Efficient Computation of Diverse Query Results*

Erik Vee, Jayavel Shanmugasundaram, Sihem Amer-Yahia

Yahoo! Research
Sunnyvale, CA, USA
{erikvee,jaishan,sihem}@yahoo-inc.com

Abstract

We study the problem of efficiently computing diverse query results in online shopping applications, where users specify queries through a form interface that allows a mix of structured and content-based selection conditions. Intuitively, the goal of diverse query answering is to return a representative *set of top-k answers from all the tuples that satisfy the user selection condition. For example, if a user is searching for cars and we can only display five results, we wish to return cars from five different models, as opposed to returning cars from only one or two models. A key contribution of this paper is to formally define the notion of diversity, and to show that existing score based techniques commonly used in web applications are not sufficient to guarantee diversity. Another contribution of this paper is to develop novel and efficient query processing techniques that guarantee diversity. Our experimental results using Yahoo! Autos data show that our proposed techniques are scalable and efficient.*

1 Introduction

Online shopping is increasing in popularity due to the large inventory of listings available on the Web. Users can issue a search query through a combination of fielded forms and keywords, and only the most relevant search results are shown due to the limited "real-estate" on a Web page. An important but lesser-known concern in such applications is the ability to return a *diverse* set of results which best reflects the inventory of available listings. As an illustration, consider a user searching for used 2009 MotoPed scooters. If we only have space to show five results, we would rather show five different MotoPed models (e.g., MotoPed Zoom, MotoPed Putt, MotoPed Bang, MotoPed Zip and MotoPed Vroom) instead of showing cars from just one or two models. Similarly, if the user searches for 2009 MotoPed Zoom scooters, we would rather show 2009 MotoPed Zoom scooters in different colors rather than simply showing scooters of the same color. Other applications such as online auction sites and electronic stores also have similar requirements (e.g., showing diverse auction listings, cameras, etc.).

While there are several existing solutions to this problem, they are either inefficient or do not work in all situations. For instance, the simplest solution is to obtain all the query results and then pick a diverse subset from these results. A more scalable variant of this method is commonly used in web search engines: in order to show k results to the user, first retrieve $c \times k$ results (for some $c > 1$) and then pick a diverse subset from these

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*An earlier version of this work appeared in *Proceedings of the 24th International Conference on Data Engineering*, ICDE 2008, April 7-12, 2008, Cancun, Mexico, pp. 228-236.

results [3, 11, 12]. However, while this method works well in web search where there are few duplicate or near-duplicate documents, it does not work as well for structured listings since there are many more duplicates. For instance, it is not uncommon to have hundreds of cars of a given model in a regional dealership, or thousands of cameras of a given model in a large online store. Thus, c would have to be of the order of 1000s or 10000s, which is clearly inefficient and furthermore, does not guarantee diverse results.

Another commonly used method is to issue multiple queries to obtain diverse results. For instance, if a user searches for purple MotoPed scooters, this method would issue a query for purple MotoPed Zooms, another for purple MotoPed Putts, and so on. While this method guarantees diverse results, it is inefficient for two reasons: it issues multiple queries, which hurts performance, and many of these queries may return empty results (e.g., if there are no purple MotoPed Zooms)

A final method that is sometimes used is to retrieve only a sample of the query results (e.g., using techniques proposed in [9]) and then pick a diverse subset from the sample. However, this method often misses rare but important listings that are missed in the sample.

To address the above limitations, we initiate a formal study of the diversity problem in search of methods that are scalable, efficient and guaranteed to produce diverse results. Towards this goal, we first present a formal definition of diversity, including both unscored and scored variants, that can be used to evaluate the correctness of various methods. We then explore whether we can use "off-the-shelf" technology to implement diversity efficiently and correctly. Specifically, we explore whether we can use optimized Information Retrieval (IR) engines with score-based pruning to implement diversity, by viewing diversity as a form of score. Unfortunately, it turns out that the answer is no — we prove that no possible assignment of static or query-dependent scores to items can be used to implement diversity in an off-the-shelf IR engine (although there is an open conjecture as to whether we can implement diversity using a combination of static and query-dependent scores).

We thus devise evaluation algorithms that implement diversity *inside the database/IR engine*. Our algorithms use an inverted list index that contains item ids encoded using Dewey identifiers [6]. The Dewey encoding captures the notion of *distinct values* from which we need a representative subset in the final query result. We first develop a one-pass algorithm that produces k diverse answers with a single scan over the inverted lists. The key idea of our algorithm is to explore a bounded number of answers within the same distinct value and use B+-trees to skip over similar answers. Although this algorithm is optimal when we are allowed only a single pass over the data, it can be improved when we are allowed to make a small number of probes into the data. We present an improved algorithm that is allowed to probe the set of answers within the same distinct value iteratively. The algorithm uses just a small number of probes — at most $2k$. Our algorithms are provably correct, they can support both unscored and score versions of diversity, and they can also support query relaxation Our experiments show that they are scalable and efficient. In summary, the main contributions of this paper are

- A formal definition of diversity and a proof that "off-the-shelf" IR engines cannot be used to implement diversity (Section 2)
- Efficient one-pass and probing algorithms for implementing diversity (Section 3)
- Experimental evaluation using Yahoo! Autos data (Section 4)

2 Diversity Definition and Impossibility Results

We formally define the notion of diversity and present some impossibility results for providing diversity using off-the-shelf IR systems.

Data and Query Model. We assume that the queried items are stored as tuples in a relation R. A query Q on a relation R is defined as a conjunction or disjunction of two kinds of predicates: *scalar predicates* of the form $\texttt{att} = \texttt{value}$ and *keyword predicates* of the form $\texttt{att} \ni \texttt{keywords}$ where $\texttt{att}$ is an attribute of R and $\ni$ stands

for keyword containment. Given a relation R and a query Q, we use the notation $\text{RES}(R, Q)$ to denote the set of tuples in R that satisfy Q.

In many online applications, it is also often useful to allow tuples to have scores. One natural case is in the presence of keyword search queries, e.g., using scoring techniques such as TF-IDF [10]. Another case is in the context of "soft matches," where we give a weight to tuples so long as they satisfy *some* of the predicates in a given query (e.g., see [2]). We use the notation $score(t, Q)$ to denote the score of a tuple t that is produced as a result of evaluating a query Q.

Id	Make	Model	Color	Year	Description
1	MotoPed	Zoom	Green	2009	Low miles
2	MotoPed	Zoom	Blue	2009	Low miles
3	MotoPed	Zoom	Red	2009	Low miles
4	MotoPed	Zoom	Black	2009	Low miles
5	MotoPed	Zoom	Black	2008	Low price
6	MotoPed	Putt	Blue	2009	Best price
7	MotoPed	Putt	Red	2008	Good miles
8	MotoPed	Bang	Green	2009	Rare
9	MotoPed	Bang	Green	2008	Good miles
10	MotoPed	Zip	Red	2009	Fun car
11	MotoPed	Zip	Orange	2008	Good miles
12	Skoot	Hawk	Tan	2009	Low miles
13	Skoot	Raptor	Black	2009	Low miles
14	Skoot	Falcon	Blue	2009	Low miles
15	Skoot	Eagle	Blue	2009	Low miles

(a)

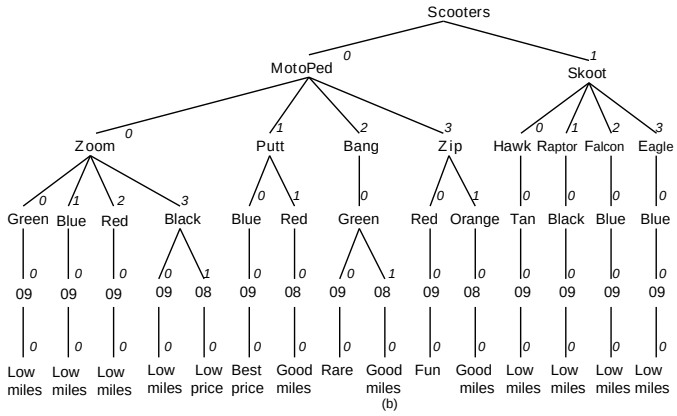

Figure 1: Example Database and Dewey Tree Representation

Diversity Definition. Consider the database of Figure 1(a). If the user issues a query for all cars and we have room for 3 results, then clearly, we should show at least one Skoot scooter and one MotoPed. If the user issues a query `Make = Skoot`, then we would show 3 different models of Skoot scooters. In general, there is a priority ordering of attributes: Make is more important than model, which is more important than say, color. This ordering is domain specific, and can be modified to suit the context. We define it below.

Definition 1: Diversity Ordering. A diversity ordering of a relation R with attributes A, denoted by $\prec_R$, is a total ordering of the attributes in A.

In our example, `Make` $\prec$ `Model` $\prec$ `Color` $\prec$ `Year` $\prec$ `Description` $\prec$ `Id`. (We ignore the suffix in $\prec_R$ when it is clear from the context.)

Given a diversity ordering, we can define a similarity measure between pairs of items, denoted $\text{SIM}(x, y)$, with the goal of finding a result set S whose items are least similar to each other (and hence most diverse); i.e., we wish to find a result set that minimizes $\sum_{x,y \in S} \text{SIM}(x, y)$.

With an eye toward our ultimate goal, let us take a very simple similarity function: $\text{SIM}(x, y) = 1$ if x and y agree on the highest priority attribute, and 0 otherwise. It is not hard to check that by using this similarity measure, minimizing the all-pairs sum of similarities in Skoots and MotoPeds (within one), so long as there are enough Skoots and MotoPeds to display.

However, we need to diversify not just on the top level, but on lower levels as well. If ρ is a prefix and S is a set, then denote $S_\rho = \{x \in S : \ \rho \text{ is a prefix of } x\ \}$. Then, if ρ is a prefix of length ℓ, define $\text{SIM}_\rho(x, y) = 1$ if x, y agree on their $(\ell + 1)$st attribute, and 0 otherwise. Again thinking of our example database, notice that if ρ = `MotoPed Zoom`, then minimizing $\sum_{x,y \in S_\rho} \text{SIM}_\rho(x, y)$ guarantees that we will not display two `Black` Zooms before displaying the `Green`, `Blue`, and `Red` ones, so long as they all satisfy a given query.

We are now almost ready for our main definition. Let $\mathcal{R}_k(R, Q)$ denote the set of all subsets of $\text{RES}(R, Q)$ of size k. For convenience, we will suppress the R, Q when it is clear from context.

Definition 2: Diversity. Given a relation R, a diversity ordering $\prec_R$, and a query Q, let $\mathcal{R}_k$ be defined as above. Let ρ be a prefix consistent with $\prec_R$. We say set $S \in \mathcal{R}_k$ is *diverse with respect to* ρ if $\sum_{x,y \in S_\rho} \text{SIM}_\rho(x, y)$ is minimized, over all sets $T \in \mathcal{R}_k$ such that $|T_\rho| = |S_\rho|$.

We say set $S \in \mathcal{R}_k$ is a *diverse result set (for* $\prec_R$*)* if S is diverse with respect to every prefix (for $\prec_R$).

Our definition for scored diversity is analogous. We define $\mathcal{R}_k^{score}$ to be the collection of all result sets of size k that have the largest score possible. Then scored diversity is defined as in Definition 2, with $\mathcal{R}_k$ replaced by $\mathcal{R}_k^{score}$. It can be shown that a diverse set of size k always exists, for both scored and unscored diversity.

Impossibility Results. We define Inverted-List Based IR Systems as follows: each unique attribute value/keyword contains the list of items that contain that attribute value/keyword. Each item in a list also has a score, which can either be a global score (e.g., PageRank) or a value/keyword -dependent score (e.g., TF-IDF). The items in each list are ordered by their score (typically done so that top-k queries can be handled efficiently). Given a query Q, we find the lists corresponding to the attribute values/keywords in Q, and aggregate the lists to find a set of k top-scored results. The score of an item that appears in multiple lists is aggregated from the per-list scores using a monotone aggregation function (efficient algorithms such as the Threshold Algorithm [5] require this). We have the following impossibility result.

Theorem 3: There is a database such that no Inverted-List Based IR System always produces an unscored diverse result set, even if we only consider non-null queries.

In fact, Figure 1(a) is such a database. Though strong, this result does not rule out every conceivable IR system. If we allow f to consider both static and value/keyword -dependent scores, then for every database, there is a set of scores and a monotonic function f such that the top-k list for every query is a diverse result set. However, the construction produces an f that essentially acts as a look-up table into the database, clearly an inefficient and infeasible solution. (It takes $O(n^2)$ space, where n is the number of items, to just write down this f.) We leave open the question of whether there is a "reasonable" aggregation function f that produces diverse result sets.

3 Algorithms

Data Structures and Processing. Given a diversity ordering, items can be arranged in a tree (which we refer to as a *Dewey tree*, as illustrated in Figure 1(b). Each item is then assigned a unique id, reminiscent of the Dewey encoding as done in XML query processing [6]. Each leaf value is obtained by traversing the tree top down and assigning a distinct integer to siblings. Since we only need to distinguish between siblings in the tree, we can re-initialize the numbering to 0 at each level. So, for example, the `MotoPed Bang Green 08 'Good miles'` has Dewey id 0.2.0.1.0.

Our basic functionality for processing lists follows standard IR systems. We use a WAND-like algorithm, which supports the calls `next` and `prev`. Since items are identified with and sorted by their Dewey ids, we can think of `next` as moving left-to-right through the leaves of the Dewey tree, returning the first item satisfying a given query. It also supports skipping to a new Dewey id; we will think of this as skipping to the beginning of a new subtree/branch. So for example, we could skip to find the first matching result in the `MotoPed Zip` subtree. Note that if no such matches exist in that subtree, `next` will continue on through the leaves until a result was found. (So, e.g., a result from the `Skoot Falcon` branch might be returned.) The method `prev` works in the same way, but moving *right-to-left*. For scored results, we modify `next` and `prev` to return the first matching item with some minimum score.

One-Pass Unscored Algorithm. In the one-pass algorithm, we make calls only to `next`, potentially skipping items, but never going back. Conceptually, this means that we encounter results from our query in left-to-right order of our tree nodes, as pictured in Figure 1(b). We maintain a *tentative result set*. This tentative result set is a valid result set, and further, is the most diverse result set possible, given the items we have encountered so far.

Consider a query Q requesting k items. Our tentative result set starts as the first k items matching our query. We then skip to the next dewey id that could possibly improve the diversity of our tentative result set. We repeat this until we have no more items matching the query.

The key steps of this tentative set maintenance are (1) deciding how far to skip, and (2) deciding which element to remove from the tentative result set to make room for the latest item found. We first describe (2). Let the *branch weight* of a branch in a tree to be the total number of leaves in the subtree rooted at (either) endpoint of the branch.

The algorithm for removal works as follows: Imagine the $k + 1$ items in the tentative result set together with the latest found item, arranged in a tree. Working from the root, always follow a branch with the highest branch weight (breaking ties by, say, choosing the rightmost among highest-weight branches). Eventually, we reach a leaf of this tree; remove the corresponding item from our set. Notice that the most diverse set will have "balanced" branches. The removal algorithm always seeks to remove a leaf from the most unbalanced branch.

The intuition behind the algorithm for (1) is to skip over any item that would result in immediate removal from the tentative result set We again imagine the tentative result set arranged in a tree. Working from the root, always examine the rightmost branch, i.e., the branch leading to the leaf corresponding to the most recently added item. If this branch is ever the highest-weight branch, then stop; skip to the first dewey id past this branch. On the other hand, if this branch ever has weight more than one less than the heaviest-weight branch, then stop; we cannot skip at all. Otherwise, when the branch has weight exactly one less than the heaviest-weight branch, continue down toward the leaf.

The key savings in this algorithm come from knowing when it is acceptable to skip ahead (i.e. jumping to a new branch in the Dewey tree). In the worst case, at most $k \ln^d(3k)$ calls to `next` are made to compute a top-k list with a Dewey tree of depth d. When the list of matching items is large, this can be a significant saving over the naive algorithm.

One-pass Scored Algorithm. The main difference with the unscored algorithm lies is what parts of the tree we can skip over. Each time we skip in the scored version, we can only skip to the *smaller* of the original skip Dewey id and the next item whose score is greater than or equal to the minimum score among items in our current tentative result set. Likewise, we can only remove items from the tentative result set whose scores are minimum. In all other respects, the algorithm is the same.

Unscored Probing Algorithm. The probing algorithm simply probes the Dewey tree, searching the branch that will be most beneficial in generating a diverse result set. Imagine the set of items arranged in a tree, as in Figure 1(b). The first probe searches the first branch of the tree, sweeping left-to-right. Note that since we are using a WAND-like implementation, this is accomplished by calling `next` on the all-zeros dewey id. We next wish to probe a new branch of the tree, in order to get the maximum diversity. However, rather than calling `next` on a new branch, we instead sweep *right-to-left* by calling `prev` on the all-infinity dewey id. (Thus, we start at the rightmost leaf and move left until finding a result.)

We continue in this manner, alternating between calling `next` on the branch *after* the branch of the most recently found item from a `next` call, and calling `prev` on the branch *before* the branch of the most recently found item from a `prev` call. If we continue in this way for k calls without returning two items from the same branch, we are done. However, if one of our calls returns an item from a previously used branch, then we have more work to do.

In this case, suppose we have found k' items so far. We divide $k - k'$ as evenly as possible among the

branches with found items. We then recurse on each subtree. For example, suppose $k = 6$ and we have found one MotoPed and one Skoot; searching more will repeat a branch, so we recurse. Here, we would assign 2 more items to the MotoPed branch and 2 more items to the Skoot branch. We continue our algorithm on each subtree. In our example, we would first process the subtree rooted at the MotoPed branch. Since the last call to the MotoPed branch was a `next` call, we begin by making a `prev` call, finding the rightmost item in the MotoPed subtree satisfying the query. If we are able to find enough items in the MotoPed subtree, then we proceed to the next subtree; otherwise, we redistribute the necessary items to other subtrees. For instance, suppose that we were unable to find any additional items in the MotoPed subtree. Then we would search for 4 additional matches in the Skoot subtree. (In our actual implementation, we ensure even distribution by recursing on each subtree is round-robin fashion.)

By making calls using both `next` and `prev`, we ensure that do not make many unnecessary probes. In fact, it is possible to show that we use at most $2k$ probes to find a k-item diverse result set.

Scored Probing Algorithm. The scored probing algorithm works in much the same way as the unscored version. First, we call WAND to obtain a top-k scored list. Let θ be the smallest score in the list. By the definition of scored diversity, all items with score greater than θ must be kept. We arrange these items into a tree, based on their dewey ids, ignoring items with score exactly θ. We now use this tree to make decisions on where to probe, in much the same way as in the unscored case. Each probe returns the next (or previous) item matching our query and having score θ. The details are omitted here.

4 Experiments

We compared the performance of five algorithms in the unscored and the scored cases. `MultQ` is based on rewriting the input query to multiple queries and merging their result to produce a diverse set. `Naive` evaluates a query and returns all its results. We do not include the time this algorithm takes to choose a diverse set of size k from its result. `Basic` returns the k first answers it finds without guaranteeing diversity. `OnePass` performs a single scan over the inverted lists (Section 3). Finally, `Probe` is the probing version (Section 3). We prefix each algorithm with a "U" for the unscored case. and with an "S" for their scored counterparts. Scoring is achieved with additional keyword predicates in the query. Recall that all of our diversity algorithms are *exact*. Hence, all results they return are maximally diverse.

4.1 Experimental Setup

We ran our experiments on an Intel machine with 2GB RAM. We used a real dataset containing car listings from Yahoo! Autos. The size of the original cars relation was varied from 100K to 1M rows with a default value set to 100K. Queries were synthetically generated using the following parameters: Number of cars $\in [10K - 100K]$ (default $50K$); Number of predicates/query $\in [1, 5]$ (default none); predicate selectivity $\in [0, 1]$ (default 0.5); $k \in [1, 100]$ (default 10). Query predicates are on car attributes and are picked at random. We report the total time for running a workload of 5000 different queries. In our implementation, the cars listings were stored in a main-memory table. We built an index generation module which generates an in-memory Dewey tree which stores the Dewey of each tuple in the base table. Index generation is done offline and is very fast (less than 5 minutes for 100K listings).

Varying Data Size: Figure 2(a) reports the response time of `UNaive`, `UBasic`, `UOnePass` and `UProbe`. `UOnePass` and `UProbe` have similar performance and are insensitive to increasing number of listings.

Varying Query Parameters: Figure 2(b) reports the response time of our unscored algorithms. Once again, `UOnePass` and `UProbe` have similar performance. The main two observations here are: (i) all our algorithms

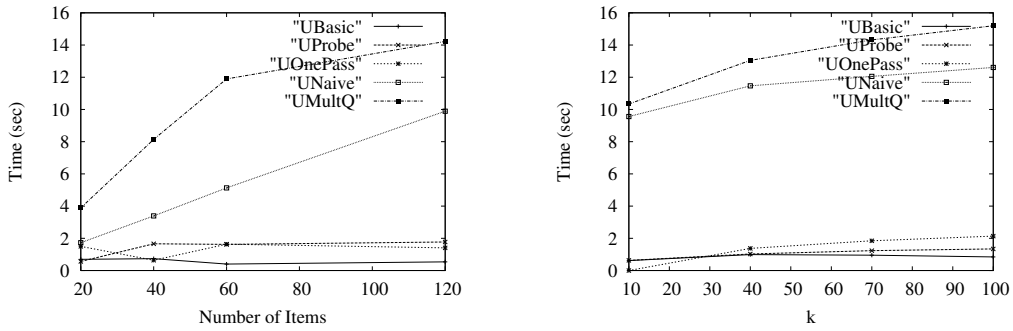

Figure 2: (a) Varying Data Size (Unscored), and (b) Varying k (Unscored)

outperforms the naive case which evaluates the full query and (ii) diversity incurs negligible overhead (over non-diverse `UBasic`) even for large values of k.

Figure 3(a) shows the response time of our unscored algorithms for different query selectivities. We grouped queries according to selectivity, measuring the average response time for each. `UOnePass` and `UProbe` remain stable with increasing selectivity, while `UNaive` is very sensitive since it retrieves all query results.

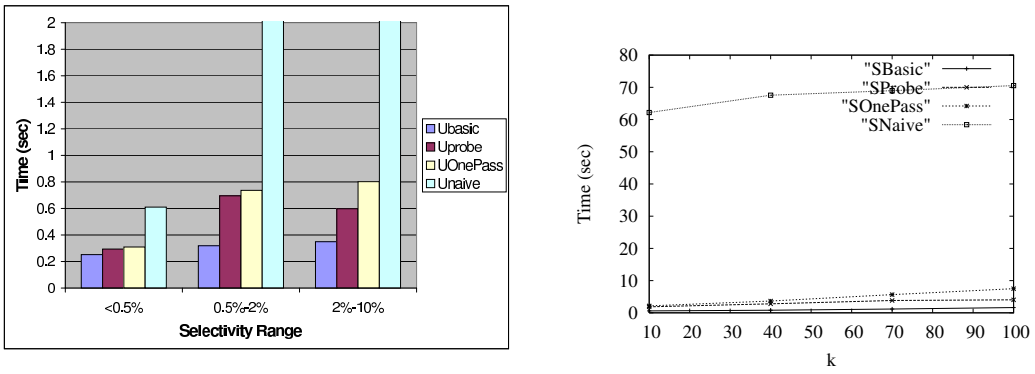

Figure 3: (a) Varying Q's Selectivity, and (b) Varying Data Size (scored)

Varying Query Parameters (scored): Figure 3(b) shows the response time of the scored algorithms as the number of results requested is varied. With increasing k, more listings have to be examined to return the k best ones. Thus, the response time of both `SOnePass` and `SProbe` increases linearly with k but as observed in the unscored case, the naive approach is outperformed. We note that varying query selectivity and data size is similar to the unscored case.

Experiments Summary: The naive approaches, `MultQ`, `UNaive`, `SNaive` are orders of magnitude slower than the other approaches. The most important finding is that returning diverse results using probing algorithms incurs negligible overhead (in the unscored case) and incurs very little overhead (in the scored case). Specifically, UProbe matches the performance of UBasic and SProbe comes very close to the performance of SBasic.

5 Related Work

The notion of diversity has been considered in many different contexts. Web search engines often enforce diversity over (unstructured) data results as a post-processing step [3, 11, 12]. Chen and Li [4] propose a notion of diversity over structured results which are post-processed and organized in a decision tree to help

users navigate them. In [8], the authors define the Précis of a query as a generalization of traditional query results. For example, if the query is "Jim Gray", its précis would be not just tuples containing these words, but also additional information related to it, such as publications, and colleagues. The précis is diverse enough to represent all information related to the query keywords. In this paper, we study a variant of diversity on structured data and combine it with top-k processing and efficient response times (no post-processing.)

In some online aggregation [7], aggregated group are displayed as they are computed and are updated at the same rate by index striding on different grouping column values. This idea is similar to our notion of equal representation for different values. However, in addition to considering scoring and top-k processing, we have a hierarchical notion of diversity, e.g., we first want diversity on `Make`, then on `Model`. In contrast, Index Striding is more "flat" in that it will simply consider (`Make`, `Model`) as a composite key, and list all possible (make, model) pairs, instead of showing only a few cars for each make.

6 Conclusion

We formalized diversity in structured search and proposed inverted-list algorithms. Our experiments showed that the algorithms are scalable and efficient. In particular, diversity can be implemented with little additional overhead when compared to traditional approaches.

A natural extension to our definition of diversity is producing weighted results by assigning weights to different attribute values. For instance, we may assign higher weights to MotoPeds and Skoots when compared to other scooter brands, so that the diverse results have more MotoPeds and Skoots. Another extension is exploring an alternative definition of diversity that provides a more symmetric treatment of diversity and score thereby ensuring diversity across different scores.

References

[1] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, J. Y. Zien. Efficient Query Evaluation Using a Two-Level Retrieval Process. CIKM 2003.

[2] N. Bruno, S. Chaudhuri, L. Gravano. Top-K Selection Queries Over Relational Databases: Mapping Strategies and Performance Evaluation. ACM Transactions on Database Systems (TODS), 27(2), 2002.

[3] J. Carbonell and J. Goldstein. The use of MMR, diversity-based reranking for reordering documents and producing summaries. SIGIR 98.

[4] Z. Chen, T. Li. Addressing Diverse User Preferences in SQL-Query-Result Navigation. SIGMOD 2007.

[5] R. Fagin. Combining Fuzzy Information from Multiple Systems. PODS 1996.

[6] L. Guo, F. Shao, C. Botev, J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. SIGMOD 2003.

[7] J. M. Hellerstein, P. J. Haas, H. J. Wang. Online Aggregation. SIGMOD 1997.

[8] G. Koutrika, A. Simitsis, Y. Ioannidis. Précis: The Essence of a Query Answer. ICDE 2006.

[9] F. Olken. Random Sampling from Databases. PhD thesis, UC Berkely, 1993.

[10] G. Salton and M. McGill. Introduction to Modern Information Retrieval. McGraw-Hill, 1983.

[11] D. Xin, H. Cheng, X. Yan, J. Han. Extracting Redundancy-Aware Top-k Patterns. KDD 2006.

[12] C-N Ziegler, S.M. McNee, J.A. Konstan, and G. Lausen. Improving Recommendation Lists Through Topic Diversification. WWW 2005.