Efficient Computation of Diverse Query Results

Erik Vee, Utkarsh Srivastava, Jayavel Shanmugasundaram, Prashant Bhat, Sihem Amer Yahia

Yahoo! Research
Sunnyvale, CA, USA
{erikvee,utkarsh,jaishan,pbhat,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 Honda cars and we can only display five results, we wish to return cars from five different Honda models, as opposed to returning cars from only one or two Honda 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.**

I. Introduction

Online shopping is increasing in popularity due to the large inventory of listings available on the Web.[1] 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 Yahoo! Autos user searching for used 2007 Honda cars. If we only have space to show five results, we would rather show five different Honda models (e.g., Honda Civic, Honda Accord, Honda Odyssey, Honda Ridgeline and Honda S2000) instead of showing cars from just one or two models. Similarly, if the user searches for 2007 Honda Civic cars, we would rather show 2007 Honda Civic cars in different colors rather than simply showing cars 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. However, this method does not scale to large data sets, where a query can return a large number of results. A 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 results [3], [11], [12]. Usually, $c \times k$ is much smaller than the total number of results, so it is more efficient than the previous method. 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, f 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 Honda "convertible"s (where "convertible" is a keyword search query), this method would issue a query to see if there are any Honda Civic convertibles, issue another query to see if there are any Honda Accord convertibles, issue another query to see if there are any Honda Ridgeline convertibles, 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., there are no Honda Civic, Honda Accord, or Honda Ridgeline convertibles).

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. As an illustration, there are only a few Honda S2000 cars, but we wish to include them in the results to show the full range of choices.

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-

[1]E.g., *shopping.yahoo.com, amazon.com, ebay.com*

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 II)
- Efficient one-pass (Section III) and probing (Section IV) algorithms for implementing diversity
- Experimental evaluation using Yahoo! Autos data (Section V)

II. 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.

A. 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 `att = value` and *keyword predicates* of the form `att C keywords` where `att` is an attribute of R and `C` stands for keyword containment. Given a relation R and a query Q, we use the notation $\textsc{Res}(R, Q)$ to denote the set of tuples in R that satisfy Q.

As an illustration, consider the `Cars` relation shown in Figure 1(a). The query `Make = 'Honda'` would return all the cars whose make is Honda. Similarly, the query `Description contains 'Low Mileage'` would return all the cars whose description contains the keywords 'Low miles', the query `Make = 'Honda' and Description contains 'Low miles'` would return all the cars that satisfy both the conditions, and the query `Make = 'Honda' or Description contains 'Low miles'` would return cars that satisfy at least one of the conditions.

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 disjunctive queries such as `Make = 'Honda' and Description contains 'Low miles'`, where a tuple may not satisfy all the predicates. Here, a weight can be assigned to each disjunct, and the score of a tuple can be defined to be a monotonic combination function of the weights of the predicates satisfied by the tuple [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. When the query is clear from the context, we simply refer to the score of a tuple t as $score(t)$.

B. Diversity Definition

Consider the database shown in Figure 1(a). If the user issues a query for all cars and we can only display three results, then clearly we wish to show one Honda and two Toyotas (or vice-versa). Similarly, if the user issues a query `Make = 'Honda'`, we wish to show different models of Hondas, e.g., the top relation in Figure 1(b) is more diverse than the bottom relation because it shows three different models of Hondas. However, if the user query is `Description contains 'Low miles'`, then the bottom relation is a good result because only Honda Civics satisfy the keyword search condition and this relation shows different colors of Honda Civics.

The key take-away from the above examples is that there is a priority ordering of attributes when it comes to defining diversity, whereby we wish to vary the values of higher priority attributes before varying the values of lower priority attributes. In our example, `Make` has a higher priority than `Model`, which has a higher priority than `Color`, which in turn has a higher priority than `Year`. Note that this ordering is domain-specific and can be defined by a domain expert (for instance, `Year` can be defined to have a higher priority than `Color`, if desired). This notion is captured 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 $\textsc{Sim}(x, y)$, 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 $\sum_{x,y \in S} \textsc{Sim}(x, y)$.

With an eye toward our ultimate goal, let us take a very simple similarity function: $\textsc{Sim}(x, y) = 1$ if x and y agree on the highest priority attribute, and 0 otherwise. It is not hard to see that by using this similarity measure, minimizing the all-pairs sum of similarities in our running example guarantees that we have equal numbers of Hondas and Toyotas (within one), so long as there are enough Hondas and Toyotas to display.

However, as mentioned above, we wish to diversify on not just the first attribute in the diversity ordering. For instance, if

Id	Make	Model	Color	Year	Description
1	Honda	Civic	Green	2007	Low miles
2	Honda	Civic	Blue	2007	Low miles
3	Honda	Civic	Red	2007	Low miles
4	Honda	Civic	Black	2007	Low miles
5	Honda	Civic	Black	2006	Low price
6	Honda	Accord	Blue	2007	Best price
7	Honda	Accord	Red	2006	Good miles
8	Honda	Odyssey	Green	2007	Rare
9	Honda	Odyssey	Green	2006	Good miles
10	Honda	CRV	Red	2007	Fun car
11	Honda	CRV	Orange	2006	Good miles
12	Toyota	Prius	Tan	2007	Low miles
13	Toyota	Corolla	Black	2007	Low miles
14	Toyota	Tercel	Blue	2007	Low miles
15	Toyota	Camry	Blue	2007	Low miles

(a)

Id	Make	Model	Color	Year	Description
1	Honda	Civic	Green	2007	Low miles
6	Honda	Accord	Blue	2007	Best price
8	Honda	Odyssey	Green	2007	Rare

Id	Make	Model	Color	Year	Description
1	Honda	Civic	Green	2007	Low miles
2	Honda	Civic	Blue	2007	Low miles
3	Honda	Civic	Red	2007	Low miles

(b)

Fig. 1. Example Database and Query Results

we show more than one Honda Civic in a result set, we wish to diversify their color. Thus, we define a *prefix with respect to* $\prec$ to be a sequence of attribute values, in order given by $\prec$, moving from highest to lower priority. For example, `Honda Odyssey` is a prefix for items 8 and 9 in Figure 1(a). On the other hand, neither `Blue 2007` nor `Toyota Green` are, since all prefixes must start with the `Make` attribute, and all attributes must follow in contiguous order.

We now capture the notion of diversifying at lower levels. 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 = `Honda Civic`, then minimizing $\sum_{x,y \in S_\rho} \text{SIM}_\rho(x, y)$ guarantees that we will not display two `Black` Honda Civics 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 $\prec_R$).

It can be shown that a diverse set of size k always exists.

We now generalize the above definition to the case where tuples can have scores. Intuitively, scored diversity always picks tuples with higher scores over tuples with lower scores (in the top-k results). However, if many tuples are tied for the lowest score (in the top-k results), then the lowest score tuples are picked in a diversity preserving way. Note that this approach generalizes both score-based ranking and unscored diversity: if every item has a unique score, it reduces to score-based ranking, while if every item has the same score, it reduces to unscored diversity. We can also achieve greater diversity by choosing a coarse scoring function (i.e., one that assigns the same score to many tuples).

In order to formally define scored diversity, we need one further concept. Given relation R and query Q, let $maxval = \max_{T \in \mathcal{R}_k} score(T)$, where $score(T)$ is the sum of the scores of tuples in T. Then define $\mathcal{R}_k^{score}$ to be the set of sets in $\mathcal{R}_k$ whose total score is $maxval$. Then we can define scored diversity analogously to unscored diversity by replacing $\mathcal{R}_k$ with $\mathcal{R}_k^{score}$ in Definition 2.

C. Impossibility Results

We now show that we cannot use traditional IR scoring to produce (unscored or scored) diverse results. The intuition here is that the IR score of an item depends only on the item and possibly statistics from the the entire item corpus, but diversity depends on the other items in the query result set.

The class of IR systems that we consider are those based on inverted lists: 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 usually ordered by their score so that top-k queries can be handled efficiently.

Given a query Q, we find the lists corresponding to the attributes 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 list is usually aggregated from the per-list scores, and efficient algorithms such as the Threshold Algorithm [5] assume that the aggregation function is monotone.

Without loss of generality, we only consider systems with value/keyword -dependent score (global scores can be simulated by assigning the same score to an item for every value/keyword). For any attribute/keyword A, denote its scoring function by $\textsc{Score}_A(\cdot)$. Let f be a monotonic aggregating function, which takes a set of scores and outputs an aggregated score. Given a query Q that uses attributes/keywords $A_1, ..., A_\ell$, we may choose weights $w_{A_1}, ..., w_{A_\ell}$ based on Q (and k). The aggregated score of item i from the database is then $f(w_{A_1}\textsc{Score}_{A_1}(i), ..., w_{A_\ell}\textsc{Score}_{A_\ell}(i))$. The top-$k$ algorithm outputs the k items with the highest aggregated score. We call such systems Inverted-List Based IR Systems, and we should that such systems cannot be used to output an *unscored* diverse result set.

Theorem 1: 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.

Proof: Consider the database shown in Figure 1(a), and suppose there is an Inverted-List Based IR System that produces diverse result for this data. First, consider the top-8 results for the query `Year = 2007`. Since it must be diverse, this list must include every Toyota. On the other hand, diversity implies that exactly one of the Honda Civics can be in that top-8 list. Let h_1 be that Honda Civic. Hence, the inverted list for `2007` must rank the Toyotas and h_1 in its top 8. Also notice that no other tuple in that top 8 can satisfy `Description contains 'mileage'`.

Likewise, we see that the inverted list for `'mileage'` must also place every Toyota somewhere in its top 8, and exactly one Honda Civic can be in those top 8 spots; call this Honda Civic h_2. Further, no other tuple in that top 8 can satisfy `Year = 2007`.

Now, consider the top-6 list produced by the query `Year = 2007 AND Description contains 'mileage'`. By the monotonicity condition on f, the tuples for `Toyota` can be beaten by at most h_1 and h_2. Hence, the top-6 list must contain all of these 4 Toyotas and at most 2 Hondas, violating diversity, a contradiction. ■

The above result seems to indicate that traditional IR systems will have difficulty in producing diverse result sets, even in the unscored setting. However, they do not rule out the possibility completely. 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.

III. One-pass Algorithms

We first introduce our data structures and discuss an unscored and a scored version of our one-pass algorithm.

A. Data Structures

Each tuple is uniquely identified by the concatenation of the values of its attributes. By making sure that those values are concatenated in the order specified in the diversity ordering, each tuple would reflect the diversity ordering. For example, the tuple with `Id` value 2 in the `Cars` relation in Figure 1 is uniquely represented by the value *Honda.Civic.Blue.2007.'Low miles'*. By assigning a distinct integer identifier to each value in an attribute, this representation becomes more compact. The `Index` column of Figure 2(b) shows how this is done for the `Cars` relation. This value assignment reminds us of the Dewey encoding as done in XML query processing [6] and can be represented in a *Dewey tree* as illustrated in Figure 2(a). 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.

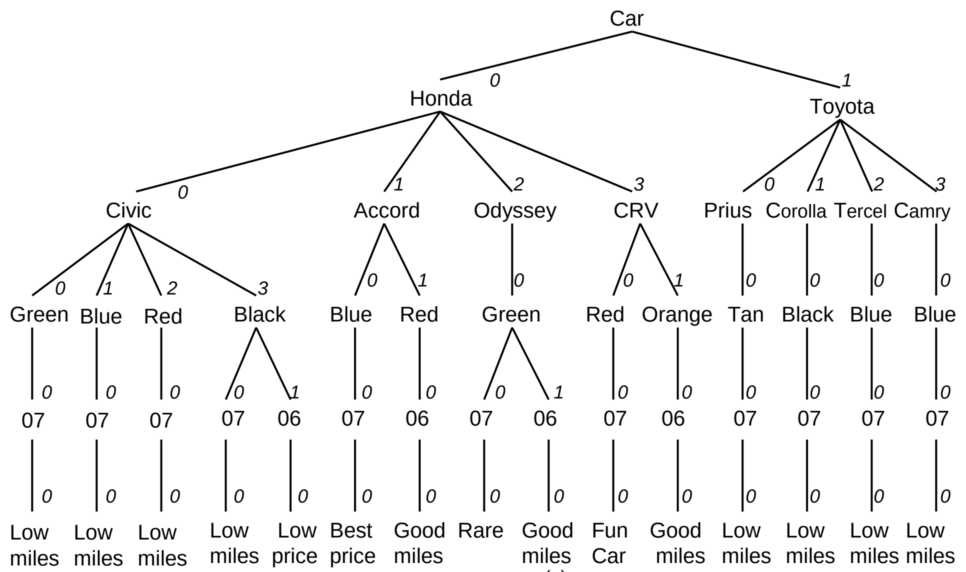
B. Basic operations

We will often need to iterate over a set of inverted lists. Conceptually, we can think of these lists are being merged into one list, `mergedList`, and making calls to `mergedList.next(id)`, which simply returns the smallest dewey ID in `mergedList` that is greater than or equal to `id`.

We also define `mergedList.next(id, RIGHT)`, which is needed to *move backwards* in `mergedList` (only used in Section IV). This call returns the *largest* dewey ID that is less than or equal to `id`. Calls to `mergedList.next(id, LEFT)` behave as in the ordinary `next`. We also need to extend our results to handle scored items. For a given score θ, calls to `mergedList.next(id,` θ`)` return the smallest dewey ID greater than or equal to `id` that has score at least θ (and that also appears in the merged list). Calls using `RIGHT` are defined similarly.

Our implementation of `next(·)` uses the same techniques as the WAND algorithm of [1]. WAND is an efficient method for obtaining top-K lists of scored results, without explicitly merging the full inverted lists. We use it as a starting point in our algorithms.

We will also be iterating through branches of the dewey tree. As such, we need the key operator, `nextId`. The value `nextId(id, level, LEFT)` is `id` with its `level`-th entry increased by 1 and all entries beyond the `level`-th set to 0. So, for example, `nextId(0.3.1.0.0, 2, LEFT)` returns 0.4.0.0.0, even though it is not actually in the tree. For convenience, we assume that no dewey entry is greater than 9. Then the value of `nextId(id, level, RIGHT)` is `id` with its `level`-th entry *decreased* by 1 and all entries beyond the `level`-th set to 9.



(a)

Id	···	Index
1	...	0.0.0.0.0
2	...	0.0.1.0.0
3	...	0.0.2.0.0
4	...	0.0.3.0.0
5	...	0.0.3.1.0
6	...	0.1.0.0.0
7	...	0.1.1.0.0
8	...	0.2.1.0.0
9	...	0.2.1.1.0
10	...	0.3.0.0.0
11	...	0.3.1.0.0
12	...	1.0.0.0.0
13	...	1.1.0.0.0
14	...	1.2.0.0.0
15	...	1.3.0.0.0

(b)

Fig. 2. Indexing the `Cars` Relation

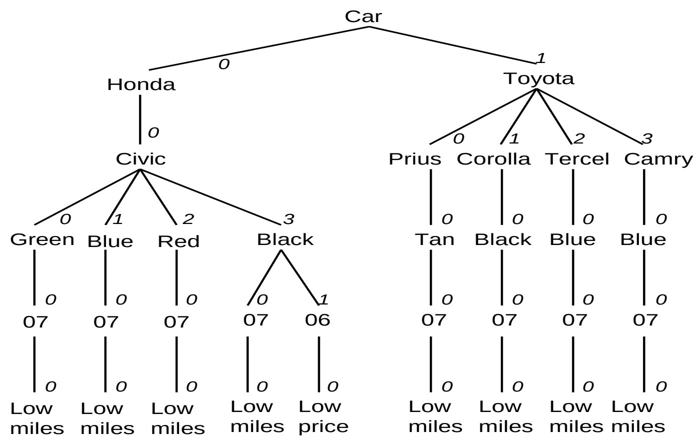


Fig. 3. Tree of nodes satisfying query Q

C. One-Pass Unscored Algorithm

The pseudocode is given in Algorithm 1. The core idea of this algorithm is to explore buckets of distinct values sequentially and ensure each time that k answers are found. For example, given query Q which looks for descriptions with `'Low'`, we are left with the tree in Figure 3. Assuming $k = 3$, a balanced result would return (at least) one car of each make from the database. This corresponds to a Honda and a Toyota. In order to produce that output, Algorithm 1 uses an inverted list to retrieve all cars matching the query and scans them in the order in which they appear (left-to-right) in Figure 3. It first selects the first two Civics, then it determines that it only needs to pick one from the next level, so, it picks one Accord. Note that if there were no more cars, this set would constitute a diverse set. However, since there are more cars that satisfy the query, the algorithm decides that it should pick at most one entry from the next set to make the previous set diverse, in which case, it chooses the first-appearing Odyssey. At this stage, the algorithm has to decide which entry to prune from the partial result set and prunes one of the Civics since this will make the set diverse (one Civic, one Accord and one Odyssey). It then decides how many entries in the next level (i.e., Toyota), it needs to retrieve and decides to select one entry. In this case, it picks the next Toyota it sees and it skips over all other Toyotas since none of them will make the result set diverse. Finally, it stops and returns the Toyota and two of the already-selected Hondas(after pruning one of the three Hondas). Note that if there had been another car make (say, Chevrolet), the algorithm would decide to pick one entry there and would finally return one Honda, one Toyota and one Chevrolet.

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 `mergedList` is large, this can be a significant saving over the naive algorithm.

D. One-pass Scored Algorithm

The main difference with the unscored algorithm lies is what parts of the tree we can skip over. In the former algorithm, it was easy to determine the smallest ID that would not be removed on the next call to `remove`(). In the scored case, we must add any item whose score is strictly greater than the current `minScore` (i.e. the smallest score in our result set).

Algorithm 1 Unscored one-pass Algorithm

```
Driver Routine:
 1: id = mergedList.next(0)
 2: root = new Node(id, 0)
 3: id = mergedList.next(id)
 4: while (root.numItems() < k  && id ≠ NULL)
 5:     root.add(id)
 6:     id = mergedList.next(id+1)
 7: while (id ≠ NULL)
 8:     root.add(id)
 9:     root.remove()
10:     skipId = root.getSkipId()
11:     id = mergedList.next(skipId)
12: return  root.returnResults()
```

However, we can find the smallest ID that would immediately be removed, given that its score is no greater than minScore. Hence, we replace line 11 of the unscored one-pass algorithm with the line

id = mergedList.next(id+1, skipId, root.minScore)

The semantics of the above line is to return the smallest id greater than or equal to id+1 such that either (1) score(id) >root.minScore, or (2) score(id)≥root.minScore, and the return id is greater than skipId. With the above exception, the algorithm proceeds as before.

IV. Probing Algorithms

A. Unscored Probing Algorithm

The main idea of the probing algorithm is to traverse the available levels many times by picking one item at a time until K answers are found. Again, consider query Q which looks for cars with 'Low' in the description (shown in Figure 3). Assuming $k = 3$, the algorithm would first pick the first Honda Civic, then the *last* Toyota. (We use a bidirectional probing algorithm, which searches both left and right through the tree.) It then looks for a car make "between" Honda and Toyota. (See Figure 3.) Since there are not any, it continues to pick the next car which guarantees diversity, in this case, the first Toyota Prius. At this stage, the algorithm is done.

We first walk through several steps of the probing algorithm in the unscored case. The main routine (Algorithm 2) simply makes repeated calls to getProbeId(), which returns potential IDs to add. The driver then calls next on the mergedList, and adds the result to our diversity data structure (given in Algorithm 3.) It repeats this until we have a top-K list, or there are no more results.

For convenience, we will say that an ID, id *belongs to* a node, node, written id ∈ node, if the node corresponding to id lies in the subtree rooted at node. For example, 1.2.0.0.0 belongs to the node labeled "Toyota" in Figure 3.

In its initialization step, the algorithm first calls id = mergedList.next(0, LEFT). This simply returns the first id ≥ 0 (moving from left to right) that appears in mergedList. In our case, id is 0.0.0.0.0. We initialize root = new Node(id, 0, LEFT). and pick 5 nodes,

Algorithm 2 Driver for Unscored Probing Algorithm

```
1: id = mergedList.next(0, LEFT)
2: root = new Node(id, 0, LEFT)
3: (probeID, dir) = root.getProbeId()
4: while (root.numItems() < k && probeId ≠ NULL)
5:     id = mergedList.next(probeId, dir)
6:     root.add(id, dir)
7:     (probeID, dir) = root.getProbeId()
8: return root.getItems()
```

Algorithm 3 Data Structure for Unscored Probing

```
Initializer: (new Node(nid, lev, dir))
1: Initialize id = nid, level = lev and children to empty.
2: if (isLeaf(level)) then mark as DONE
3: else
4:     edge[dir] = nextId(id, level+1, dir)
5:     nextDir = toggle(dir)
6:     edge[nextDir] = nextId(id, level+1, nextDir)
7:     children.add(newNode(id, level+1, dir))

Get probeId: (getProbeId())
1: if (edge[LEFT] ≤ edge[RIGHT]) then
2:     return (edge[nextDir], nextDir)
3: while (there are children that are not marked DONE)
4:     minChild = child with the minimum number of items,
       ignoring children that are marked DONE;
5:     return minChild.getProbeId() if it is not NULL
6: mark as DONE
7: return NULL

Adding a result: (add(id, dir))
1: if (marked as DONE) then return
2: child = child in children corresponding to id
3: if (child ≠ NULL) child.add(id, dir)
4: else children.add(new Node(id, level+1, dir))
5: if (edge[LEFT] ≤ edge[RIGHT])
6:     edge[dir] = nextId(id, level+1, dir)
7:     nextDir = toggle(dir)
```

one for each level of the Dewey tree. Each of these nodes also initializes edge[LEFT]. For instance, the root node initializes edge[LEFT] = 1.0.0.0.0, while its child initializes edge[LEFT] = 0.1.0.0.0. Each node also initializes edge[RIGHT] to some large Dewey ID, which is conceptually the largest possible ID belonging to that node (line 6 of the initializer). Suppose we know that no Dewey ID has an entry greater than 9. Then we set root.edge[RIGHT] = 9.9.9.9.9, we set edge[RIGHT] = 0.9.9.9.9 for its child, we set edge[RIGHT]=0.0.9.9.9 for its grandchild, and so on. Think of these edge values as left and right boundaries, where all unexplored branches of a node lie between the values. In fact, we have the following invariant:

- Whenever id ∈ node, either id belongs to some child of node in our data structure, or node.edge[LEFT]≤id≤ node.edge[RIGHT].

Next, the call to root.getProbeId() will return (9.9.9.9.9, RIGHT) (line 3 of the driver). Hence, the call to mergedList.

`next`(9.9.9.9.9, `RIGHT`) returns the largest `id` $\leq$ 9.9.9.9.9 (moving from right to left) that appears in `mergedList`. In our example, `id` is 1.3.0.0.0. When this result is added to the potential answers, the values `edge`[`RIGHT`] are set to the largest IDs possibly belonging to each node, while the values of `edge`[`LEFT`] are set to be the smallest IDs possible (unless the value has already been set). So, `root.edge`[`RIGHT`] = 0.9.9.9.9, while its right child (labeled "Toyota") sets `edge`[`RIGHT`] = 1.2.9.9.9 and `edge`[`LEFT`] = 1.0.0.0.0.

For the third probe (and the second call to `getProbeId`(), line 7 of Algorithm 2), we return (1.0.0.0.0,LEFT). Hence, `id` is subsequently set to 1.0.0.0.0. When we add this result, `root.edge`[`LEFT`] is set to 2.0.0.0.0. For the first time, we have `root.edge`[`LEFT`] > `root.edge`[`RIGHT`], indicating that every child of `root` has at least one result added to it. Notice that we have added two results to the node labeled "Toyota," and only one to the node labeled "Honda." However, it is not hard to see that such an imbalance cannot be greater than one item, since the next iteration will necessarily add to the child with the fewest results.

On our next call to `root.getProbeId`() does not simply return `edge`[$\cdot$], since `root.edge`[`LEFT`] > `root.edge`[`RIGHT`]. (We think of this as `root` being in the second phase of exploration.) Instead, it sets `minChild` to be the child with the fewest results (in this case, the child labeled "Honda") and calls `getProbeId`() on `minChild` (line 5 of `getProbeId`()). Notice that `minChild` will explore from the `RIGHT` (since the last real result added to it came from the `LEFT`). The next result added to the data structure will thus be 0.2.1.0.0.

The advantages of bidirectional exploration. At this point, we begin to see the advantages of calling `next` moving in both directions, rather than just left to right. Notice that in the full Dewey tree of possible answers, the node "Honda" has just a single child. In our calls using `LEFT` and `RIGHT`, we found this automatically. However, an algorithm using only calls to `next`($\cdot$, `LEFT`) would need to check whether "Honda" had additional children, wasting a call to `next`. In extreme cases, (say, if "Civic" had only one child), such an algorithm would waste even more calls to `next`.

On the other hand, we guarantee that calls to `next` always result in an `id` that "stays in the final subtree." A little more formally, we maintain the following invariant:

- Let `node` be some node in our data structure, and suppose during the execution of the algorithm, we call `node.getProbeId`(), returning (`probeId`,`dir`). Then we have `mergedList.next`(`probeId`, `dir`)$\in$ `node`.

Unlike the one-pass algorithm, every time a call to `next` is made, it results in an `id` that is part of our diverse set, unless it has already been added to the structure. It is never necessary to remove a result we have added. In fact, we have the following stronger result, whose proof appears in the full paper.

Theorem 2: The unscored probing algorithm given in Algorithms 2, 3 makes at most $2k$ calls to `next`.

B. Scored Probing Algorithm

The first stage of the algorithm (pseudocode in Algorithm 4) is to call WAND (or any scoring algorithm) to obtain an initial top-K list. Let θ be the score of the lowest-scoring item in the top-K list returned. Diversity is only guaranteed among items whose score is θ. The difficulty comes from not knowing the exact value of θ.

Algorithm 4 Driver for Scored Probing

```
Driver Routine:
 1: Run WAND to obtain a top-K list, List.
 2: Let $\theta$ be the lowest score in that list.
 3: Let maxid be the item in List with the highest score.
 4: root = new Node(maxid, 0, MIDDLE)
 5: Foreach item $\in$ List such that score(item) > $\theta$:
 6:     root.add(item, MIDDLE)
 7: (probeID, dir) = root.getProbeID()
 8: while ( root.numItems() < k  && probeID $\neq$ NULL )
 9:     currID = mergedList.next(probeID, dir, $\theta$)
10:     root.add(currID, dir)
11:     (probeID, dir) = root.getProbeID()
12: return  root.returnResults()
```

It is the initial insertions that make the algorithm more complicated. As we can see on line 6, each of the items inserted initially is marked `MIDDLE`, indicating that it does not give any information about `edge`[`LEFT`] or `edge`[`RIGHT`]. In the unscored case, we knew that every `id` we obtained was a useful result to add (unless it had already been added).

However, in the scored case, we may potentially obtain an `id` that is not useful, in the following way. Many items with scores higher than θ have been added already, in a non-regular fashion. Thus, when we first encounter an item with score exactly θ, adding it may make our result set less diverse; unfortunately, since we do not know how many items have score θ, we cannot know immediately whether adding a given item helps or hurts diversity. Thus, many items we encounter are first cached as a *tentative* ID. (If the item is known to be useful, it is added normally.) When the algorithm later calls `getProbeId`(), it checks whether any cached IDs would be helpful; since we know more about the distribution of items with score θ, we slowly move tentative items into the 'useful pile.' If an item is helpful, it is returned (along with direction `MIDDLE`, indicating that the call to `next`(`probeId`, `MIDDLE`) should simply return `probeId` with no additional work).

It is possible to show that the number of tentative items is never more than $2k'$, where k' is the number of items with score strictly greater than θ. In fact, Theorem 2 applies even to the scored algorithm, guaranteeing that we make at most $2k$ calls to `next` (ignoring calls with `MIDDLE`).

The asymptotic running time of this algorithm is not impacted by maintaining these tentative values. However, in order to find the child with the fewest results (corresponding to line 4 of `getProbeId`() in the unscored version), we must maintain a heap. Hence, the worst-case running time for each operation is $O(d \lg k)$.

Parameter	Values (default in bold)
Number of cars	$[10K - 100K]$ - **50K**
Queries	Number of predicates: $[1 - 5]$ - None Predicate selectivity $[0 - 1]$ - **0.5** Number of results (k): $[1 - 100]$ - **10**

Fig. 4. Experimental Parameters

V. 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 III). Finally, Probe is the probing version (Section IV). 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.

A. 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 by using the parameters in Figure 4 (the default value for each parameter is shown in bold). 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 5 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 6 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 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 7 shows the response time of our unscored algorithms for different query selectivities. We grouped queries according to selectivity and measured the average response time in each group. UOnePass and UProbe remain stable with increasing selectivity while UNaive is very sensitive to the selectivity since it retrieves all query results.

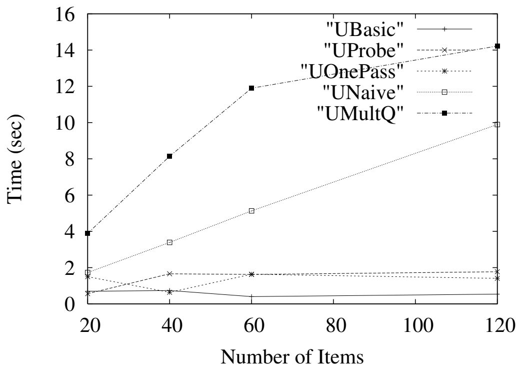


Fig. 5. Varying Data Size (Unscored)

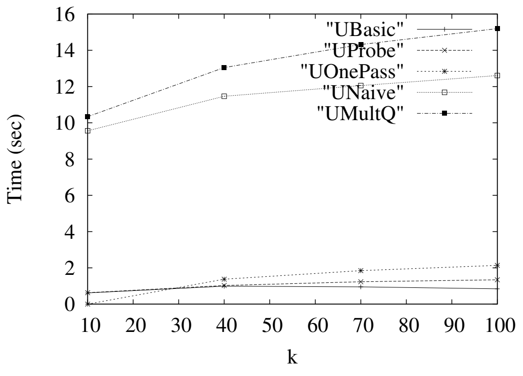


Fig. 6. Varying k (Unscored)

Varying Query Parameters (scored): Figure 8 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 does not incur any 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.

VI. 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

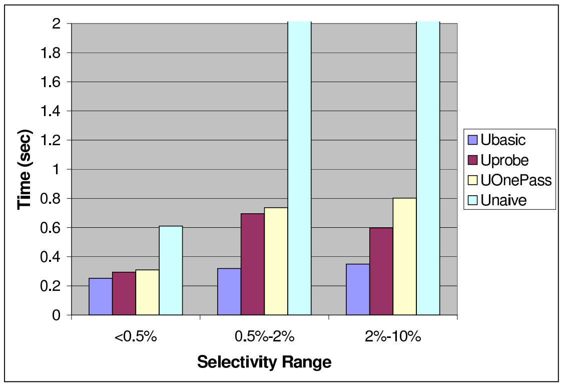


Fig. 7. Varying Q's Selectivity (Unscored)

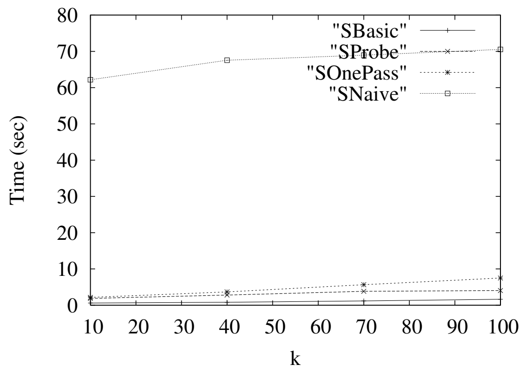


Fig. 8. Varying k (Scored)

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.

VII. 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 Hondas and Toyotas when compared to Teslas, so that the diverse results have more Hondas and Toyotas. 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.