since the number of buckets is much smaller than the number of tuples.

For *ranking*, we can use a summary grid for efficient processing as well. For each cluster, the grid for the tuples in the cluster is constructed over the ranking attributes. For the tuples in each bucket, the upper-bound and lower-bound of their scores can be computed based on the boundaries of the corresponding bins on individual attributes. The bounds enable us to prune those buckets that do not contain any of the top k tuples. The top k in the unpruned candidate buckets are guaranteed to be the top k among all the tuples.

The clustering and ranking operate on two orthogonal summary grids built over clustering and ranking attributes, respectively. Note that the grids are query dependent since different queries may have different clustering and ranking attributes. Thus how to efficiently construct the grids *on-the-fly* at query time is one big challenge. Also, the clustering and ranking are on the results of Boolean conditions, thus we must integrate the Boolean filtering, clustering, and ranking in an efficient way.

We use bitmap indexes to meet the challenge and the integration goal. A bitmap index uses one vector of bits to indicate the membership of tuples for each value or each value range on an attribute. By intersecting the bit vectors for the bins over the individual clustering attributes, we construct the summary grid for clustering. The grid for ranking is constructed similarly. In summary, the bit vec-

tors serve as the basic unit in unifying Boolean filtering, clustering, and ranking through the following steps: (1) Bit vectors are used to process the Boolean conditions, (2) The resulting bit vectors are used in building the summary grid for clustering, (3) Clustering is performed on the grid, (4) The resulting bit vectors corresponding to each cluster are used in constructing the summary grid for ranking, and (5) Ranking is performed within each cluster.

4.2 Data and Query Model

We assume the tables have a snowflake-schema, consisting of one fact table and multiple dimension tables. There are multiple dimensions, each of which is described by a hierarchy, with one dimension table for each node on the hierarchy. The fact table is connected to the dimensions by foreign keys. The tables on each dimension are also connected by keys and foreign keys. As a special case of snowflake-schema, star-schema has only one table on every dimension, thus no hierarchy.

With respect to the *ClusterRank* queries in Section 3.1, we make the following assumptions on the Boolean, clustering, and ranking conditions.

- $\mathcal{B}(b_1, \ldots, b_h)$: The Boolean condition consists of conjunctive key and foreign-key joins and range selections, including two-side range selection (*e.g.*, $10 \leq a$ AND $a<20$, or $10 \leq a < 20$), one-side range selection (*e.g.*, $a \leq 20$), and equality selection (*e.g.*, a=10). Both sides of the two-side selection condition can be either open-end or closed-end. Note that one-side and equality selections are extreme cases of two-side selection.
- $c_1, \ldots, c_m$: The clustering attributes are all numerical attributes. We assume a K-means clustering algorithm. Note that the summary-based approach can be applied to other distance-based clustering algorithms, as long as the distance function is based on the proximity of attribute values (thus the insight of considering the tuples in the same bucket inseparable is applicable). The only difference observed by the clustering algorithms is that the buckets instead of real tuples are clustered. Therefore the number of tuples in the buckets, or their weights, must be taken into consideration. The algorithms can be simply adjusted to consider such weights [31]. In general, applicable algorithms can be supported as clustering operators to choose, or even registered as external functions. The algorithms may require parameters such as stopping criteria and distance functions (*e.g.*, Euclidean or Manhattan distances). These parameters can be specified through configuration settings in database systems.
- $\mathcal{F}(r_1, \ldots, r_n)$: The ranking function is *monotonic* over numerical ranking attributes, as commonly assumed in ranking query processing [9]. Without losing generality, in this paper we focus on the weighted-sum, a typical monotonic function. Note that our approach in fact is valid for any monotonic ranking function.

Under these assumptions, the sketch of the resulting simplified query is shown below.

select	*		
from	$T_1, \ldots, T_s$		
where	$v_1^1 \leq b_1 \leq v_1^2$ **and** ... **and** $v_p^1 \leq b_p \leq v_p^2$		
	and $b_{p+1} = b_{p+2}$ **and** ... **and** $b_{h-1} = b_h$		
cluster by	$c_1, \ldots, c_m$	**into**	t
order by	$w_1 \times r_1 + \ldots + w_n \times r_n$	**limit**	k

4.3 A Review of Bitmap Index

Bitmap index [22, 24] is an efficient indexing structure in OLAP and decision support applications. The usefulness of bitmap index has been realized and it is implemented in commercial database engines. The bitmap index over an attribute consists of a set of bit vectors, one vector per unique value of the attribute. The length of each bit vector equals the number of tuples, *i.e.*, the cardinality of the indexed relation. Regarding the bit vector for a value v on attribute a, its ith bit is set (*i.e.*, 1) if the ith tuple of the relation has value v on attribute a, otherwise 0. Bit-wise operations on bit vectors such as AND, OR, XOR, and NOT are very efficient and can be even supported by architecture features. Therefore complex selection queries can be answered efficiently by bitmap indices. Moreover, typical aggregate values such as COUNT can also be efficiently obtained.

Building one bit vector per attribute value makes the storage and maintenance cost of bitmap index prohibitive when high-cardinality attributes are indexed. To address this problem, various encoding schemes for bitmap index are proposed in the literature, *e.g.*, [5, 29]. For example, instead of using one bit vector for each unique attribute value, a vector can be built for a value range, *i.e.*, the partitioning or binning mentioned in Section 4.1.

5. REALIZATION: DATA STRUCTURE AND ALGORITHMS

Building on the insights provided in the overview (Section 4.1), we present the detailed algorithms in this section. In Section 5.1, we first give a formal description of summary grid, and show how to construct the grid using bitmap index. We then introduce the summary-based algorithms for clustering (Section 5.2.1) and ranking (Section 5.2.2). To simplify the discussion, our discussion first focuses on single table queries without Boolean conditions. In Section 5.2.3 we investigate how to extend our framework to incorporate selection and join conditions.

5.1 Data Structure: Building Summary Grids

Consider a relation T. A *partitioning attribute* a over T has a set of disjoint ranges that partition the value domain of a. More formally, a has y *partitioning points* $\{a^1, \ldots, a^y\}$ and two special *endpoints* a^0=min_a and a^{y+1}=max_a. The endpoints give the domain of a. That is, $[min_a, max_a)$ subsumes the a values of all the instances in T. The partitioning points and endpoints together define y+1 ranges over a, that are $ranges$=$\{range^0, \ldots, range^y\}$, where $range^i$=$[a^i, a^{i+1})$. Given a set of x partitioning attributes A=$\{a_1, \ldots, a_x\}$, their partitioning ranges determine a *grid* $\mathcal{G}(T, A, \{ranges_1, \ldots, ranges_x\})$. [5] The grid partitions the multi-dimensional space over A into z=$\prod_i (y_i+1)$ buckets $\mathcal{B}$=$\{B_1, \ldots, B_z\}$ [6], where each bucket is given by the ranges over A, one range per attribute. More formally, the subscription of a bucket is determined by the subscriptions of the corresponding ranges. That is, B_{id}=$<range_1^{id_1}, \ldots, range_x^{id_x}>$ = $<[a_1^{id_1}, a_1^{id_1+1}), \ldots, [a_x^{id_x}, a_x^{id_x+1})>$, where $id = \sum_{i=1}^{x-1} (id_i \times \prod_{j=i+1}^{x} (y_j + 1)) + id_x$. A bucket in the grid thus represents the intersections of the corresponding ranges. By partitioning the multi-dimensional space, the grid also partitions the tuples of T into the z buckets. That is, B_{id} = $\{t | t \in T \bigwedge t.a_i \in [a_i^{id_i}, a_i^{id_i+1}), \forall i\}$. A *summary grid* $\mathcal{S}_\mathcal{G}$ has $|B_{id}|$, the cardinality (*i.e.*, the number of tuples) of each B_{id} in $\mathcal{G}$.

Example 2 (Summary Grid): Figure 1 shows a grid over 10 tuples $t_1, \ldots, t_{10}$ by partitioning attributes $\{a, b\}$. The ranges on a and b are $ranges_a$ = $\{[0, 3), [3, 6), [6, 9)\}$ and $ranges_b$ = $\{[0, 3), [3, 6), [6, 9)\}$. There are 9 buckets, $B_0, \ldots, B_8$. The ID of each tu-

[5] We will often use the simplified notation $\mathcal{G}(T, A)$ when there is no need to emphasize the ranges in the context.

[6] For the ease of presentation, we abuse the notation $\mathcal{B}$ to denote buckets, although $\mathcal{B}$ denotes Boolean conditions in Section 3 and 4.

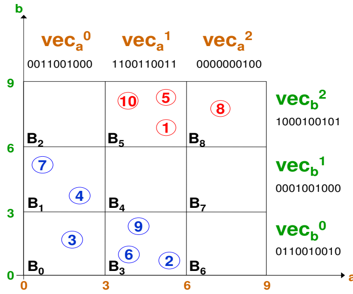


Figure 1: Summary Grid.

ple is shown inside its bucket. For instance, B_5=$<range_a^1$,$range_b^2>$ = $<[3, 6)$, $[6, 9)>$. It has 3 tuples (t_1, t_5, t_{10}). ■

To construct a summary grid over a relation T by a set of partitioning attributes A, we may simply go through all the tuples in T. In other words, we fully scan T if T is a base table or fully materialize T if T is the (join) query result over base table(s). As motivated in Section 3.2, our goal is to avoid such materialize-cluster-rank approach. We thus propose a novel method to construct summary grids by intersecting bitmap index. This method not only is efficient in building the summary grids, but also benefits the clustering and ranking operations based on the grids.

Given a set of partitioning attributes A=$a_1, \ldots, a_x$, we require the existence of a bitmap index I_i over each a_i, which contains y_i+1 bit vectors $\{vec_i^0, \ldots, vec_i^{y_i}\}$, corresponding to the y_i+1 ranges over a_i, $ranges_i$. Each vector vec_i^j is a sequence of $|T|$ bits, where the k-th bit is 1 if the value of attribute a_i in the k-th tuple of relation T is within $range_i^j$, the (j+1)-th range of a_i, otherwise 0.

As a bucket in the summary grid represents the intersections of the corresponding ranges, we can obtain the members in a bucket by intersection (*bit-and* operation, *i.e.*, &) of the bit vectors for the ranges. To be more specific, consider a bucket B_{id} = $<range_1^{id_1}$, . . ., $range_x^{id_x}>$, we aim to construct a bit vector $vec_{B_{id}}$ that contains $|T|$ bits, where the k-th bit is 1 if the k-th tuple of T belongs to the bucket B_{id}, otherwise 0. It is thus obvious $vec_{B_{id}}$ = $vec_1^{id_1}$ & . . . &$vec_x^{id_x}$. The set bits (*i.e.*, 1 bits) in $vec_{B_{id}}$ give the IDs of the tuples that fall in the bucket. Moreover, it is easy to obtain the cardinality of the bucket B_{id} by counting the number of set bits (*bit-count* operation, *i.e.*, #) in the resulting vector $vec_{B_{id}}$. That is, $\#vec_{B_{id}}$=$|B_{id}|$.

Example 3 (Constructing Summary Grid): Continuing with Example 2 and the grid in Figure 1, we obtain the members in each bucket by intersecting the corresponding bit vectors over attributes a and b. For instance, vec_{B_5} = vec_a^1 & vec_b^2 = 1100110011 & 1000100101 = 1000100001. The 1st, 5th and 10th bits in vec_{B_5} are set, indicating that B_5=$\{t_1, t_5, t_{10}\}$. ■

5.2 Algorithms

5.2.1 Summary-Based Clustering

With the summary grid, we are able to cluster much more efficiently. The key idea is to cluster the buckets in data summary and assign the tuples in the same bucket to the same cluster.

Given a set of tuples T to be clustered and the clustering attributes C=$\{c_1, \ldots, c_m\}$, we obtain the summary grid $S_{\mathcal{G}(T,C)}$ using C as the partitioning attributes. Associated with each bucket is a *virtual point*, located at the center of that bucket. We approximate the tuples in the bucket as a set of identical tuples at the virtual point, with the number of identical tuples equaling the cardinality of the bucket. Such approximation is based on the intuition that the tuples inside the same bucket are close enough to each other if the grid is fine-grained enough, so that their differences can be ignored without introducing significant impacts to the clustering results.

Procedure
1: choose k virtual tuples as the initial cluster centroids;
2: **repeat**
3: assign each virtual tuple to its closest cluster, with weight n, as if n identical copies are assigned into the same cluster;
4: update the centroid of the clusters;
5: **until** the clusters converge;

Figure 2: Weighted K-means Algorithm.

We apply clustering on the virtual points. The algorithms are similar to the conventional clustering algorithms, except that the algorithms must take into consideration the weights of the virtual points, where the weight of a virtual point is the cardinality of the corresponding bucket. For instance, in the weighted K-means algorithm (Figure 2), when the virtual point of a bucket with n tuples is inserted into a cluster, the centroid of the cluster is updated as if n identical points are inserted. Note that such simple weighted K-means extension has been used in various data mining and machine learning applications [17, 21], although the "weight" in their situation has different meaning.

With such adaptation, the algorithm continues for multiple rounds, as centroids are updated and virtual points are reassigned, until the clusters converge. At the end, the virtual points (*i.e.*, the buckets and thus the corresponding original tuples) are grouped into t clusters. The union (*bit-or* operation, *i.e.*, |) of the vectors for the buckets in the same cluster gives us the members in that cluster.

Example 4 (Weighted K-means): Continue our running example in Figure 1. Consider the case when we partition the 10 tuples into 2 clusters, using a and b as the clustering attributes. Suppose at the beginning we choose $<4.5, 7.5>$ (the virtual point of B_5) and $<1.5, 1.5>$ (the virtual point of B_0) as the initial centroids of $cluster_1$ and $cluster_2$, respectively. Then the virtual points of all the buckets are inserted into their closest clusters. Suppose virtual point $<4.5, 7.5>$ with weight 3 (since there are 3 tuples in B_5) is inserted into $cluster_1$ first. Later $<7.5, 7.5>$ with weight 1 (the virtual point of B_8) is inserted into $cluster_1$. The centroid of $cluster_1$ is changed to $(5.25, 7.5)$, because $5.25 = (4.5 * 3 + 7.5)/(3 + 1), 7.5 = (7.5 * 3 + 7.5)/(3 + 1)$.

Suppose, when the clustering algorithm in Figure 2 ends, *i.e.*, the clusters converge, the two clusters are $cluster_1$=$\{B_5, B_8\}$ and $cluster_2$=$\{B_0, B_1, B_3\}$. The union of vec_{B_5} and vec_{B_8} thus gives us the members of $cluster_1$. That is , $vec_{cluster_1}$ = vec_{B_5} | vec_{B_8} = 1000100001 | 0000000100 = 1000100101. Therefore $cluster_1$ contains 4 tuples, t_1, t_5, t_8, and t_{10}. The members for $cluster_2$ can be similarly obtained, as $vec_{cluster_2}$=0111011010. ■

Compared with clustering the original tuples, the summary-based clustering has clear advantages, as only one virtual point is needed for a large number of tuples in the same bucket. The number of virtual points can be much smaller than the number of original tuples. This reduction of data size saves not only the CPU cost in assigning tuples to clusters, but also more importantly the I/O cost in scanning the tuples from base tables or intermediate relations. More importantly, such a summary-based method allows us to integrate clustering and ranking seamlessly, as we shall see in Section 5.2.2.

5.2.2 Summary-Based Ranking

The structure of summary grid can be used in ranking as well. The essence of the idea is that we can prune most of the tuples that are bound to be outside of the top k tuples and zoom into the

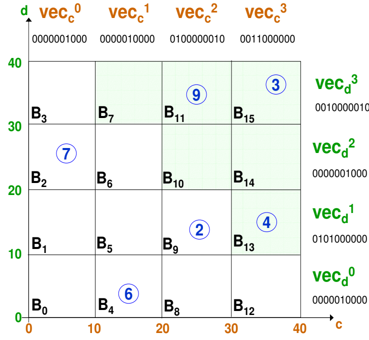


Figure 3: Summary-Based *top*-k Ranking.

candidate tuples, based on the upper-bound and lower-bound scores of the tuples within each bucket. For a bucket, such bounds are derived from the corresponding ranges of the partitioning attributes on the bucket. The details are given below.

Given a set of tuples T to be ranked and the ranking function $\mathcal{F}$ over the ranking attributes R=$\{r_1, \ldots, r_n\}$, we obtain the summary grid $S_{\mathcal{G}(T,R)}$ using R as the partitioning attributes. The bit vector for each bucket in the grid is given by intersecting the bit vectors corresponding to the ranges on the ranking attributes. The resulting vectors give us the tuple IDs in each bucket. Moreover, by counting the set bits in a vector, we obtain the cardinality of the corresponding bucket.

In addition to the cardinality, we can obtain the upper-bound and lower-upper scores for tuples in each bucket. As mentioned in Section 4.2, we focus on ranking functions that are monotonic with respect to the ranking attributes. Therefore, given a bucket, the highest (lowest) possible score of the tuples in that bucket is reached when the values of ranking attributes are equal to the right (left) endpoints of the corresponding ranges on these attributes. More formally, given B_{id}=$<range_1^{id_1}, \ldots, range_n^{id_n}>$ $=<[r_1^{id_1}, r_1^{id_1+1}), \ldots, [r_n^{id_n}, r_n^{id_n+1})>$, the upper-bound score for tuples in B_{id} is $upper_{B_{id}}$=$\mathcal{F}(r_1^{id_1+1}, \ldots, r_n^{id_n+1})$ and the lower-bound score is $lower_{B_{id}}$=$\mathcal{F}(r_1^{id_1}, \ldots, r_n^{id_n})$. That is, $\mathcal{F}[t]$=$\mathcal{F}(t.r_1, \ldots, t.r_n) \in [lower_{B_{id}}, upper_{B_{id}}), \forall t \in B_{id}$.

Example 5 (Upper- and Lower-Bounds for Buckets): Continuing with our running example, suppose we rank the 6 tuples of $cluster_2$ in Figure 1 and obtain the top 2 tuples, with the ranking function c+d. Figure 3 illustrates a summary grid for the tuples in $cluster_2$, using the ranking attributes c and d as the partitioning attributes. For the grid, $ranges_c$=$\{[0, 10), [10, 20), [20, 30), [30, 40)\}$, and $ranges_d$=$\{[0, 10), [10, 20), [20, 30), [30, 40)\}$. Thus there are 16 buckets. For instance, B_{13}=$<range_c^3, range_d^1>$ = $<[30, 40), [10, 20)>$. The scores of the tuples in B_{13} are bounded by $[30 + 10, 40 + 20)$. That is, $lower_{B_{13}}$=40 and $upper_{B_{13}}$=60. Similarly we can obtain the bounds for other buckets. ■

Based on the upper-bounds and lower-bounds of the buckets, we can derive a set of candidate buckets that are guaranteed to contain *all* the top k tuples in the grid. Correspondingly the rest of the buckets can be safely pruned as the tuples in these buckets are guaranteed to be ranked lower than top k. By performing union ($|$) of the vectors for the candidate buckets, we can thus retrieve tuples in the candidate buckets to obtain their exact scores. The top k tuples in these candidate buckets form the top k tuples in the grid as well. The intuition is demonstrated in the following example.

Example 6 (Pruning Based on Bounds): The upper-bounds for the buckets in the non-shaded region of Figure 3 are at most 50. Note that the lower-bounds for buckets B_{11} and B_{15} are at least 50 and there are already 2 tuples in these buckets. Therefore we conclude that the tuples in the non-shaded region have no chance to be top 2. On the other hand, we are not able to obtain the same conclusion for any of the buckets in the shaded region, which thus constitute the candidate buckets. Therefore we union the vectors of those non-empty candidate buckets, resulting in $vec_{candidate}$=$vec_{B_{11}}$ $|$ $vec_{B_{13}}$ $|$ $vec_{B_{15}}$ = 0011000010. Thus the candidate tuples are t_3, t_4, and t_9. We retrieve these tuples by random I/O access and identify t_3 and t_9 as the top 2 answers in $cluster_2$. ■

Procedure
/* table: T; ranking attributes: R; ranking function: $\mathcal{F}(R)$; summary grid: $S_{\mathcal{G}(T,R)}$; candidate buckets: $\mathcal{B}'$ */
begin
1: $\mathcal{B}' \leftarrow \phi$
2: **for** each bucket $B_i \in S_{\mathcal{G}(T,R)}$ **do**
3: $total \leftarrow 0$
4: **for** each bucket $B_j \in S_{\mathcal{G}(T,R)}$ **do**
5: **if** $lower_{B_j} \geq upper_{B_i}$ **then**
6: $total \leftarrow total + \#vec_{B_j}$
7: **if** $total < k$ **then**
8: $\mathcal{B}' \leftarrow \mathcal{B}' \cup \{B_i\}$ /* candidate buckets */
9: $vec_{\mathcal{B}'} \leftarrow |_{B_i \in \mathcal{B}'} vec_{B_i}$ /* union of the vectors */
10: $T_{\mathcal{B}'} \leftarrow$ retrieve tuples whose bits are set in $vec_{\mathcal{B}'}$ /* candidate tuples */
11: sort $T_{\mathcal{B}'}$ based on $\mathcal{F}(R)$
12: **return** the top k tuples in $T_{\mathcal{B}'}$
end

Figure 4: Summary-Based *top*-k Algorithm.

The detailed algorithm for ranking is shown in Figure 4. Formally, we prove both the correctness and the optimality of the algorithm, *i.e.*, we prune all and only those buckets that can be pruned.

Property 1: With respect to a relation T, a ranking function $\mathcal{F}(R)$, and k, suppose the top k tuples are T_k. The set of candidate buckets $\mathcal{B}'$ obtained by the algorithm in Figure 4 is both *correct*: $\mathcal{B}'$ contains all the top k tuples, *i.e.*, $T_k \subseteq T_{\mathcal{B}'}$; and *optimal*: $\mathcal{B}'$ is the smallest set of buckets that contain T_k, *i.e.*, $\forall \mathcal{B}''$ *s.t.* $\exists B_i \in \mathcal{B}'$ and $B_i \notin \mathcal{B}''$, there exists an instance of T *s.t.* $T_k \nsubseteq T_{\mathcal{B}''}$. ■

Proof: $\forall t \in T$, we use $B(t)$ to denote the bucket which t falls into in $\mathcal{G}(T, R)$. With respect to a bucket B_i, we use $T^+_{B_i}$ to denote the set of tuples that belong to the buckets whose lower-bound scores are higher than or equal to the upper-bound score of B_i, and $T^-_{B_i}$ to denote the remaining tuples $(T - T^+_{B_i})$, *i.e.*, $T^+_{B_i}$=$\{t | t \in T$ and $lower_{B(t)} \geq upper_{B_i}\}$ and $T^-_{B_i}$=$\{t | t \in T$ and $lower_{B(t)} < upper_{B_i}\}$.

Correctness: We prove the correctness by proving $\forall t \in T_k$, $B(t) \in \mathcal{B}'$, by contradiction. If $B(t) \notin \mathcal{B}'$, then $|T^+_{B(t)}| > k$ and $\forall t' \in T^+_{B(t)}$, $\mathcal{F}(R)[t'] \geq lower_{B(t')} \geq upper_{B(t)} > \mathcal{F}(R)[t]$ [7] (step 4-8 in Figure 4). This means there exists at least k tuples whose scores are higher than that of t, thus $t \notin T_k$, contradicting $t \in T_k$.

Optimality: Consider any $\mathcal{B}''$ *s.t.* $B_i \in \mathcal{B}'$ and $B_i \notin \mathcal{B}''$. Since $B_i \in \mathcal{B}'$, we have $|T^+_{B_i}| < k$ and $|T^-_{B_i}| > |T| - k$. Use $max(T^-_{B_i})$ to denote the maximal score among the tuples in $T^-_{B_i}$. We prove the optimality by proving there exists an instance of T *s.t.* $\exists t$, $t \in T_k$ and $B(t)$=B_i. One such instance is: $\forall t' \in T^-_{B_i}$, $\mathcal{F}(R)[t']$=$lower_{B(t')}$, and $\exists t$, $B(t)$=B_i and $\mathcal{F}(R)[t]$=$\frac{1}{2} \times (max(T^-_{B_i}) + upper_{B_i})$, thus $\mathcal{F}(R)[t] > \mathcal{F}(R)[t'], \forall t' \in T^-_{B_i}$. According to $|T^-_{B_i}| > |T| - k$, $t \in T_k$. Since $B(t)$=B_i and $B_i \notin \mathcal{B}''$, therefore $T_k \nsubseteq T_{\mathcal{B}''}$. ■

[7] Note that $upper_{B(t)} > \mathcal{F}(R)[t]$ since the ranges are defined as left-end closed and right-end open, *cf.* Section 5.2.2.

So far we have assumed that we are given the set of tuples to be ranked. Intersecting the bit vectors for the ranking attributes results in a grid over all the tuples, by using the ranking attributes as the partitioning attributes. However, in our queries, we are required to obtain the top k results in each cluster. In other words, a grid must be constructed for the tuples in each cluster. We realize this easily by intersecting the vectors for the buckets in the aforementioned grid with the bit vector for each cluster, as obtained in Section 5.2.1. An example is shown below.

Example 7 (Integrating Ranking with Clustering): Continue Example 5 and Figure 3. The summary grid in Figure 3 was obtained by assuming that the bitmap indices on ranking attributes c and d are built for the tuples in $cluster_2$ only. While in reality, we can only build bitmap indices for the whole table T, without knowing what will be the clusters for users' dynamic queries. Below is how we can obtain the summary grid in Figure 3 in reality.

According to Example 4, $vec_{cluster_2}$=0111011010, which gives the 6 tuples in $cluster_2$. For T, the bitmap index on c has vectors $real_vec_c^0$, $real_vec_c^1$, $real_vec_c^2$, $real_vec_c^3$. Suppose $real_vec_c^1$ =1000010000. We intersect $real_vec_c^1$ with $vec_{cluster_2}$ to obtain the tuples in $cluster_2$ that fall in $range_c^1$. Thus we have $real_vec_c^1$ & $vec_{cluster_2}$=1000010000 & 0111011010 = 0000010000, which is the vec_c^1 in Figure 3. We can similarly obtain all the vectors, thus obtain the summary grid in Figure 3. ■

5.2.3 Dealing with Boolean Conditions

We have assumed that we cluster the original relation T without considering Boolean conditions. However, the tuples to be clustered are actually the result of Boolean conditions, *i.e.*, $\sigma_{\mathcal{B}}(T)$ [8]. Therefore before constructing the summary grid in Figure 1, the vectors over the clustering attributes must take into consideration the filtering effect of the Boolean conditions. If a tuple does not belong to $\sigma_{\mathcal{B}}(T)$, the corresponding bits in the vectors must be set to 0. Bit vector operations smoothly allow such processing of clustering together with Boolean conditions, as explained below.

Selections: Suppose our query has a set of conjunctive range selection conditions $v_1^1 \leq b_1 \leq v_1^2$, $\ldots$, $v_p^1 \leq b_p \leq v_p^2$. We first obtain a vector $vec_{\mathcal{B}}$, which contains the tuples that satisfy all the selection conditions [9]. Then given the grid on T over the clustering attributes C, $\mathcal{G}(T, C)$, we intersect $vec_{\mathcal{B}}$ with each vector for the individual range on every attribute (the $vec_a^0, vec_a^1, vec_a^2, vec_b^0, vec_b^1, vec_b^2$ in Figure 1), to obtain the grid on $\sigma_{\mathcal{B}}(T)$, before generating virtual data points and applying the weighted clustering algorithm.

There is a vast literature on using bitmap index to answer Boolean queries, which contains the details about how to obtain $vec_{\mathcal{B}}$. Briefly, for each condition $v_i^1 \leq b_i \leq v_i^2$, given a bitmap index over b_i, we can use bitmap operations to obtain a vector $vec_{v_i^1 \leq b_i \leq v_i^2}$ (in simplified form vec_{b_i}) that contains the tuples satisfying the condition. By intersecting the vectors for all the individual conditions, we obtain $vec_{\mathcal{B}}$. The bitmap index may need to be encoded in some way so that a very small number of bitmap operations can allow us to obtain such vec_{b_i}. There are many encoding schemes in the literature(*e.g.*, [5, 29, 28]). For instance, some scheme requires only one bitmap operation for any one-side range selection condition and some requires two for any two-side condition.

Note that even without using bitmap index to handle the selection conditions, we can construct the bit vector $vec_{\mathcal{B}}$ upon $\sigma_{\mathcal{B}}(T)$ that is obtained using any conventional query processing techniques.

[8]Here $\mathcal{B}$ denotes Boolean conditions, not the buckets in Section 5.1- 5.2.2.

[9]More strictly speaking, the corresponding bits for the satisfying tuples in T are all set in the vector.

Join Queries: Under the assumption of snowflake-schema made in Section 4.2, our technique can be easily extended to handle join queries. Such join queries are so-called "star-joins" under star-schema, a special case of snowflake-schema. Consider a simple case with only two tables, where S is the fact table and R is the dimension table, $j1$ is a key of R and $j2$ is the corresponding foreign key in S. Due to the foreign key constraint, there exists one and only one tuple in R joining with each and every tuple $s \in S$. Therefore for a join condition $R.j1$=$S.j2$, virtually all the join results are in S, with some attributes in S and some other in R. Therefore, for each attribute a in the schema of R except $j1$ (since $R.j1$=$S.j2$ and we already have $j2$ in S), we can construct a bitmap index on a for the tuples in S, even though a is not an attribute of S. In general, we can follow this way to construct bitmap index for the tuples in the single fact table, on all relevant attributes in the dimension tables. Thus the Boolean selection conditions involving these attributes can be viewed as being applied on the fact table only. A join query can then be processed like a single table query. More details about such bitmap join index are in [23].

Example 8 (Handling Boolean Conditions): Continue Example 3 and Figure 1. Suppose our query has a condition $10 \leq e \leq 20$. By operations on the bitmap index over e, suppose we obtain the vector vec_e=1100101000, indicating tuples t_1, t_2, t_5, and t_7 satisfy the condition. After intersecting with the vectors on the ranges, we get $(vec_a^1)'$=vec_a^1 & vec_e=1100110011 & 1100101000=1100100000, $(vec_b^2)'$=vec_b^2 & vec_e=1000100101 & 1100101000=1000100000. Thus $(vec_{B_5})'$ = $(vec_a^1)'$ & $(vec_b^2)'$ = 1100100000 & 1000100000 = 1000100000, indicating the new bucket B_5'=$\{t_1, t_5\}$. ■

6. OPTIMIZATION HEURISTICS

In this section, we present the optimization heuristics in summary-based clustering (Heuristic 1 and 2) and ranking (Heuristic 3).

Heuristic 1– Pruning Underpopulated Buckets in Grid Construction for Clustering:
For clustering on high dimensions, there are potentially huge number of buckets in the summary grid even if the number of ranges per attribute is small. However, likely many of the buckets are empty (if there exist clusters at all). During construction of the grid, we get rid of the empty intermediate buckets before the vectors from all the attributes are intersected. More generally, we prune the buckets whose cardinality is under certain threshold, *i.e.*, the underpopulated buckets that likely will result in many empty buckets if they further intersect with the remaining attributes. The pruned buckets do not participate in clustering. After clustering the non-pruned buckets in the grid, we need to use random access to retrieve the tuples belonging to the pruned buckets. (The IDs of these pruned tuples are obtained by bit-negation (*i.e.*, $\sim$) of the union of vectors for all the clusters.) The pruned tuples are then assigned to their closest clusters, whose vectors are modified by setting the bits corresponding to the pruned tuples.

Heuristic 2– Dynamically Selecting Partitioning Ranges:
To construct the summary grid in Section 5, the bitmap index on each partitioning attribute must have a set of bit vectors, one per range of the attribute values. However, we may not know in prior what is the appropriate number of ranges, *i.e.*, the number of vectors to build. On the one hand, too many ranges result in too many buckets in the grid, thus large number of bitmap intersections. On the other hand, insufficient number of buckets due to too few ranges result in poor clustering quality or pruning power, for summary-based clustering or ranking, respectively.

We address this problem by starting with large buckets (*i.e.*, small number of buckets) and splitting buckets dynamically. At the beginning, we start with 2 ranges on each attribute, resulting in 2^n buckets after intersecting the vectors from all the n attributes, among them x_2 buckets are nonempty, thus x_2 bucket vectors. If more ranges are necessary, we split each range into 2 ranges. For each of the x_2 bucket vectors, on each attribute, we intersect the bucket vector with the 2 vectors for the smaller ranges within the original range corresponding to the bucket. We stop splitting an individual bucket if its cardinality is under certain threshold. After this step, totally we obtain x_4 vectors for the smaller nonempty buckets. We stop the whole splitting procedure when the number of nonempty buckets is over another threshold.

Such binary splitting of ranges (vectors) can be easily supported by bitmap encoding scheme such as bit-sliced index (BSI) [24]. A BSI on an attribute a consists of m+1 vectors $vec_0, \ldots, vec_m$, where the ith bit of vec_j is set if the jth bit is set in the binary representation of $t_i.a$. Thus these vectors together form the binary representation of the attribute values in all the tuples. Therefore vec_m and $\sim vec_m$ provide the 2 vectors for the initial 2 ranges on a. Similarly $vec_m \& vec_{m-1}$, $vec_m \& (\sim vec_{m-1})$, $(\sim vec_m) \& vec_{m-1}$, and $(\sim vec_m) \& (\sim vec_{m-1})$ give the 4 vectors when the ranges are split, and so on. We can easily adopt BSI for ranges. The idea is to partition the value domain of an attribute into a sufficiently large number (2^{m+1}) of "minimal" ranges, and number the ranges with $0, \ldots, 2^{m+1}$-1, ordered from the range with the lowest value to the highest. We then use BSI to capture the binary representation of the numbers corresponding to the minimal ranges, thus allow various sizes of the dynamic ranges during splitting.

According to our experiments (discussed in Section 7), empirically a small number of ranges such as 10 is sufficient for 2 or 3 clustering attributes, 5 is sufficient for 4 to 6 attributes, and even only 3 ranges is sufficient for 8 or more attributes. Intuitively, with large number of clustering attributes, two tuples are unlikely to be in the same range on many attributes, therefore large range is sufficient to differentiate the tuples for the clustering. The grid has 3^8 buckets when there are 3 ranges on each of the 8 attributes, which can be sufficient.

Heuristic 3– Incrementally Constructing Grid for Ranking: Directly following Section 5.2.2, we would have to fully construct the summary grid for ranking. However, there is no need for such full grid, since most of the buckets can be pruned even before they are actually constructed. Following this intuition, we construct the summary grid for ranking in lock-step fashion, similar to the NRA *top-k* algorithm [9]. Detailed algorithm is omitted due to space limitation. Instead, we explain the algorithm using an example. For simplicity, suppose there are 2 ranking attributes a and b, each of which has 4 ranges, $ranges_a$ = $\{range_a^0$, $range_a^1$, $range_a^2$, $range_a^3\}$ = $\{[a^0, a^1)$, $[a^1, a^2)$, $[a^2, a^3)$, $[a^3, a^4)\}$ and $ranges_b$ = $\{range_b^0$, $range_b^1$, $range_b^2$, $range_b^3\}$ = $\{[b^0, b^1)$, $[b^1, b^2)$, $[b^2, b^3)$, $[b^3, b^4)\}$.

At step–1, we start by intersecting the first range from each ranking attribute, *i.e.*, $range_a^3$ and $range_b^3$, resulting in the single bucket with the highest upper-bound score in the whole grid, *i.e.*, $B_{15} = < range_a^3, range_b^3 >$. Then at each following step–i, we use the next range on every attribute (in our example $range_a^{4-i}$ and $range_b^{4-i}$) and intersect them with previous ranges. The seen ranges classify the buckets in the full grid into three types: (1) *completely seen buckets* or *csb*, whose corresponding ranges are seen on every attribute in previous steps; (2) *partially seen buckets* or *psb*, whose ranges are seen on some attributes; and (3) *unseen buckets* or *usb*, whose ranges are unseen on every attribute. The upper-bound and lower-bound scores for *csb* are computed from the corresponding ranges. For one *psb* B, on attribute a, if the corresponding range is $range_a^j$, where $j < 4 - i$ (*i.e.*, $range_a^j$ is unseen yet), we use $[a^{4-i-1}, a^{4-i})$ as the range of B on a, otherwise we use the corresponding seen range if $j \geq 4 - i$. We thus obtain the bounds for *psb*. Similarly we obtain the bounds for *usb*, as it is a special case of *psb*. At some step, if there is a subset of *csb* containing at least k tuples in total such that their lower-bounds are higher than or equal to the upper-bounds of all the *psb* and *usb*, then the top k tuples in the *csb* are the top k answers and our algorithm terminates.

parameter	meaning	values
s	# tuples	80K,400K,800K, 4M, 8M
t	# clusters	2,4,6,8,10,20,50,100
c	# clustering attributes	$[2, 8]$
p	# ranges per clustering attribute	5, 10, 20, 30, 40
k	retrieval size per cluster	1, 5, 10, 50, 100
r	# ranking attributes	$[2, 5]$
p'	# ranges per ranking attribute	10, 20

Table 1: Configuration Parameters.

7. EXPERIMENTS

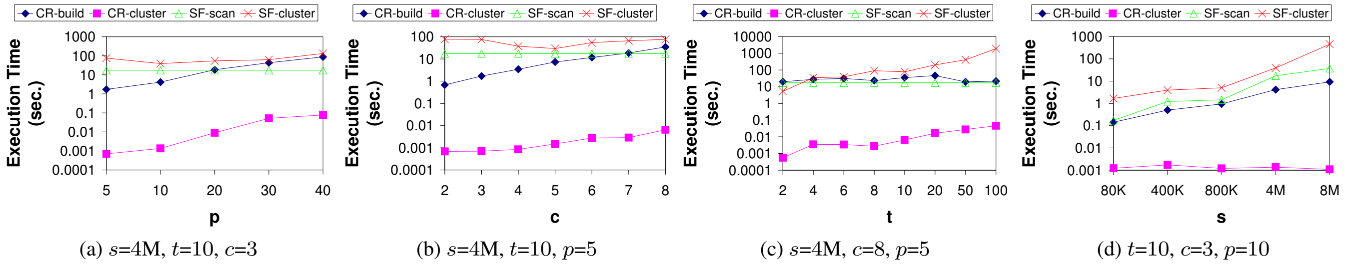
The framework and algorithms are implemented in C++. Moreover, the bitmap index implementation is based on [26], which builds multiple bitmap indices at different domain resolutions and compresses them using the WAH compression method [28]. The weighted K-means is built upon an publicly available implementation of K-means algorithm from [10]. For the straightforward approach of materialize-cluster-rank, we apply this K-means implementation on real tuples instead of the virtual tuples from summary, and use an implementation of external merge-sort for ranking.

To verify its effectiveness, we conducted experiments to compare the proposed framework (denoted as *ClusterRank*) with the straightforward approach (denoted as *StraightFwd*), on both efficiency and quality. Moreover, we investigated how they are affected by important factors under various configurations. The detailed experimental results are presented below. Section 7.1 describes the settings of experiments. Regarding *efficiency*, the experimental results show that *ClusterRank* is orders of magnitude more efficient than *StraightFwd* (Section 7.2). Regarding *quality*, the results indicate that clustering based on the summary gird achieves close to the same quality of results as clustering on the full data does (Section 7.3).

7.1 Experimental Settings

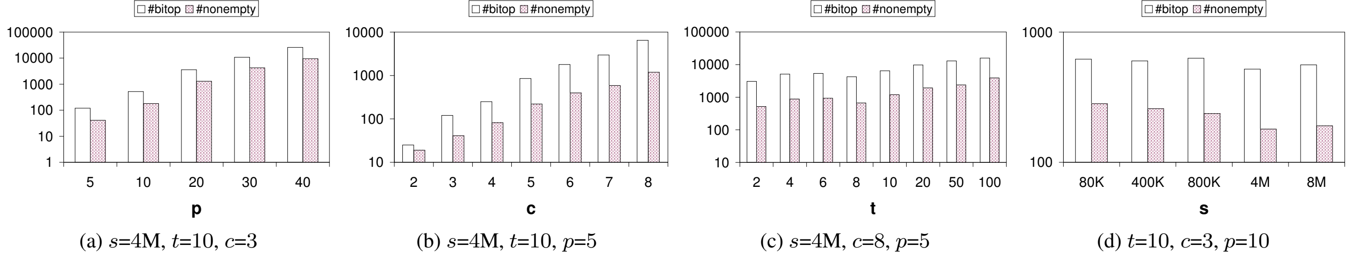
Our experiments were conducted over a synthetic table with a set of 4-byte integer clustering attributes, a set of 4-byte floating number ranking attributes, and other attributes to pad the clustering and ranking attributes to form 100-byte per tuple. The tuple values of these two sets of attributes are independently created. The values of the ranking attributes are independently generated by various distributions, including uniform, Gaussian, and cosine distributions. The values of the clustering attributes are produced by a data generator for clustering algorithms from [10]. The generator creates values based on underlying data models, one model per cluster. A model specifies, for the corresponding cluster, the mean and standard deviation of each attribute individually. The values on an attribute are generated by following the Gaussian distribution with the specified mean and standard deviation.

Each query used in our experiments clusters the tuples by all the clustering attributes and uses the sum of the ranking attributes as the ranking function. Note that we do not experiment with Boolean



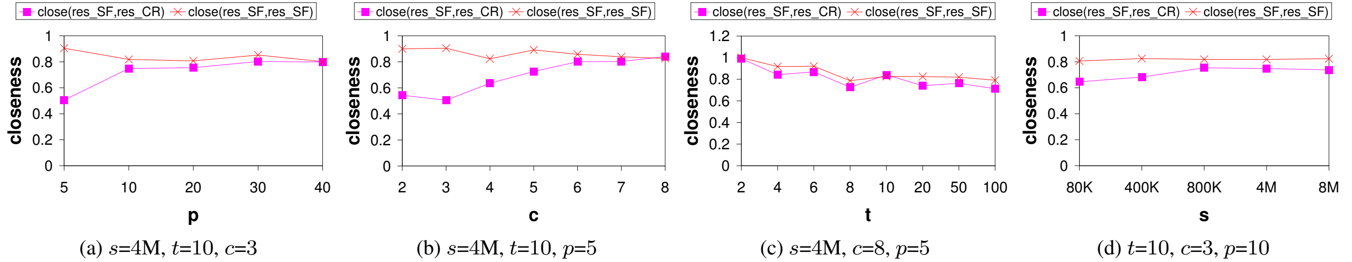
(a) s=4M, t=10, c=3 (b) s=4M, t=10, p=5 (c) s=4M, c=8, p=5 (d) t=10, c=3, p=10

Figure 5: Clustering Efficiency.



(a) s=4M, t=10, c=3 (b) s=4M, t=10, p=5 (c) s=4M, c=8, p=5 (d) t=10, c=3, p=10

Figure 6: Number of Bitmap Operations and Nonempty Buckets.



(a) s=4M, t=10, c=3 (b) s=4M, t=10, p=5 (c) s=4M, c=8, p=5 (d) t=10, c=3, p=10

Figure 7: Clustering Quality.

selection and join conditions. The synthetic table can be viewed as the results after such conditions are applied. To obtain the results, the approach of using bitmap index has been well-studied and is shown to be very efficient for range selections and star-joins [22, 23, 24, 5, 29]. In Section 5.2.3 we have discussed how to integrate with such techniques. Therefore, to focus on the performance study of the new clustering and ranking methods proposed, we do not mix with the performance measurements on Boolean conditions, whose results are well-known in the literature.

The experiments were run on a PC with 2.8GHz Intel Xeon SMP (dual hyperthreaded CPUs each with 1MB cache), 2GB RAM, and a RAID5 array of 3 146GB SCSI disks, running Linux 2.6.15.

7.2 Efficiency

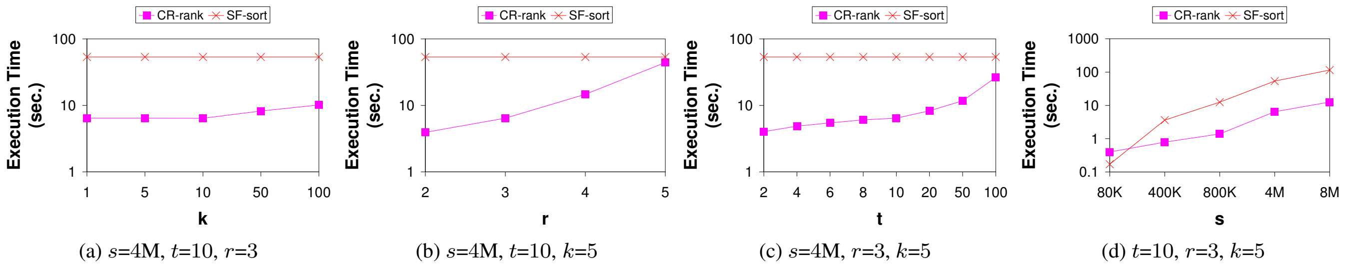
We evaluated the performances of *ClusterRank* and *StraightFwd* and studied how they are affected by several important configuration parameters, which are summarized in Table 1.

The Efficiency of Clustering:
To evaluate the performance of clustering, we conducted experiments under groups of configurations by the value combinations of the four relevant parameters, s, t, c, and p. In each group of experiments, we varied the value of one parameter and fixed the values of the remaining three. We then run *ClusterRank* and *StraightFwd*, and studied how their performances are affected as the value of the varying parameter changed. The results on wall-clock execution time under four sample groups of experiments are shown in Figure 5. In the figure, for *ClusterRank*, we use *CR-build* to represent the time for building summary grid and *CR-cluster* for clustering using the summary. For *StraightFwd*, we use *SF-scan* to denote the time for scanning the table and *SF-cluster* for directly clustering the original tuples instead of using the summary.

Overall, the values of these four time measurements are in general in the order of *CR-cluster*<*CR-build*<*SF-scan*<*SF-cluster*. Due to the small number of virtual tuples in the summary grid, *CR-cluster* is several orders of magnitude smaller than others, therefore is almost negligible. While on the other hand, *SF-cluster* is orders of magnitude larger than others. *SF-scan* is normally larger than *CR-build*, with the differences often being an order of magnitude. In summary, Figure 5 shows that with respect to the efficiency in clustering, our approach (*CR-build* + *CR-cluster*) is much more efficient than the straightforward one (*SF-scan* + *SF-cluster*).

To understand how the parameters affect the performance, below we further analyze the individual graphs in Figure 5.

(a) s**=4M,** t**=10,** c**=3**: Varying the number of ranges per attribute in constructing the grid, 4 million tuples are partitioned into 10 clusters by 3 clustering attributes. As the number of ranges (p) increases, the less efficient grid construction (*CR-build*) is due to more bitmap operations for intersecting the ranges. Moreover, as p increases, there are also more nonempty buckets (thus more virtual tuples) in the summary grid when it is constructed, therefore *CR-cluster* takes longer. This is further verified by Figure 6, which shows the number of bitmap operations and the number of nonempty buckets in the grid. Figure 6(a)-(d) illustrate the results under four groups of configurations, individually corresponding to the configurations in Figure 5(a)-(d). Although the cost of *ClusterRank* in-



(a) s=4M, t=10, r=3 (b) s=4M, t=10, k=5 (c) s=4M, r=3, k=5 (d) t=10, r=3, k=5

Figure 8: Ranking Efficiency.

creases as p increases, usually a small value of p such as 10 or 5 is sufficient for good quality of clustering results.

(b) s=4**M,** t=10**,** p=5: In this configuration c is changing and other parameters are fixed. We can see that the effect of c is similar to that of p, as its increasing results in more bitmap intersections and more nonempty buckets (*cf.* Figure 6(b)), thus more expensive *CR-build* and *CR-cluster*. *ClusterRank* enjoys clear advantages over *StraightFwd* until the number of clustering attributes goes beyond 8, which we believe is sufficiently large in our target applications.

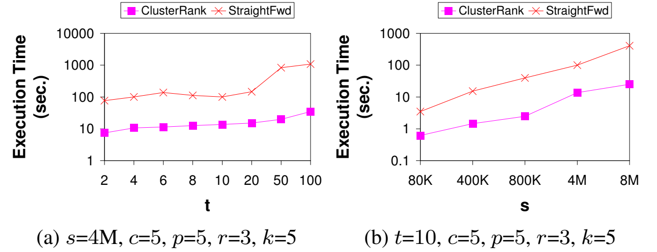
(c) s=4**M,** c=8**,** p=5: As expected, the more clusters to produce, the less efficient the clustering is, thus resulting in longer execution times for both *CR-cluster* and *SF-cluster*.

(d) t=10**,** c=3**,** p=10: As expected, increasing the number of tuples increases the cost of everything.

The Efficiency of Ranking:
We conducted experiments under configurations of the four relevant parameters, s, t, r, and k. Similar to the experiments in clustering efficiency, in each group of experiments, we changed the value of one parameter and fixed the remaining ones. Note that the number of ranges per ranking attribute (p') with value 20 works quite well in general in all the configurations. Therefore we do not present the results with respect to various p' values. The wall-clock execution time under four sample groups of experiments is shown in Figure 8, where we use *CR-rank* to denote the time for grid-based ranking in *ClusterRank* and *SF-sort* for the sorting in *StraightFwd*.

Overall, *SF-sort* is one order of magnitude more expensive than *CR-rank*. We further analyze the individual graphs. Figure 8(a) shows that *CR-rank* only increases slowly as k increases, thus *ClusterRank* is effective for sufficiently large retrieval size within each cluster; Figure 8(b) shows that *CR-rank* increases as the number of ranking attributes (r) increases, and becomes close to *SF-sort* when r=5. However, as commonly acknowledged in the literature of ranking queries (*e.g.*, [9, 3, 8, 6, 1, 15, 7, 20, 14]), in many cases a very small number of ranking attributes suffice. We believe this is especially true in our motivating applications, where users are not expected to articulate too complicated ranking criteria involving more than 5 attributes; Figure 8(c) indicates that *CR-rank* increases as the number of clusters (t) increases. This is because *ClusterRank* constructs a grid for each cluster and performs ranking within each grid. Although *SF-sort* is not affected by t, it may be smaller than *CR-rank* only when there are a very large number of clusters. We argue the number of clusters t is small in our target applications, because clustering is used to organize large query results for users and such large t is not helpful and thus unnecessary. Finally, Figure 8(d) shows that both *CR-rank* and *SF-sort* increase as the number of tuples increases, as expected.

The Overall Efficiency:
We compared *StraightFwd* and *ClusterRank* with the execution time for clustering and ranking combined. Consider both Figure 5 and 8, we see that *SF-sort* in general is close to *SF-cluster*. Thus these two



(a) s=4M, c=5, p=5, r=3, k=5 (b) t=10, c=5, p=5, r=3, k=5

Figure 9: Total Execution Time.

costs together dominate the execution time of *StraightFwd*, which is orders of magnitude more than the time of *ClusterRank*. In Figure 9 we show such comparisons under two sample configurations. In Figure 9(a) we vary the number of clusters and fixed the values of others, while in Figure 9(b) we vary the number of tuples.

7.3 Quality

We compared res_{CR}, the clustering results from the weighted K-means on the summary grid (*ClusterRank*), with res_{SF}, the results from the conventional K-means on the original tuples (*StraightFwd*). We measured how close res_{CR} is to the ground truth res_{SF}, *i.e.*, $close(res_{SF}, res_{CR})$. This metric is defined below.

Suppose two methods generate two different sets of clusters res=$\{c_1, \ldots, c_t\}$ and res'=$\{c'_1, \ldots, c'_t\}$, respectively, where each c_i and c'_j is a set of tuples. The *closeness* of res' to the ground-truth res is

$$close(res, res') = \frac{\sum_i (|c_i| \times max_j(sim(c_i, c'_j)))}{\sum_i |c_i|},$$

where

$$sim(c_i, c'_j) = 2\frac{|c_i \cap c'_j|}{|c_i| + |c'_j|}.$$

This metric is asymmetric. Since $close(res, res')$ measures how well the clusters in res are captured by the clusters in res', it should be used when res instead of res' is the ground-truth. Its value range is $[0, 1]$, as 1 indicates identical results and 0 indicates totally different results. The metric sim has been used in comparing clustering results, *e.g.*, in [18] and [12]. It is equivalent to the *F-measure* for *precision/recall* in IR literature.

K-means algorithm is known to be unstable and its behavior depends on the initial centroids chosen [16]. Even running K-means twice on the same data may not give us very high closeness between the two results. Therefore instead of interpreting the value of $close(res_{SF}, res_{CR})$ directly, we compare it with $close(res_{SF}, res_{SF})$, which is the average closeness among the results from multiple runs of *StraightFwd*. If $close(res_{SF}, res_{CR})$ is close to $close(res_{SF}, res_{SF})$, we are confident that the quality of the results from *ClusterRank* is comparable to that from *StraightFwd*.

Figure 7(a)-(d) show $close(res_{SF}, res_{CR})$ and $close(res_{SF}, res_{SF})$ under four groups of configurations, corresponding to the configurations in Figure 5(a)-(d) and Figure 6(a)-(d). The figures show that the quality of clustering results from *ClusterRank* is of-

ten close to the quality from *StraightFwd*. The quality increases as the number of ranges per clustering attribute (p) increases (Figure 7(a)), because the summary grid becomes more and more fine-grained. However, we observe that a relatively small p such as 5 and 10 usually is sufficient. As Figure 7(b) shows, for the same p, the more attributes, the higher quality. This is simply because it is easier to partition data when they have more dimensions to compare with each other. Therefore with 8 clustering attributes, p=5 is sufficient under various number of clusters requested (Figure 7(c)), and with 3 clustering attributes, p=10 is pretty good (Figure 7(d)). Moreover, Figure 7(c) verifies that it is more difficult to perform clustering when more clusters are quested.

8. CONCLUSION

This paper proposes to generalize **group-by** to enable fuzzy grouping (clustering in particular) of database query results, and to integrate grouping with ranking and further with Boolean filtering, for supporting structured data retrieval applications. We define a new type of *ClusterRank* queries for this purpose. We design a summary-based framework to meet the challenges in supporting such integration. We realize the framework by utilizing bitmap index to construct the summary on-the-fly, and to efficiently integrate Boolean filtering, clustering, and ranking altogether. Experimental study with our implementation shows that the framework achieves orders of magnitude better efficiency than the straightforward approach available in current databases, and at the same time it maintains high clustering quality.

To the best of our knowledge, this work is the first to propose such generalization of fuzzy grouping and integration with ranking within RDBMSs. We believe, as an important first step in this direction, the concept and framework in this paper will inspire us to conduct future work on many interesting topics. For instance, how will the approach perform differently under various datasets and query workloads? How to support categorical attributes in clustering and even ranking? How to efficiently process more interesting semantics other than global clustering/local ranking? How to design a cost model and incorporate the framework into relational query optimizer? How to design a useful user interface upon *ClusterRank* to fully support structured data retrieval? We thus plan to investigate these issues.

Acknowledgements: We thank Rishi Rakesh Sinha for providing the source code of bitmap index.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. In *CIDR*, 2003.

[2] P. Berkhin. Survey of clustering data mining techniques. Technical report, Accrue Software, San Jose, CA, 2002.

[3] M. J. Carey and D. Kossmann. On saying "enough already!" in SQL. In *SIGMOD*, pages 219–230, 1997.

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

[5] C. Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *SIGMOD*, 1999.

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

[7] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR technologies: What is the sound of one hand clapping? In *CIDR*, pages 1–12, 2005.

[8] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top n queries. In *VLDB*, 1999.

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

[10] F. Farnstrom, J. Lewis, and C. Elkan. Scalability for clustering algorithms revisited. 2(1):51–57, August 2000.

[11] V. Ganti, J. Gehrke, and R. Ramakrishnan. CACTUS - clustering categorical data using summaries. In *KDD*, pages 73–83, 1999.

[12] M. Gavrilov, D. Anguelov, P. Indyk, and R. Motwani. Mining the stock market (extended abstract): which measure is best? In *SIGKDD*, pages 487–496, 2000.

[13] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, New York, 2000.

[14] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. *SIGMOD*, 2001.

[15] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765, 2003.

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

[17] K. Kerdprasop, N. Kerdprasop, and P. Sattayatham. Weighted k-means for density-biased clustering. In *DaWaK*, pages 488–497, 2005.

[18] B. Larsen and C. Aone. Fast and effective text mining using linear-time document clustering. In *SIGKDD*, pages 16–22, 1999.

[19] A. Leuski and J. Allan. Improving interactive retrieval by combining ranked lists and clustering. In *RIAO*, pages 665–681, 2000.

[20] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. RankSQL: Query algebra and optimization for relational top-k queries. In *SIGMOD*, pages 131–142, 2005.

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

[22] P. E. O'Neil. Model 204 architecture and performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 40–59, 1987.

[23] P. E. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, 1995.

[24] P. E. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD*, pages 38–49, 1997.

[25] G. Sheikholeslami, S. Chatterjee, and A. Zhang. WaveCluster: A multi-resolution clustering approach for very large spatial databases. In *VLDB*, pages 428–439, 1998.

[26] R. R. Sinha, S. Mitra, and M. Winslett. Bitmap indexes for large scientific data sets: A case study. In *IPDPS*, 2006.

[27] W. Wang, J. Yang, and R. R. Muntz. STING: A statistical information grid approach to spatial data mining. In *VLDB*, pages 186–195, 1997.

[28] K. Wu, E. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM TODS*, 31(1):1–38, 2006.

[29] M.-C. Wu and A. P. Buchmann. Encoded bitmap indexing for data warehouses. In *ICDE*, pages 220–230, 1998.

[30] F. Zemke, K. Kulkarni, A. Witkowski, and B. Lyle. Introduction to OLAP function. *Change proposal. ANS-NCTS H2-99-14 (April)*, 1999.

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